Universidad Nacional Autonóma de México

Facultad de Ciencias

Una implementación de máquinas de reducción

TESIS

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

Presenta:

Alexis Arturo Rodríguez Hernández

DIRECTORA DE TESIS:
Dra. en Ciencia e Ing. de la Computación
Karla Ramírez Pulido

Ciudad Universitaria, CD. MX.
2023

Agradecimientos

A todas las hermosas amistades que forjé en el transcurso de mi vida universitaria por su apoyo y compañía, a mis profesores por su paciencia y conocimientos, a mi tan querida Universidad que me acogió en sus aulas por tantos años, y sobre todo a mi amada familia.

Índice general

Introducción

Si se desea programar es posible enfrentarse alguna vez con la noción de *abstracción funcional*. Este concepto ha usado varios nombres dependiendo del paradigma donde se busque; ya sea función, subrutina, método o proceso siempre se trata de una serie de pasos a seguir utilizando una entrada para después entregar un resultado [1]. Estudiar, comprender y analizar esta noción es una tarea que valdría la pena intentar alguna vez dado su gran alcance teórico. Este concepto ha sido objeto de muchas discusiones a lo largo de la historia de las Ciencias de la Computación desde que fue introducido en la teoría matemática con el cálculo lambda de Church y la lógica combinatoria de Curry y Schönfinkel. Analizando la abstracción funcional se han dado avances en áreas clave de las Ciencias de la Computación tales como:

1. **Diseño de lenguajes de programación:** Varias nociones relacionadas al tipado que se encuentran en los lenguajes de programación modernos fueron inspiradas por los mecanismos de tipado encontrados al formalizar la abstracción funcional, un ejemplo notable de esto es la noción de *polimorfismo*.

2. **Semántica de los lenguajes de programación:** Una de las escuelas de pensamiento predominantes en esta área es la *semántica denotacional*, en la cual una expresión inspirada en una abstracción funcional se usa para dar significado a un programa.

3. **Computabilidad:** Un uso clásico de las abstracciones funcionales es el de estudiar las limitaciones teóricas de los sistemas formales para describir cálculos. De hecho el primer resultado en computabilidad es uno referente a la relación entre el cálculo lambda y las Funciones Recursivas de Kleene, ambos temas fuertemente relacionados al concepto de abstracción funcional.

Ligado a la abstracción funcional se encuentra el concepto de *aplicación de función*, uno puede pensar intuitivamente en la aplicación como el proceso mediante el cual uno *llama* o *aplica* una función, método o procedimiento a un término (comúnmente llamado *argumento* de la función) en un lenguaje de programación. Estos dos conceptos toman forma en

la teoría matemática de Church usando expresiones lambda y reducciones de distintos tipos; que corresponden a la abstracción funcional y a la aplicación de función respectivamente.

Reducir una expresión de distintas formas da sentido semántico a un programa porque el simple hecho de elegir una subexpresión sobre otra al momento de hacer la reducción puede cambiar por completo el propósito y el significado de un programa e incluso si podrá terminar algún día de ejecutarse, ¿cómo es que se piensa?, ¿qué debería hacer?, ¿existen mejores formas de hacerlo o no?, ¿se producirá un resultado no deseado? El estudio de todas estas cuestiones podría llevarnos a dar con nuevas formas de resolver un problema, nuevos enfoques para atacarlo, desentrañar todo lo que las abstracciones funcionales tienen para ofrecer y acercarnos cada día más a una teoría computacional más completa.

Las máquinas abstractas son *máquinas* porque permiten ejecutar un programa paso a paso; son *abstractas* porque omiten muchos detalles de las máquinas reales (hardware). Las máquinas proveen un estado de abstracción intermedio para la comprensión de un lenguaje, esto es, cierran la brecha entre el alto nivel de un lenguaje de programación y el bajo nivel de una máquina real, las instrucciones de una máquina abstracta están hechas a medida para las operaciones particulares requeridas para implementar un lenguaje fuente o un grupo de lenguajes pertenecientes a un paradigma específico. Desde un punto de vista pedagógico, las máquinas simplifican la presentación y enseñanza de los principios para la implementación de un lenguaje de programación [3]. Adicionalmente a todas sus ventajas prácticas las máquinas abstractas son útiles porque permiten probar varios aspectos teóricos de los lenguajes, su corrección (*correctness*) y sus transformaciones [4, 5]. En este trabajo se implementarán algunas de las máquinas más importantes para el estudio e implementación de lenguajes de programación que siguen el paradigma funcional.

Las siguientes máquinas serán implementadas en este trabajo:

1. **Máquina SECD**
   Creada por Peter Landin en su artículo *The Mechanical Evaluation of Expressions* [30] esta máquina fue la primera en ser creada específicamente para el cálculo lambda visto como un lenguaje de programación, la máquina SECD (***S****tack,* ***E****nvironment,* ***C****ontrol,* ***D****ump*) es especialmente importante porque dio pauta al desarrollo subsecuente de muchas máquinas abstractas para lenguajes de programación funcional, es el punto de partida de algunos cursos universitarios.

2. **Máquina de Krivine**

"*Esta máquina utiliza una reducción de cabeza débil para ejecutar el cálculo lambda, lo cual significa que la redex*[1] *activa debe estar al comienzo del λ-término. Por lo tanto, la evaluación se detiene si no hay redex a la cabeza del λ-término. De hecho, reducimos a la vez una cadena completa* $\lambda x_1 ... \lambda x_n$*. Por lo tanto, la ejecución también se detiene si no hay suficientes argumentos*" [2].

Es importante notar que la máquina es *perezosa* no en el sentido estricto de los lenguajes de programación, esto es, evaluar las expresiones a lo más una vez, sino más bien evalúa las expresiones hasta cierto punto y solo cuando es necesario [1].

3. **Máquina CEK**
   Desarrollada por Matthias Felleisen y Dan Friedman esta máquina actúa sobre el cálculo lambda y provee una estrategia para escribir intérpretes, que además de ser eficiente tiene varias propiedades útiles incluyendo su facilidad para añadir rasgos más complejos a un lenguaje cómo lo son las continuaciones, es fácil detener al intérprete para obtener la pila de ejecución actual y es sencillo introducir hilos [26].

Las máquinas pueden ayudarnos a mejorar la comprensión que se tiene de la reducción y de las abstracciones funcionales en general. Todas y cada una de ellas fueron creadas para estudiar cómo es que reaccionaron ante una expresión, cómo debe reducirla, cuáles estructuras auxiliares debe utilizar en su tarea y cuál estado es el siguiente. Las máquinas abstractas como la CEK (llamada así por sus tres componentes: control, ambiente y continuaciones respectivamente) y la de Krivine son sistemas de transición de estado que representan el núcleo de la implementación real de un lenguaje de programación. Por otro lado, el análisis semántico de un programa se dedica a aproximar propiedades intencionales de estas máquinas, como el orden de reducción o la complejidad en tiempo y espacio, de forma segura mientras ejecutan un programa. Resulta natural entonces, el querer derivar un análisis sistemático de éstas y su comportamiento [6]. Las máquinas abstractas han sido usadas efectivamente como intermediarias o como arquitecturas de bajo nivel apropiadas para ayudar en la implementación seria de una gran variedad de lenguajes de programación, incluyendo los lenguajes pertenecientes a los paradigmas funcional, imperativo y lógico. Facilitan la portabilidad, las optimizaciones y la generación nativa de código máquina. Su estructura simple las hace apropiadas para el análisis y la experimentación [13].

[1] Del inglés ***red****ucible* ***ex****presion* traducido como *expresión reducible*.

En este trabajo de titulación se implementan las máquinas abstractas usando el lenguaje de programación funcional Haskell[2] debido a que es un lenguaje de programación funcional puro y estás máquinas han sido pensadas usando cálculo lambda (una teoría puramente funcional).

Algunas de la principales caracteristicas de Haskell son [14]:

1. **Fuertemente tipificado**. Cada expresión en el lenguaje tiene un tipo que se determina en tiempo de compilación.

2. **Inferencia de tipos**. Haskell determina el tipo de una expresión sin que explícitamente se le indique en el código, como por ejemplo en el siguiente código:

   ```
   a = "Hello"
   b = " World"
   c = a ++ b
   ```

   Haskell es capaz de operar con las variables determinando sus tipos automáticamente, a diferencia del siguiente código Java:

   ```
   String a = "Hello";
   String b = " World";
   String c = x + y;
   ```

   Como se puede ver es necesario indicarle a Java que la variables a,b y c son de tipo *String*.

3. **Evaluación perezosa**. Una estrategia que retrasa la evaluación de una expresión hasta que su valor sea necesario.

4. **Puramente Funcional**. Una función que asocia cada posible valor de entrada con uno de salida y nada más, es llamada pura. En particular, llamarla no tiene efectos secundarios y el resultado arrojado no depende más que de sus parámetros. Un lenguaje de programación puede ser llamado *puramente funcional* si la evaluación de expresiones es pura [17].

[2]No es el propósito de este trabajo dar una introducción completa a Haskell por lo que suponemos que el lector está familiarizado con el lenguaje. En caso contrario referimos la siguiente bibliografía introductoria [15, 16].

El presente trabajo contiene conceptos importantes para el diseño de lenguajes de programación, por ejemplo abtracciones funcionales, reducciones, cerraduras, ambientes entre otros. Estas resultan interesantes y útiles para los estudiantes de la licenciatura de Ciencias de la Computación que quieren ahondar en dichos temas.

En el capítulo 1 desarrollaremos las nociones básicas referentes al cálculo lambda mostrando su sintaxis y semántica tanto estática como dinámica. En los capítulos 2 a 4 mostraremos cada una de las máquinas anteriormente descritas a profundidad, definiendo su semántica operacional, sus estados y las transiciones entre estos estados. En el capítulo 5 implementaremos cada una de las máquinas mediante el lenguaje de programación funcional Haskell. Por último se tiene el capítulo de Conclusiones y trabajo futuro.

Parte I

Marco Teórico

Capítulo 1

El Cálculo Lambda puro

A inicio de la década de 1930 Alonzo Church desarrolló la notación *lambda* a partir de su intento de formalizar los fundamentos de las matemáticas; lamentablemente el sistema original resultó ser inconsistente[1] como método de deducción lógica al ser susceptible a la paradoja descubierta por Kleene y Rosser [19]. Sin embargo, en 1935 Church depuró el sistema original para conservar únicamente la parte referente al cómputo de aplicación de funciones [36], y poco tiempo después probó junto con Rosser que este nuevo sistema era consistente al probar el Teorema de Church-Rosser [37], este teorema garantiza que todas las expresiones lambda reducibles tienen una forma normal única por lo que no es posible derivar una contradicción.
En este capítulo describiremos la relación que tiene el cálculo lambda con nuestra noción de *función* en los lenguajes de programación, su sintaxis básica y algunas convenciones que nos serán útiles para su fácil lectura y escritura.

1.1. Funciones

En teoría de conjuntos es común que una función sea representada por una gráfica. La gráfica de una función la define por su comportamiento de entradas y salidas; por ejemplo, una función de un solo argumento es representada por un conjunto de pares donde el primer componente de cada par especifica el argumento y el segundo componente especifica el resultado correspondiente. Desde esta perspectiva la función en los números naturales que

[1] La consistecia se refiere a la propiedad de que no es posible derivar una contradicción dentro del sistema, esto es, derivar una afirmación y su negación al mismo tiempo. La consistencia es una propiedad importante en cualquier sistema formal ya que garantiza que el sistema no producirá resultados absurdos o contradictorios.

suma sus dos argumentos es representada como:

$$\{((0,0),0),((0,1),1),...((1,0),1),((1,1),2),...\}$$

o de forma más general:

$$\{((m,n),p)|m,n \in \mathbb{N}, p = n+m\}$$

La noción de que dos funciones son iguales si tienen la misma gráfica es llamada igualdad *extensional* [1].
Sin embargo, desde el punto de vista de la Ciencias de la Computación esta representación no es muy útil. Usualmente estamos interesados en *cómo* una función evalúa su resultado, más que en *qué* es lo que evalúa.
Tomemos como ejemplo todas las funciones de ordenamiento. Ya que todas toman un conjunto desordenado y lo ordenan todas tienen la misma gráfica y son (extensionalmente) iguales, aún así se ha dedicado una gran parte de la literatura de la Ciencias de la Computación al análisis y definición de diferentes algoritmos de ordenamiento, esto es por las diferencias que cada algoritmo muestra en su procedimiento al realizar el ordenamiento. El uso del término *algoritmo* en la oración pasada es clave; deberíamos definir a las funciones usando una regla, que diga *cómo* se evalúa el resultado, en lugar de su gráfica. En este esquema, dos funciones son iguales si ambas se definen usando las mismas reglas o sus equivalentes; esta forma de igualdad es llamada igualdad *intensional*. El cálculo lambda provee un formalismo para expresar funciones como *reglas de correspondencia* entre argumentos y resultados [1].

1.2. Sintaxis

Daremos una definición inductiva de términos lambda, para ello necesitamos definir un conjunto infinito de identificadores que llamaremos *variables*. Denotaremos los elementos de este conjunto usando letras minúsculas [18].

Definición 1: $\mathbb{X}$ Variables

$$\mathbb{X} = a|b|c| \ ... \ |x|y|z| \ ...$$

La clase de términos lambda consiste de palabras construidas del siguiente alfabeto [1]:

$$a, b, c, ... \qquad \text{(variables)}$$
$$\lambda \qquad \text{(lambda)}$$
$$(,) \qquad \text{(paréntesis)}$$

Definimos los términos lambda formalmente como sigue [1]:

Definición 2: Λ λ-términos

La clase Λ de λ-términos es la mínima clase que cumpla lo siguiente:

1. $x \in \Lambda$, *x una variable.*
2. *Si* $M \in \Lambda$ *entonces* $(\lambda x M) \in \Lambda$.
3. *Si* $M, N \in \Lambda$ *entonces* $(MN) \in \Lambda$.

Algunos λ-términos son:

$$x \qquad (xx) \qquad ((xx)(yz)) \qquad (\lambda x(\lambda y(\lambda z((xz)(yz)))))$$

Los términos formados con la segunda regla corresponden a la acción de definir una función, donde la variable después del símbolo λ especifica el nombre del argumento, y los términos formados con la tercera regla corresponden a una aplicación de función. Nuestra primera aproximación a definir la función que suma dos números sería:

$$(\lambda x\ (\lambda y\ (+\ x)\ y))$$

Se debe tener cuidado con el símbolo de suma "$+$" ya que de acuerdo con nuestra sintaxis no tiene un significado específico, por lo que debe tratarse como una variable más, sin embargo, es posible definir este operador usando el cálculo lambda puro [1].
Una gran limitación de la notación parece ser que solo podemos definir funciones unarias, únicamente podemos introducir un argumento formal a la vez. El hecho de que esto no es realmente una restricción fue primero observado por Schönfinkel. Dada alguna función binaría con argumentos formales x y y, digamos $f(x, y)$, entonces definimos:

$$a \quad \equiv \quad (\lambda y(\lambda x(f(x, y))))$$

nótese que a es equivalente a la función original pero toma sus argumentos uno a la vez. Una función como a, que tomas sus argumentos uno a la vez, es comúnmente llamada una

función *currificada* (en honor al matemático Haskell B. Curry [1].

1.3. Convenciones sintácticas

Para facilitar la lectura de los λ-términos evadiremos la proliferación de paréntesis, generalmente utilizando una notación alterna para los términos construidos de acuerdo a la segunda cláusula de la definición:

$$\lambda x.M$$

más aún, asumiremos que este tipo de expresiones asocian a la derecha, de tal forma que los siguientes términos son equivalentes:

$$\lambda x_1 \ldots x_n.M \quad \equiv \quad \lambda \vec{x}.M \quad \equiv \quad (\lambda x_1(\ldots(\lambda x_n M)\ldots))$$

donde $\vec{x}$ es nuestra notación para la secuencia $x_1, \ldots, x_n$. Generalmente se usará el símbolo $\equiv$ para denotar igualdad sintáctica entre los términos [1].

Para los términos correspondientes a la aplicación de funciones se adoptará la convención de que la aplicación asocia a la izquierda [1]. Por lo que:

$$MN_1 \ldots N_n \quad \equiv \quad M\vec{N} \quad \equiv \quad (\ldots(MN_1)\ldots N_n)$$

1.4. Variables libres y ligadas

El símbolo λ actúa ligando variables de la misma forma que $\int \ldots dx$ en el cálculo intergral y los cuantificadores $\exists$ y $\forall$ en el cálculo de predicados [1]. El conjunto de variables ligadas se define inductivamente usando la siguiente función, BV[2] $: \Lambda \rightarrow \wp(\mathbb{X})$ [3]:

$$\begin{aligned} BV\ (x) &= \varnothing \\ BV\ (\lambda x M) &= (BV\ M)\ \cup\ \{x\} \\ BV\ (M\ N) &= (BV\ M)\ \cup\ (BV\ N) \end{aligned}$$

[2] Del inglés *Bound Variables* traducido como *Variables ligadas*

[3] $f : A \rightarrow B$ significa que f es una función que toma argumentos del conjunto A y da resultados del conjunto B. La notación $\wp(A)$ indica el conjunto potencia de A. [32]

Es necesario definir también el conjunto de variables libres en un término, define inductivamente con la siguiente función, $FV^4 : \Lambda \rightarrow \wp(\mathbb{X})$:

$$\begin{aligned} FV\ (x) &= x \\ FV\ (\lambda x M) &= (FV\ M) - \{x\} \\ FV\ (M\ N) &= (FV\ M) \cup (FV\ N) \end{aligned}$$

Ambas funciones definidas en [1]. Cuando $FV\,(M)$ es el conjunto vacío, $\varnothing$, se dice que M es *cerrado*. Nótese que el conjunto de variables libres y el de variables ligadas no necesariamente son disjuntos, x aparece siendo libre y ligada en $x(\lambda xy.x)$ [1].

1.5. Sustitución

Ahora definimos la operación de sustitución sobre expresiones-λ. La *sustitución* es una operación ternaria, denotada mediante $M[x := N]$, donde M, N son expresiones-λ y x es una variable, la notación debe leerse como: "reemplaza todas las ocurrencias libres de x en M por N", sin embargo debe tenerse cuidado ya que una aproximación apresurada podría generar una *captura de variable*. Este problema ocurre cuando ingenuamente se sustituye un término que contiene una variable libre en un alcance en donde la variable se vuelve ligada. Por ejemplo en

$$(\lambda f.f\ x)[x := fy] \quad = \quad \lambda f.f\ (f\ y)$$

Obsérvese el operador de sustitución en el lado izquierdo de la ecuación, sucede que la variable f aparece libre en la tercera parte del operador de sustitución, sin embargo, en el lado derecho de la ecuación ha sido capturada (ligada) por la abstracción. Podemos pensar en f siendo libre antes de la operación como una variable global en programación, al aplicarse la sustitución la variable global ha sido confundida con la variable ligada (argumento) [1].
A continuación mostraremos una aproximación al problema de captura de variable, con base en el trato que originalmente Church le dio a la sustitución [1]. Formalmente la sustitución se define:

[4]Del inglés *Free Variables* traducido como *Variables libres*

Definición 3: Sustitución

1. $x[x := N] = N$
2. $y[x := N] = y$, si x es distinto de y
3. $(\lambda x.M)[x := N] = \lambda x.M$
4. $(\lambda y.M)[x := N] = \lambda y.M[x := N]$, si $x \notin FV(M)$ o bien $y \notin FV(N)$
5. $(\lambda y.M)[x := N] = \lambda z.(M[y := z])[x := N]$, si $x \in FV(M)$ y además $y \in FV(N)$, z una nueva variable
6. $(M_1 M_2)[x := N] = (M_1[x := N])(M_2[x := N])$

Examinemos las reglas 3 y 5 a detalle. La regla 3 se aplica cuando la variable que tratamos de sustituir está ligada por la abstracción inmediata, en cuyo caso la sustitución no tendría efecto pues no habría instancias libres de x en el cuerpo de la abstracción. La regla 4 se aplica cuando no es posible que suceda una captura de variable, o bien x no aparece libre en M (resultando la sustitución sin efecto de nuevo) o bien la variable ligada en la abstracción no aparece libre en el término siendo sustituido (no captura). En cualquier caso podemos continuar y aplicar la sustitución al cuerpo de la abstracción. La regla 5, se aplica cuando una captura de variable podría ocurrir, esto es, la variable ligada en la abstracción aparece libre en el término siendo sustituido, en este caso, primero renombramos la variable ligada por una completamente nueva que no aparece en ninguna parte de las expresiones examinadas [1].

La regla 5 es válida solo bajo el supuesto de que términos similares[5], teniendo las mismas variables libres y que son diferentes únicamente en sus variables ligadas, son en esencia las mismas. En la presentación original del cálculo lambda Church formaliza esta discusión añadiendo un áxioma adicional, (α). La α-reducción es::

$$\lambda x.M = \lambda y.M[x := y], y \notin FV(M)$$

[5]Aquí "similares" parece un poco arbitrario, aclaramos estableciendo que nos referimos a que tienen el mismo árbol sintáctico.

1.6. β-reducción

Aunque es posible currificar[6] para agregar más variables, en su modelo básico el cálculo lambda modela funciones de solamente *una variable*. Para asociar un valor a está variable única es necesario llegar a una expresión de la forma $(\lambda x.M)\ N$, este término indica que se está realizando una *aplicación de función* sobre alguna otra expresión, asociando así la expresión N a x [18].

Este tipo de expresiones son llamadas β-redex o simplemete *redex*[7] y son fundamentales para el desarrollo y estudio de una expresión Lambda. Mediante la β-reducción se busca *reducir* las expresiones en términos más sencillos (aunque este no siempre sea el caso) que normalmente tienen un sentido semántico más concreto (valores) [1].

Definición 4: $\underset{\beta}{\rightarrow}$ β-reducción

Un β-redex se reduce usando $\underset{\beta}{\rightarrow}$ de la forma:

$$(\lambda x.M)\ N \quad \underset{\beta}{\rightarrow} \quad M[x := N]$$

Esto es, el reducir un redex significa sustituir en el cuerpo de la abstracción M la variable x por su parámetro N. Algunos ejemplos de β-reducción son:

$$\begin{aligned} (\lambda x.z)\ y &\underset{\beta}{\rightarrow} z \\ (\lambda x.z)\ (\lambda x.x) &\underset{\beta}{\rightarrow} z \\ (\lambda x.x)\ (\lambda x.z) &\underset{\beta}{\rightarrow} (\lambda x.z) \end{aligned}$$

[6]Una gran limitación de la notación parece ser que solo podemos definir funciones unarias, únicamente podemos introducir un argumento formal a la vez. El hecho de que esto no es realmente una restricción fue primero observado por Schönfinkel. Dada alguna función binaría con argumentos formales x y y, digamos $f(x, y)$, entonces definimos:

$$a \quad \equiv \quad (\lambda y(\lambda x(f(x, y))))$$

nótese que a es equivalente a la función original pero toma sus argumentos uno a la vez. Una función como a, que tomas sus argumentos uno a la vez, es comúnmente llamada una función *currificada* (en honor al matemático Haskell B. Curry [1].

[7]Del inglés ***red****ucible* ***ex****presion* traducido como *expresión reducible*.

En este capítulo se expuso una parte del origen del cálculo lambda, así como su uso para describir *intencionalmente* los algoritmos en las Ciencias de la Computación, se dio su sintaxis y algunas convenciones para su fácil lectura y escritura. También se expuso la forma de reducir expresiones mediante el operador de *sustitución*. Es necesario comprender el cálculo lambda ya que algunas de las máquinas que se implementarán en este trabajo operan sobre el mismo.

Capítulo 2

Máquina SECD

Creada por Peter Landin en su artículo *The Mechanical Evaluation of Expressions* [30] esta máquina fue la primera en ser creada específicamente para el cálculo lambda visto como un lenguaje de programación, la máquina SECD (***S**tack, **E**nvironment, **C**ontrol, **D**ump*) es especialmente importante porque dio pauta al desarrollo subsecuente de muchas máquinas abstractas para lenguajes de programación funcional, es el punto de partida de algunos cursos universitarios y libros de texto además ha sido objeto de muchas variaciones y optimizaciones. Como notables ejemplos de su aportación tenemos *Lispkit Lisp* que es un compilador basado en la máquina [33], el sistema *Lisp/370* [34] y además en 1989 un grupo de investigadores de la universidad de Calgary trabajaron en una implementacón en hardware de la máquina [35].

La máquina SECD usa cuatro componentes:

1. Una pila que guarda los resultados intermedios.
2. Un ambiente que mapea variables a valores.
3. El control que tiene la expresión actual en evaluación.
4. La descarga contiene una lista de tripletas, cada tripleta a su vez contiene una foto o *snapshot* de los componentes de la pila, ambiente y control.

La máquina SECD está definida con un conjunto de transiciones entre los cuatro componentes mencionados anteriormente [8].

2.1. Semántica operacional

En esta sección desarrollaremos estrategias de evaluación deterministas que nos permitirán implementar un evaluador que reduzca mecánicamente una expresión-λ a un *valor*. Este

capítulo está basado (en su mayoría) en el trabajo de Diego Murillo Albarrán [18].

2.1.1. Valores

Al evaluar una expresión-λ quisiéramos obtener un *valor*, esto es una expresión cuyo significado sea atómico y en correspondencia con algún objeto matemático que consideremos significativo, y que por lo tanto no debe de ser evaluada más allá de su forma actual.

Elegir el tipo de expresiones que son consideradas valores depende altamente del propósito del lenguaje, por lo que decidir cuales expresiones son valores y cuales no llega a ser un tanto arbitrario. Muchos lenguajes de programacion no consideran a las funciones como valores lo cual usualmente impide pasar a una función como argumento.

Normalmente, instancias concretas como números, booleanos y estructuras de datos como listas y arreglos son consideradas valores, pues conllevan un significado concreto. Ya que todos estos tipos de datos tendrían que ser codificados como variables y abstracciones-λ, entonces consideraremos estas dos construcciones como valores del cálculo lambda, denotaremos al conjunto de valores mediante el símbolo $\mathcal{V}$.

A continuación se definen las *constantes y operadores primitivos* para después dar el conjunto de valores que usaremos en la máquina SECD.

2.1.2. Constantes y operadores primitivos

Consideraremos un conjunto de constantes primitivas al cual denotaremos mediante $\mathcal{C}_{\mathcal{P}}$. En este conjunto vivirán todas las constantes que consideremos necesarias según los problemas que queramos resolver. Distinguiremos a las constantes de las variables poniendo un punto sobre las constantes; así $\dot{a}$ sera una constante y b una variable.

También necesitaremos de un mecanismo que nos permita efectuar operaciones sobre estas constantes; para ello definiremos al conjunto $\mathcal{O}_{\mathcal{P}}$ que contiene a los operadores primitivos. Diremos que $o^{(n)} \in \mathcal{O}_{\mathcal{P}}$ es un operador primitivo de aridad n, esto es que para ser evaluado espera recibir n argumentos que se evalúen cada uno a una constante primitiva.

Con la inclusión de los conjuntos $\mathcal{C}_{\mathcal{P}}$ y $\mathcal{O}_{\mathcal{P}}$, podemos denotar la aplicación de un operador primitivo sobre una serie de constantes, es necesario también un mecanismo que efectivamente calcule el resultado de esta aplicación; para esto definiremos la función δ.

Definición 5: $\delta : \mathcal{O}_{\mathcal{P}} \times [\mathcal{C}_{\mathcal{P}}] \to \mathcal{C}_{\mathcal{P}}$ Evaluación de operadores

Definiremos la función $\delta(o^{(n)}, [\dot{c}_i])$, donde $o^{(n)}$ es un operador primitivo de aridad n y $[\dot{c}_i]$ es una lista de n constantes primitivas; que devuelve la constante primitiva resultante de evaluar el operador $o^{(n)}$ sobre los argumentos dados.

Armados con está teoría procedemos a definir los valores que usaremos, así como los conjuntos $\mathcal{C}_{\mathcal{P}}$ y $\mathcal{O}_{\mathcal{P}}$ además de la función δ que usará la máquina SECD.

Definición 6: $\mathcal{C}_{\mathcal{P},SECD}$ Constantes primitivas en la máquina SECD

Definiremos el conjunto de constantes primitivas usadas para la máquina SECD como el conjunto de los números enteros $\mathbb{Z}$.

$$\mathcal{C}_{\mathcal{P},SECD} = \mathbb{Z}$$

Algunos elementos de $\mathcal{C}_{\mathcal{P},SECD}$ serían -1, 2, y 0.

Definición 7: $\mathcal{O}_{\mathcal{P},SECD}$ Operadores primitivos en la máquina SECD

Definiremos el conjunto de operadores primitivos usadas para la máquina SECD como la función sucesor de aridad uno en los enteros.

$$\mathcal{O}_{\mathcal{P},SECD} = \{SUC\}$$

Definición 8: δ_{SECD} Evaluación de operadores en la máquina SECD

Definimos la evaluación de operadores como la función sucesor.

$$\delta(SUC, n) = n + 1$$

$$\text{donde } n \in \mathbb{Z}$$

Así pues $\delta(SUC, 1) = 2$, $\delta(SUC, 2) = 3$, ..., etc.

Ahora si estamos listos para definir los valores que usará la semántica operacional de la máquina SECD.

Definición 9: $\mathcal{V}_{SECD}$ Valores de la máquina SECD

$$\begin{aligned}\mathcal{V}_{SECD} = \dot{c} \\ \mid \quad o \\ \mid \quad \lambda x.M\end{aligned}$$

$$\text{donde } \dot{c} \in \mathcal{C}_{\mathcal{P},SECD} \quad o \in \mathcal{O}_{\mathcal{P},SECD} \quad M \in \Lambda$$

Así pues los valores usados en la máquina SECD son: constantes y operadores primitivos o bien alguna abstracción lambda.

2.1.3. Reglas de inferencia

Formalizaremos la semántica operacional mediante la notación de reglas de inferencia. Una regla de inferencia se denota con una línea horizontal donde la parte superior contiene una serie de precondiciones o premisas y en la parte inferior tiene una conclusión. Si mostramos que las precondiciones se cumplen, entonces podemos usar la regla de inferencia para deducir que la conclusión también es válida.

Definición 10: Semánica operacional para la máquina SECD

$$\frac{N \to N'}{(M\ N) \to (M\ N')} \tag{1}$$

$$\frac{v \in \mathcal{V}_{SECD} \quad M \to M'}{(M\ v) \to (M'\ v)} \tag{2}$$

$$\frac{v \in \mathcal{V}_{SECD}}{(\lambda x.M)\ v \to M[x := v]} \tag{3}$$

$$\frac{M \to v \quad v \in \mathbb{Z}}{SUC(M) \to v + 1} \tag{4}$$

La regla (1) se lee como: "si N se evalúa a N', entonces la aplicación $M\ N$ se evalúa a $M\ N'$". Esta regla determina el orden de evaluación en una aplicación indicando que primero se evalúa la expresión de la derecha hasta que se reduzca a un valor.

La regla (2) dice que si v es un valor y M se evalúa a M', entonces la aplicación $M\ v$ se evalúa a $M'\ v$. Haciendo uso de esta regla y la anterior definimos que la evaluación de una aplicación deberá realizarse de derecha a izquierda.

La regla (3) especifica que solamente cuando se aplica una función a un valor es posible la β-reducción.

La regla (4) estipula el uso de los operadores y constantes primitivas, esto es, la función sucesor de los enteros.

2.2. Listas

Definimos una estructura auxiliar de suma importancia para las definiciones futuras: La *lista*. El concepto que una lista representa es el de una estructura de datos que permite definir una secuencia. Tener esta facilidad en los lenguajes de programación es de gran ayuda pues es común encontrarse con la necesidad de expresar colecciones de datos y manipularlos de manera ordenada [7]. Damos algunas convenciones sintácticas para su fácil lectura y escritura, así como funciones básicas para operar con ésta:

Definición 11: $L(A)$ **Listas de A**

La clase $L(A)$ de listas de A es la mínima clase que cumpla lo siguiente:

1. $null \in L(A)$, *A cualquier conjunto*
2. *Si* $x \in A$ *y* $xs \in L(A)$ *entonces* $(x : xs) \in L(A)$

Algunas listas de enteros, $L(\mathbb{Z})$, son:

$$null \qquad (8 : null) \qquad (1 : (2 : (3 : null)))$$

Evadimos la proliferación de paréntesis usando la siguiente notación alterna:

$$[x_1, x_2, ..., x_n] \equiv (x_1 : (x_2 : (... : (x_n : null)...)))$$

así por ejemplo:

$$[\,] \equiv null \qquad [8] \equiv (8 : null) \qquad [1, 2, 3] \equiv (1 : (2 : (3 : null)))$$

Usaremos las siguientes funciones básicas para operar con listas:

head

Toma una lista y regresa su cabeza, la cabeza de una lista es su primer elemento:

$$head([x_1, ..., x_n]) = x_1$$

tail

Toma una lista y regresa su cola, en otras palabras quita la cabeza de una lista dejando el resto:

$$tail([x_1, x_2, ..., x_n]) = [x_2, ..., x_n]$$

length

Da el número de elementos en una lista:

$$length([x_1, ..., x_n]) = n$$

2.3. Ambiente

Introducimos la estructura conocida como *ambiente* que haciendo uso de las listas nos ayudará a realizar sustituciones de manera más eficiente. Consideremos la reducción del siguiente λ-término que suma los números 1 y 5:

$$\begin{aligned} & ((\lambda xy.+\ x\ y)\ 1\ )\ \ 5 \\ \underset{\beta}{\rightarrow} \quad & (\lambda y.+\ 1\ y)\ \ 5 \\ \underset{\beta}{\rightarrow} \quad & +\ \ 1\ \ \ 5 \\ \underset{\beta}{\rightarrow} \quad & 6 \end{aligned}$$

Notemos como los parámetros formales x y y son sustituidos por los términos correspondientes para al final poder realizar la suma[1], haremos que el intérprete busque en algún tipo de estructura el término a sustituir. La estructura guarda las *sustituciones por realizar en*

[1]Recordemos que de acuerdo con nuestra sintaxis tanto el símbolo de suma "+" como los números naturales cerecen de un significado específico, por lo que deben tratarse como una variable más, sin embargo, podemos representar a los números naturales mediante *combinadores* de la forma $\lambda sz.M$ donde M varía para representar a cada natural, también llamados *numerales de Church*, definidos por Alonzo Church en [20, 21]. Otra opción es añadir la suma como operador primitivo.

el futuro sin *realmente* hacerlo, guardando la intención en lugar de sustituir inmediatamente es posible retrasar la sustitución y realizarla solo cuando es necesario, mejorando así la complejidad en tiempo de la evaluación. La estructura de datos resultante es llamada *ambiente* y al proceso de asociar nombres de variables con sus términos se le llama *ligar variables* [22].

Para poder emular este comportamiendo cada que nuestra máquina se encuentre con un término de la forma $(\lambda x.M)N$ en la cadena de control guardará en el ambiente al argumento x junto con el término que lo sustituirá, también llamado *parámetro real* de la función. Posteriormente durante la evaluación del cuerpo de la función, M, si la máquina se encuentra con una variable buscará en el ambiente el término anteriormente guardado y lo sustiruirá para poder continuar con la ejecución. Al proceso de buscar en el ambiente el parámetro real correspondiente a su variable ligada se le conoce como *lookup* [2]. En el ejemplo anterior el ambiente terminará siendo la lista:

$$e = [(x, 1), (y, 5)]$$

Por último asumimos la existencia de la función *lookup*:

$$lookup : \mathbb{X} \times L(\mathbb{X} \times \mathcal{V}_{SECD}) \to \mathcal{V}_{SECD}$$

tal que dada una variable y un ambiente retorna el valor ligado a la variable, en el ejemplo anterior: $lookup(x, e) = 1$ y $lookup(y, e) = 5$

2.4. Cerradura

Sabemos por la semántica operacional que las funciones pueden ser un valor, pero no tenemos una respuesta clara a la pregunta *¿qué valor representa una función?*, primero veamos que podemos hacer con una función como valor. Las funciones son de un tipo diferente de valor que los valores númericos, así que no podemos, por ejemplo, sumarlas. Pero hay una cosa evidente que se puede hacer con ellas: aplicarlas a argumentos [22].

Para ver una función como un valor debemos conocer todo lo necesario para poder aplicarla a sus argumentos, empaquetando todos los componentes de la función en una estructura que indique como procesar los argumentos correctamente. Sus componentes son el nombre del argumento de la función así como el cuerpo de la misma, sin embargo, es interesante observar que siendo el cuerpo de la función una expresión arbitraria es posible tener una

[2]Del inglés *lookup* traducido como *búsqueda*.

definición de función anidada dentro de nuestra definición de función inicial. "*Esto significa que una función vista como un valor necesita* ***recordar las sustituciones*** *que han sido aplicadas previamente a ella. Ya que estamos representando a las sustituciones utilizando un ambiente, el valor funcional necesita incluir un ambiente. La estructura resultante es llamada* ***cerradura*** [22]."

Así pues la cerradura se compone de: argumento de la función, cuerpo de la función y un ambiente para recordar sustituciones pasadas.

Habiendo definido la cerradura procedemos a modificar $\mathcal{V}_{SECD}$ para que deje de incluir λ-términos y utilice cerraduras en su lugar:

Definición 12: $\mathcal{V}_{SECD}$ Valores de la máquina SECD (revisitados)

$$\begin{aligned} \mathcal{V}_{SECD} = \dot{c} \\ \mid \quad o \\ \mid \quad < e, x, M > \end{aligned}$$

$$\text{donde } \dot{c} \in \mathcal{C}_{\mathcal{P},SECD} \quad o \in \mathcal{O}_{\mathcal{P},SECD} \quad M \in \Lambda$$
$$\text{y } e \in L(\mathbb{X} \times \mathcal{V}_{SECD}) \text{ un ambiente}$$

Así pues los valores usados en la máquina SECD son: constantes y operadores primitivos o bien alguna cerradura.

2.5. Estado interno de la máquina SECD

Esta máquina tendrá un estado interno de la forma $< S, E, C, D >$.
Definimos formalmente cada componente de la siguiente manera:

Definición 13: Estado interno de la máquina SECD

El estado interno con el cual opera la máquina SECD es una cuarteta ordenada $< s, e, c, d >$ con $s \in S$, $e \in E$, $c \in C$ y $d \in D$. Siendo los conjuntos S, E, C, D definidos de la siguiente manera:

1. $S = L(\mathcal{V}_{SECD})$
2. $E = L(\mathbb{X} \times \mathcal{V}_{SECD})$
3. $C = L(\Lambda \cup \mathcal{C}_{\mathcal{P},SECD} \cup APPLY)$
4. $D = L(S \times E \times C)$

El primer componente, S, guarda una lista de valores intermedios calculados como resultado de las funciones que se van aplicando en la ejecución del programa, hasta que finalmente, entrega el valor último resultado de nuestro λ-término inicial. El segundo, E, es el registro del *ambiente* actual. El tercero, C, representa la subexpresión que estamos evaluando, también conocida como *cadena de control*. Nótese que en la componente de control además de λ-términos y constantes primitivas (en este caso el conjunto $\mathbb{Z}$) también se permite incluir una directiva especial $APPLY$, esta directiva es solamente para el funcionamiento operativo interno de la máquina y actúa como una bandera que desencadena una serie de acciones necesarias para la evaluación de una aplicación de funciones.
El último componente, D, contiene una lista de tripletas, cada tripleta a su vez contiene una foto o *snapshot* de los componentes de la pila, ambiente y control. Estos *snapshots* se usan para que las aplicaciones de función puedan continuar con la ejecución del programa después de retornar un valor y se toman al momento de aplicar una abstracción funcional a su argumento.

2.6. Reglas de transición

En esta sección desarrollaremos una máquina evaluadora para λ-términos:

Definición 14: Máquina SECD

Definimos el comportamiento de la máquina SECD mediante la siguiente tabla de transiciones donde para cada estado $< s, e, c, d >$ inspeccionamos la forma de sus componentes para especificar el siguiente estado de la máquina hasta alcanzar un estado final, en cuyo caso se entregará el valor único en s como resultado de la evaluación.

	Estado actual	$\underset{SECD}{\mapsto}$	Estado siguiente
1)	$< [v], e', [\,], [\,] >$		*estado final*
2)	$< [v], e', [\,], ((s, e, c) : d) >$		$< (v : s), e, c, d >$
3)	$< s, e, (num : c), d >$		$< (num : s), e, c, d >$
4)	$< s, e, (x : c), d >$		$< (lookup(x, e) : s), e, c, d >$
5)	$< s, e, (\lambda x.M : c), d >$		$< ((e, x, M) : s), e, c, d >$
6)	$< s, e, ((M\ N) : c), d >$		$< s, e, (N : (M : (APPLY : c))), d >$
7)	$< (SUC : (num : s), e, (APPLY : c), d >$		$< ((num + 1) : s), e, c, d >$
8)	$< ((e', x, M) : (v' : s)), e, (APPLY : c), d >$		$< [\,], ((x, v') : e'), [M], ((s, e, c) : d) >$
9)	*estado inicial*		$< [\,], [(suc, SUC)], M, [\,] >$

con $x \in \mathbb{X} \quad num \in \mathbb{Z} \quad M, N \in \Lambda$

1. La primera cláusula especifica que hacer si la cadena de control c y el componente de descarga d están vacíos, que corresponde a terminar el cálculo: el valor al tope de la pila s será el resultado final y se alcanzará el estado final [8].

2. La segunda cláusula corresponde cuando la cadena de control c está vacía y el componente de descarga d no lo está, el cual corresponde al retorno de una función: el cálculo continúa con los componentes guardados en el *snapshot* al inicio del componente de descarga d transfiriendo el valor al tope de la pila actual a la nueva pila, obsérvese que el ambiente anterior e' que corresponde a la función que se estaba evaluando es descartado por completo pues ya no será de utilidad [8].

3. La tercera claúsula indica las acciones a realizar si al inicio de la cadena de control c se encuentra una constante primitiva (en este caso un entero pues $\mathcal{V}_{SECD} = \mathbb{Z}$), en cuyo caso simplemente transferimos la constante a la pila s ya que una constante es en sí un valor.

4. La cuarta claúsula muestra la interacción si al inicio de la cadena de control c se encuentra una variable: el valor correspondiente deberá ser buscado en el ambiente actual e y transferido al tope de la pila s.

5. La quinta claúsula, cuando al inicio de la cadena de control c se encuentra una abstracción lambda la cerradura correspondiente deberá ser transferida al tope de la pila s. Esta cerradura agrupa el ambiente actual y ambos componentes de la abstracción lambda, esto es, su argumento y su cuerpo.

6. La sexta claúsula ilustra el caso si al inicio de la cadena de control c se encuentra una aplicación de función: una directiva $APPLY$, el operador y el operando se tranfieren a la cadena de control c. Nótese que se evalúa primero el argumento de la aplicación N respetando la semántica operacional (Definición 10).

7. La séptima claúsula dice que hacer si al inicio de la cadena de control c se encuentra una directiva $APPLY$, al tope de la pila s un operador primitivo (en este caso la función sucesor pues $\mathcal{O}_{\mathcal{P},SECD} = \{SUC\}$), y el segundo elemento de la pila s es una constante primitiva. Corresponde entonces a la aplicación del operador primitivo a sus respectivas constantes, cuyo número puede variar dependiendo de la aridad del operador. La pila deberá eliminar tanto al operador como sus operandos y el resultado transferido al tope de la pila.

8. La octava claúsula muestra los cambios en la máquina si al inicio de la cadena de control c se encuentra una directiva $APPLY$, al tope de la pila s una cerradura y existe un segundo elemento en la pila. Corresponde a una llamada de función: la pila eliminará sus primeros dos elementos y, junto con el ambiente actual e y el resto de la cadena de control serán transferidos al inicio del componente de descarga (tomando así el *snapshot* que guardará el estado de la máquina). La nueva pila será inicializada con la lista vacía, el nuevo ambiente con el que estaba contenido en la cerradura (extendido con los parámetros formales y reales de la abstracción), y la nueva cadena de control será el cuerpo de la cerradura.

9. La evaluación comienza con la pila s vacía, el ambiente conteniendo todos los operadores primitivos vistos como variables ligados a sus respectivos valores (en este caso solo será la función sucesor), la expresión a evaluar como único elemento de la cadena de control y el componente de descarga vacío.

En este capítulo se definieron los valores, constantes y operadores primitivos. Igualmente las reglas de inferencia que conforman la semántica operacional de la máquina SECD. También se definió la estructura *Lista* junto con algunas operaciones de uso común para manejarla para posteriormente introducir el concepto de *ambiente* como una manera eficiente de implementar sustituciones. Se introdujo la *cerradura* como una

forma de representar abstracciones funcionales como valores usando ambientes. Posteriormente se dió la definición del estado interno de la máquina SECD con una breve explicación de cada uno de sus componentes. Finalmente se expuso la definición de las reglas de transición de la máquina SECD explicando brevemente el propósito de cada una de ellas.

Capítulo 3

Máquina de Krivine

Existen varias formas de reducir una expresión, una de ellas consiste en reducir el *redex* situado más a la izquierda de un término en cada etapa, también llamada *reducción de orden normal*, otra forma de reducción es el orden *aplicativo* que reduce de forma *interior más a la izquierda* (del inglés *leftmost innermost*) [1].

La siguiente máquina ejecuta β-reducción en el cálculo lambda puro utilizando la estrategia de evaluación de orden normal y no incluye constantes, ni operadores primitivos, aunque podría extenderse para hacerlo.

"*Esta máquina utiliza una reducción de cabeza débil para ejecutar el cálculo lambda, lo cual significa que la redex activa debe estar al comienzo del λ-término. Por lo tanto, la evaluación se detiene si no hay redex a la cabeza del λ-término. De hecho, reducimos a la vez una cadena completa $\lambda x_1 ... \lambda x_n$. Por lo tanto, la ejecución también se detiene si no hay suficientes argumentos*" [2].

Es importante notar que la máquina es *perezosa* no en el sentido estricto de los lenguajes de programación, esto es, evaluar las expresiones a lo más una vez, sino más bien evalúa las expresiones hasta cierto punto y solo cuando es necesario.

La parte teórica de este capítulo es tomada principalmente del libro *Lambda Calculi: A guide for computer scientists* del autor Chris Hankin [1].

[1] Un *redex* es ***exterior*** si no está contenido en ningún otro *redex* y es ***interior*** si no contiene ningún otro *redex*. Por ejemplo consideré la expresión $(\lambda x.(\lambda y.xy)z)w$, en este término el redex exterior sería la expresión entera pues no está contenida en algún otro redex mientras que un redex interno sería la subexpresión $(\lambda y.xy)z$ al estar contenida en el término completo.

3.1. Semántica operacional

En esta sección desarrollaremos los elementos para definir las reglas de inferencia que nos permitirán implementar una forma de reducir una expresión-λ a un *valor*.

3.1.1. Notación De Bruijn

Las reducciones en el cálculo lambda a veces resultan problemáticas cuando, al querer evitar una *captura de variable*, surge la necesidad de renombrar variables ligadas. Otro caso en el que es necesario renombrar variables es cuando se quiere establecer la equivalencia de dos expresiones que solo difieren en los nombres de sus variables ligadas (llamada α-*equivalencia*) [23].
En particular en el cálculo lambda manipulado por máquinas abstractas esta actividad de renombrar variables involucra un gran esfuerzo tanto en la codificación misma (implementación) como en el tiempo de ejecución. Sería entonces valioso tratar de eliminar el renombrado o incluso deshacerse de los nombres de las variables por completo [23].

La llamada notación *De Bruijn* utiliza esta idea para representar expresiones-λ. Intuitivamente, podemos pensar en las variables ligadas como referencias al símbolo λ que las liga. Por ejemplo, en el término $(\lambda x.(\lambda y.(yx)))$, la subexpresión (yx) es parte de dos abstracciones funcionales anidadas, siendo $(\lambda y.(yx))$ una de ellas y $(\lambda x.(\lambda y.(yx)))$ la otra, por lo cual sus variables ligadas pueden tener una profundidad de a lo más 2, usando esta profundidad es posible que cada subexpresión pueda identificar a que abstracción funcional pertenecen todas sus variables ligadas. Siguiendo esta idea se puede representar a las variables como enteros que se refieren al índice de sus correspondientes símbolos λ y las abstracciones lambda tienen la forma $\lambda.e$. Es importante notar que la variable ligada en la abstracción carece de nombre [23].

Definición 15: λ-términos en notación De Bruijn

Los λ-términos en notación De Bruijn están definidos de forma inductiva como el mínimo conjunto que cumpla lo siguiente:

1. *Cualquier número natural (mayor que cero) es un término*
2. *Si M y N son términos entonces $(M\ N)$ es un término*
3. *Si M es un término entonces (λM) es un término*

Como ejemplos se pueden ver varios términos escritos con la notación estándar y con la notación De Bruijn:

Estándar	De Bruijn
$\lambda x.x$	$\lambda.1$
$\lambda z.z$	$\lambda.1$
$\lambda x.\lambda y.x$	$\lambda.\lambda.2$
$\lambda x.\lambda y.\lambda s.\lambda z.xs(ysz)$	$\lambda.\lambda.\lambda.\lambda.4\ 2(3\ 2\ 1)$
$(\lambda x.xx)(\lambda x.xx)$	$(\lambda.1\ 1)(\lambda.1\ 1)$
$(\lambda x.\lambda x.x)(\lambda y.y)$	$(\lambda.\lambda.1)(\lambda.1)$

Para representar λ-términos que contengan variables libres necesitaremos una forma de mapear variables libres a enteros. Trabajaremos con un mapeo del conjunto de variables al conjunto de los enteros mayores a cero llamado *contexto* y representado por la letra Γ.

$$\Gamma\ :\ \mathbb{X}\ \rightarrow\ \{x \in \mathbb{Z} \mid x\ >\ 0\}$$

Por ejemplo, si Γ mapea x a 1 y y a 2 entonces la representación en notación De Bruijn del término $(x\ y)$ con respecto a Γ será (1 2) mientras que el término $\lambda z.x\ y\ z$ será $\lambda.2\ 3\ 1$. Nótese que en el segundo ejemplo bajo una λ fue necesario incrementar los índices en uno para evitar capturarles [23].

En general cuando se trabaje con expresiones que contengan variables libres (es decir, cuando se trabaja con respecto a un contexto Γ) necesitaremos modificar los índices de esas variables. Por ejemplo, cuando se sustituya una expresión que contiene una variable libre bajo una λ, debemos desplazar los índices hacia arriba para que continuen refiriéndose a los mismos números con respecto a Γ después de la sustitución [23].
Por ejemplo, si sustituimos la expresión (1 2) por la primera variable ligada en $(\lambda.\lambda.2)$ el resultado sería $(\lambda.\lambda.3\ 4)$ y no $(\lambda.\lambda.1\ 2)$. Utilizaremos una función auxiliar que desplaza o

renombra los índices de las variables libres por encima de un límite i en m unidades:

$$rename_{m,i}(j) \equiv \begin{cases} j, & \text{si } j < i. \\ j + m - 1, & \text{si } j \geq i. \end{cases}$$
$$rename_{m,i}(N_1 N_2) \equiv rename_{m,i}(N_1) rename_{m,i}(N_2)$$
$$rename_{m,i}(\lambda N) \equiv \lambda(rename_{m,i+1}(N))$$

El límite i se inicializa con un valor de uno y lleva un seguimiento de las variables que estaban ligadas en la expresión original y que por lo tanto no deberían ser desplazadas conforme el operador recorre la estructura de la expresión [1]. Usando este nuevo operador podemos definir la β-reducción como aparece en [1]:

$$(\lambda P)Q \quad \underset{\beta}{\rightarrow} \quad P[1 \ := \ Q]$$

Donde:

$$n[m \ := \ N] \equiv \begin{cases} n, & \text{si } n < m. \\ n - 1, & \text{si } n > m. \\ rename_{n,1}(N), & \text{si } n = m. \end{cases}$$
$$(M_1 M_2)[m \ := \ N] \equiv (M_1[m \ := \ N])(M_2[m \ := \ N])$$
$$(\lambda M)[m \ := \ N] \equiv \lambda(M[m + 1 \ := \ N]))$$

La primera regla se encarga de sustituir el término cuándo los índices coinciden, en caso contrario cuando $n < m$ significa que se está tratando de sustituir en una variable ligada quedando la sustitución sin efecto y por último cuando $n > m$ quiere decir que la variable que se intenta afectar es libre por lo que simplente se reduce en uno para que continúen apuntando a la misma variable en Γ a la que apuntaban antes de remover la λ. La segunda regla distribuye la sustitución en la aplicación y la tercera regla incrementa el índice que representa a la variable a sustituir conforme el operador se adentra en una abstracción, esto es porque la misma variable ligada cambia su índice con cada λ que encuentra. Véase por ejemplo la expresión $\lambda x.x(\lambda y.x)(\lambda z.x)$ expresada como $\lambda.1(\lambda.2)(\lambda.3)$ en donde la variable x

adopta los índices 1, 2 y 3 [1].

Para ilustrar el funcionamiento de la reducción consideremos el siguiente ejemplo en el que se reduce la expresión $(\lambda u.\lambda v.\ u\ x)\ y$ con respecto al contexto en el que $\Gamma(x) = 1$ y $\Gamma(y) = 2$.

$$
\begin{aligned}
&(\lambda.\lambda.2\ 3)\ 2 \\
&\underset{\beta}{\rightarrow} (\lambda.2\ 3)[1\ :=\ 2] \\
&\equiv \lambda.(2\ 3)[2\ :=\ 2] \\
&\equiv \lambda.(2[2\ :=\ 2])\ (3[2\ :=\ 2]) \\
&\equiv \lambda.(2[2\ :=\ 2])\ 2 \\
&\equiv \lambda.rename_{2,1}(2)\ 2 \\
&\equiv \lambda.(2 + 2 - 1))\ 2 \\
&\equiv \lambda.3\ 2
\end{aligned}
$$

El resultado $\lambda.3\ 2$ escrito en notación estándar y con respecto a Γ sería $(\lambda v.\ y\ x)$[2]. La notación De Bruijn no es muy legible pero la β-reducción es fácil de implementar; de hecho inspiró la creación de la Máquina CAM (*Categorical Abstract Machine*) [9].

3.1.2. Cerraduras en Notación De Bruijn

Presentamos una representación alternativa del concepto de *cerradura* que funciona utilizando índices De Bruijn y que será esencial para definir las reglas de inferencia de nuestra semántica operacional.

Definición 16: Cerraduras en notación De Bruijn

Las cerraduras en notación De Bruijn, $\mathcal{R}$, están definidos de forma inductiva como el mínimo conjunto que cumpla lo siguiente:

1. *Si M un término en notación De Bruijn y además $u_1, ..., u_n \in \mathcal{R}$ (para n finito y $n \geq 0$) entonces tenemos que $M[u_1, ...u_n] \in \mathcal{R}$*

En la definición, la lista de cerraduras entre corchetes [...] se conoce como *ambiente*. Es importante notar que se se incluye el caso del ambiente vacío cuando $n = 0$. En el futuro

[2]Para una guía del proceso de conversión de notación De Bruijn a notación éstandar consultar en [23].

usaremos $\cdot$ como operador de construcción para ambientes, así $u_1 \cdot u_2, ..., u_n \equiv u_1, ..., u_n$. Algunos ejemplos de cerraduras son $1[1[\lambda.1[\,]]]$ ó $(1\ 2)[\lambda.1[\,], \lambda.1[\,]]$ es importante mencionar que a veces se omiten parentésis para facilitar la lectura de los términos por lo que $1[\lambda.1[\,]]$ podría ser denotado simplemente como $1[\lambda.1]$

3.1.3. Reglas de inferencia

Concentraremos nuestros esfuerzos en cálculos que involucren cerraduras para lo cual usaremos las siguientes reglas:

Definición 17: Semánica operacional para la máquina de Krivine

Dados ρ y ρ' dos ambientes, entonces se tiene que

$$\frac{M[\rho] \xrightarrow{*} \lambda P[\rho']}{(M\ N)\ [\rho] \to P[N[\rho] \cdot \rho']} \quad \text{(Eval)}$$

$$n[u_1, ..., u_m] \to u_n \ \ (n \leq m) \quad \text{(Access)}$$

La primera regla nos dice que si el primer término en un aplicación se reduce a una abstracción lambda, entonces podemos reducir la aplicación a una cerradura que consiste en el cuerpo de la abstracción y un ambiente que es el mismo que el del ambiente de la abstracción con un nuevo elemento al inicio representando la cerradura del argumento.

La segunda regla simplemente accede al término correspondiente en el ambiente dado por el índice n.

Nótese que la primera regla dicta que el ambiente se usa para guardar argumentos, los argumentos nunca están en la posición más a la izquierda y el sistema fuerza una estrategía de reducción que reduce siempre el redex más a la izquierda del término.

Omitiendo mencionar algunos ambientes vacíos para no sobrecargar al lector con notación veamos como ejemplo la siguiente reducción:

$$\begin{aligned}
&(\lambda.1\ 1)(\lambda.1)[] && \text{usando Eval} \\
&\to 1\ 1[\lambda.1] && \text{usando Eval y dado que } 1[\lambda.1] \to \lambda.1[] \\
&\to 1[1[\lambda.1]] && \text{usando Access} \\
&\to 1[\lambda.1] && \text{usando Access} \\
&\to \lambda.1
\end{aligned}$$

3.2. Estado interno de Krivine

Esta máquina tendrá un estado interno de la forma $< \rho, M, S >$.
Definimos formalmente cada componente de la siguiente manera:

Definición 18: Estado interno de la máquina de Krivine

El estado interno con el cual opera la máquina de Krivine es una terna ordenada $< \rho, M, s >$ con $\rho \in P$, $M \in \Lambda_{DB}$ y $s \in S$. Siendo los conjuntos P, Λ_{DB}, S:

1. P, un ambiente.
2. Λ_{DB}, λ-términos en notación De Bruijn.
3. $S = L(\mathcal{R})$, una lista de cerraduras.

La primera componente, ρ, es usada para almacenar el ambiente de la expresión M, mientras que la tercera componente s, es una pila que se utiliza para almacenar las cerraduras de los argumentos mientras que la parte de la abstracción funcional en una aplicación es evaluada.

3.3. Reglas de Transición

La máquina se especifica por las siguientes reglas:

Definición 19: Máquina de Krivine [1]

	Estado actual	$\underset{kr}{\mapsto}$	Estado siguiente
1)	$< \rho, M\ N, S >$		$< \rho, M, N[\rho] : S >$
2)	$< \rho, \lambda M, u : S >$		$< u\cdot \rho, M, S >$
3)	$< u\cdot \rho, n+1, S >$		$< \rho, n, S >$
4)	$< M[v]\cdot \rho, 1, S >$		$< v, M, S >$

con $\rho, v \in P \quad n \in \mathbb{N} - \{0\} \quad M, N \in \Lambda_{DB} \quad u \in \mathcal{R}$

La primera y segunda regla corresponden a la implementación de la regla (Eval) de la semántica operacional, la primera indica que si la expresión a evaluar es una aplicación entonces se guarda en la pila la cerradura del argumento para su posterior evaluación y se procede a evaluar la función correspondiente a la aplicación colocandola en la cadena de control, la segunda regla se encarga de procesar una abstracción lambda colocando su cuerpo en la cadena de control y construyendo el ambiente con el tope de la pila. Es importante

notar que el argumento se recupera de la pila donde fue puesto por la primera regla.
Las últimas dos reglas buscan recursivamente en el ambiente implementando así la regla (Access) de la semántica operacional.

Nótese que los estados finales de la máquina son de la forma $< \rho, \lambda M, [\,] >$ o bien de la forma $< [\,], n, S >$. Estados de la primera forma corresponden a λ-términos de la forma:

$$\lambda x.M$$

Y estados de la segunda forma corresponden a λ-términos de la forma:

$$xM_1...M_n$$

donde los términos a la derecha están en la pila S de la máquina, para una prueba de la correspondencia de los estados finales de la máquina con los λ-términos descritos ver [1] y [2].

Un ejemplo de ejecución de la máquina sería la siguiente secuencia de transiciones:

$$\begin{aligned}
&< [\,], (\lambda.1\ 1)(\lambda.1), [\,] > && \text{usando 1)}\\
&\rightarrow < [\,], \lambda.1\ 1, \lambda.1 > && \text{usando 2)}\\
&\rightarrow < \lambda.1, 1\ 1, [\,] > && \text{usando 1)}\\
&\rightarrow < \lambda.1, 1, 1[\lambda.1] > && \text{usando 4)}\\
&\rightarrow < [\,], \lambda.1, 1[\lambda.1] > && \text{usando 2)}\\
&\rightarrow < 1[\lambda.1], 1, [\,] > && \text{usando 4)}\\
&\rightarrow < \lambda.1, 1, [\,] > && \text{usando 4)}\\
&\rightarrow < [\,], \lambda.1, [\,] >
\end{aligned}$$

En este capítulo se expuso una nueva notación que surgió como una solución al problema de captura y renombre de variables, la notación De Bruijn, además se dieron varios ejemplos de su utilización con expresiones que incluyeran variables libres y ligadas. Se mostró una forma de escribir las *cerraduras* usando la notación De Bruijn que posteriormente se utilizó para construir la semántica operacional de la máquina de Krivine. Finalmente se definió el estado interno de la máquina de Krivine así como sus reglas de transición incluyendo una breve explicación de las reglas y componentes de la misma.

Capítulo 4

Máquina CEK

Desarrollada por Matthias Felleisen y Dan Friedman esta máquina actúa sobre el cálculo lambda y provee una estrategia para escribir intérpretes, que además de ser eficiente tiene varias propiedades útiles incluyendo su facilidad para añadir rasgos más complejos a un lenguaje cómo lo son las continuaciones, es fácil detener al intérprete para obtener la pila de ejecución actual y es sencillo introducir hilos [26].

La máquina CEK usa tres componentes:

1. El control que es la expresión actual en evaluación.
2. Un ambiente que mapea variables a valores.
3. El componente de continuaciones que es el contexto dinámico de evaluación de la expresión.

Está máquina fue usada como base para implementar el intérprete del lenguaje de programación funcional Scheme [38].

El cálculo lambda es una base natural para un lenguaje de programación, Landin reconoció este hecho y lo usó para estudiar las diferentes características de los lenguajes de programación [25], sin embargo este era incapaz de modelar aspectos más imperativos de los lenguajes de programación, mejor conocidos como *operadores de control*, tales como *catch, throw, return, goto* entre otros. Estos operadores traen consigo diferentes ventajas y vale la pena estudiar una forma de modelar estos aspectos [26]. En este capítulo presentamos una extensión del cálculo lambda simple que utiliza *continuaciones*, una abstracción funcional que representa al resto del programa, para incorporar operadores de control así como una máquina evaluadora para dicho cálculo.

4.1. Continuaciones

El cálculo lambda modela funciones describiendo sus reglas en lugar de solamente sus gráficas. Pensar y modelar programas usando el cálculo lambda trae consigo varias ventajas ya que el carácter de regla que tienen las evaluaciones de función se acerca a la comprensión operacional de un programador acerca de los programas computacionales y, al mismo tiempo, el cálculo lambda proporciona un marco algebraico para razonar acerca de las funciones. Sin embargo está adecuación impidió el desarrollo posterior del cálculo lambda pues está basada en la simplicidad más que en modelar los programas de forma más conveniente [24].

La única forma que tiene el cálculo lambda de calcular un valor o resultado es la β-reducción que modela la aplicación de funciones. Sin embargo esta no es sufiente en muchas situaciones, por ejemplo, si a mitad de una evaluación recursiva el programa encuentra su resultado o sucede algún error el programa debería ser capaz de reportarlo *de inmediato*, sin cálculos o reducciones posteriores. Este tipo de comportamiento es muy común y fácilmente implementado dentro del paradigma imperativo utilizando *operadores de control*. El problema es que las funciones necesitan más *control* sobre su evaluación si van a ser usadas como modelo para programas computacionales [24].

"*La solución más general al problema de control dentro de la esfera funcional se originó en la semántica denotacional. Un programa puede ser evaluado evaluando sus partes y combinando sus resultados. Cuando un componente en particular está siendo evaluado uno puede pensar en las sub-evaluaciones restantes y en el proceso de combinación, es decir, el resto de la evaluación, como una* continuación *de la sub-evaluación actual. La idea crucial es escribir programas en un estilo de forma que las funciones puedan ser usadas para* simular *continuaciones.*" [24]

Las continuaciones reifican[1] el estado de control de un programa en un punto de su evaluación para que así pueda ser utilizado en lugar de permanecer oculto en el ambiente de ejecución [27].

Cuando las continuaciones son tratadas como funciones de primera clase son capaces de dirigir el proceso de evaluación: puede decidirse no usarla, guardarla para ser usada después o seguir la evaluación en algún otro punto de la ejecución.

[1]La reificación es el proceso de implementar un componente abstracto de un programa computacional, volviendolo así explicito y manejable.

Una aplicación conocida de este concepto es un estilo de programación llamado *continuation-passing style* o CPS[2] en el que cada función toma un argumento extra, una función representando una continuación para explícitamente pasar el control del programa [28].

Consideremos el siguiente ejemplo en Haskell, definimos las funciones necesarias para calcular $c = \pi r^2$, el área de un círculo de radio r:

```
pow2 :: Float -> Float
pow2 a = a ** 2

mult :: Float -> Float -> Float
mult a b = a * b

circle :: Float -> Float
circle r = mult 3.1416 (pow2 r)
```

La función `pow2` toma un argumento y lo eleva al cuadrado mientras que la funcíón `mult` toma dos elementos y devuelve su producto. Por último la función `circle` toma el elemento r correspondiente al radio del círculo y regresa el resultado que correspondería al área del mismo.

Ahora veamos su implementación en *continuation-passing style*, debemos cambiar las firmas: Cada función recibe también un argumento de tipo función que representa la continuación y el tipo de retorno dependerá de lo que retorne esa función.

```
pow2' :: Float -> (Float -> a) -> a
pow2' a cont = cont (a ** 2)

mult' :: Float -> Float -> (Float -> a) -> a
mult' a b cont = cont (a * b)

circle' :: Float -> (Float -> a) -> a
circle' r cont = pow2' r
                    (\r2 -> mult' r2 3.1416
                      cont)
```

El procedimiento completo en *continuation-passing style* se encuentra en la función `circle'`, primero se calcula r^2 y se pasa el control a la función que toma r^2 y calcula su producto con la constante π, para terminar pasamos la función identidad que retorna su argumento sin cambios como continuación indicando así el término del cálculo, no queda nada por hacer.

```
*Main> circle 0.5 id
0.7854
```

[2]Traducido como *estilo de paso de continuación*.

También podríamos pasar como continuación final la función `print` que se llama con el resultado y se imprime en la consola en lugar de ser devuelto directamente

```
*Main> circle 0.5 print
0.7854
```

Es notable que estos programas no son muy legibles y además son díficiles de construir, sería mejor contar con facilidades lingüistias que permitan acceder a la continuación actual cuando sea necesario. Programas con está funcionalidad son "mucho más simples, fáciles de entender (con un poco de práctica) y más fáciles de escribir. Son además más confiables pues la máquina que lleva el cálculo construye la continuacion mecánicamente en lugar de ser el programador el que la escribe a mano..."[3]. Ejemplos de funcionalidades de este estilo en lenguajes basados en el cálculo lambda son el operador ***J*** [30] y *call-with-current-continuation*[4] [31]. En la siguiente sección presentamos una extensión del cálculo lambda que incorpora una funcionalidad capaz de manipular continuaciones más fácilmente.

4.2. Λ_c el cálculo lambda extendido

El conjunto de términos tradicional del cálculo lambda puro es la base para el nuevo lenguaje pero adicionalmente incluye dos símbolos que representan dos tipos nuevos de aplicaciones: $\mathcal{C}$-aplicaciones y $\mathcal{A}$-aplicaciones. A continuación se define formalmente la sintaxis, nótese que a pesar de no definir constantes es posible agregarlas fácilmente. En lo siguiente se omitirán paréntesis excesivos para una mejor lectura y escritura de los términos [26].

4.2.1. Sintaxis y semántica

La clase de λ_c-términos consiste de palabras construidas del siguiente alfabeto [26]:

$x, k, f, v, ...$	(variables)
λ	(lambda)
$\mathcal{C}, \mathcal{A}$	(manejo de continuaciones)
$(,)$	(paréntesis)

Definimos los λ_c-términos formalmente como sigue [26]:

[3] C. Talcott hablando sobre CPS en Rum, un dialecto de Lisp [29], p.68
[4] Traducido como *llamar con la continuación actual*

Definición 20: Λ_c λ_c-términos [26]

La clase Λ_c de λ_c-términos es la mínima clase que cumpla lo siguiente:

1. *$x \in \Lambda_c$, x una variable*
2. *Si $M \in \Lambda_c$ y x una variable, entonces $(\lambda x.M) \in \Lambda_c$*
3. *Si $M, N \in \Lambda_c$ entonces $(MN) \in \Lambda_c$*
4. *Si $M \in \Lambda_c$ entonces $(\mathcal{C}M) \in \Lambda_c$, M es llamado el $\mathcal{C}$-argumento*
5. *Si $M \in \Lambda_c$ entonces $(\mathcal{A}M) \in \Lambda_c$, M es llamado el $\mathcal{A}$-argumento*

Algunos λ_c-términos son:

$$x \qquad (xx) \qquad ((\mathcal{C}x)(yz)) \qquad (\mathcal{C}(\lambda.x(\lambda.y(\lambda.z((xz)(\mathcal{A}z))))))$$

La noción de variables libres y ligadas en un término se transmite directamente del Cálculo Lambda *puro*, en cuanto a $\mathcal{C}$ y $\mathcal{A}$ diremos que no son símbolos ni libres ni ligados. La definicion del operador de sustitución, $M[x := N]$, sigue la misma lógica que en Cálculo Lambda puro pero la $\mathcal{C}$-aplicación y la $\mathcal{A}$-aplicación son tratadas como aplicaciones donde la parte de la abstracción es ignorada [26].

El significado de las construcciones derivadas del Cálculo Lambda puro se mantiene de manera aproximada, por ejemplo una variable representa un valor y una abstracción corresponde a una función. Sin embargo la aplicación además de invocar funciones en un argumento también puede invocar *continuaciones* en un argumento. Hablando de las nuevos tipos de aplicaciones la $\mathcal{A}$-aplicación aborta el cálculo actual y comienza uno nuevo con su argumento como punto de partida y la $\mathcal{C}$-aplicación captura la continuación actual del programa y la pasa como una abstracción funcional a su argumento [26].

Este conjunto de símbolos nos permite acceder y manipular las continuaciones más fácilmente, acercando así nuestro lenguaje a una definición funcional de los diversos operadores de control existentes en el paradigma imperativo.

4.2.2. Reducción con continuaciones, C-reducción

Dado que el cálculo lambda puro es nuestra base necesitamos la β-reducción:

$$(\lambda x.M)N \quad \underset{\beta}{\rightarrow} \quad M[x := N]$$

Sin embargo, esta no define como hay que reducir las $\mathcal{A}$-aplicaciones ni las $\mathcal{C}$-aplicaciones,

ahora definamos como se reducen las $\mathcal{A}$-aplicaciones. Intuitivamente lo que el operador $\mathcal{A}$ debe hacer es olvidarse de la continuación actual y comenzar una nueva con su argumento como punto de partida. Recordemos que una continuación es representada como el *resto* del cálculo que habría que hacer, por lo que nuestra primera aproximación de reducción sería como sigue:

$$(\mathcal{A}M)N \underset{\mathcal{A}_L}{\rightarrow} \mathcal{A}M \qquad (\mathcal{A}_L)$$

Observemos que la $\mathcal{A}$-aplicación hace que se quede solamente el término M (el argumento de $\mathcal{A}$) y lo demás se olvida. El caso en el cual $\mathcal{A}$ está a la derecha es análogo:

$$M(\mathcal{A}N) \underset{\mathcal{A}_R}{\rightarrow} \mathcal{A}N \qquad (\mathcal{A}_R)$$

Observemos ahora que la $\mathcal{A}$-aplicación hace que se quede solamente el término N (nuevamente, el argumento de $\mathcal{A}$) y lo demás se olvida.

Solo nos falta definir qué pasaría cuando $\mathcal{A}$ está en la raíz del término, esto requiere un tratamiento especial. La intuición nos dice que $\mathcal{A}M$ con una continuación vacía, en otras palabras cuando ya no queda nada por hacer, debe evaluar a M:

$$\mathcal{A}M \underset{\mathcal{A}_T}{\rightarrow} M \qquad (\mathcal{A}_T)$$

Sin embargo esto puede llevar a inconsitencias, para ver que problemas pueden surgir supongamos que I es la función identidad, entonces:

$$(\mathcal{A}I)N \underset{\mathcal{A}_L}{\rightarrow} (\mathcal{A}I) \underset{\mathcal{A}_T}{\rightarrow} I$$

Pero también es posible:

$$(\mathcal{A}I)N \underset{\mathcal{A}_T}{\rightarrow} (IN) \underset{\beta}{\rightarrow} N$$

Esto nos lleva a que I sería igual a N para cualquier N lo cual es inconsistente con el cálculo lambda razón por la cual introducimos esta operación en la raíz como una *regla de cálculo*, estableciendo además que solo es posible usarla cuando la $\mathcal{A}$-aplicación esté en la raíz del término y usamos $\rhd$ en lugar de $\rightarrow$:

$$\mathcal{A}M \rhd_{\mathcal{A}} M \qquad (\mathcal{A}_T)$$

Usando estas reglas es posible eliminar las inconsistencias, así el ejemplo anterior se reduciría de la siguiente forma:

$$
\begin{array}{ll}
(\mathcal{A}I)N & \text{usando } (\mathcal{A}_L) \\
\underset{\mathcal{A}_L}{\rightarrow}\ \mathcal{A}I & \text{usando } (\mathcal{A}_T) \\
\rhd_{\mathcal{A}}\ I &
\end{array}
$$

Resultando la función identidad, que es justamente el argumento inicial de la $\mathcal{A}$-reducción.

Es importante resaltar la forma en la cual el operador $\mathcal{A}$ se propaga, es decir que poco a poco va eliminando partes de un término hasta hacer desaparecer por completo la continuación actual, para ello observe la siguiente reducción:

$$(xy)(z((\mathcal{A}(\lambda x.x))w)) \underset{\mathcal{A}_L}{\rightarrow} (xy)(z(\mathcal{A}(\lambda x.x))) \underset{\mathcal{A}_R}{\rightarrow} (xy)(\mathcal{A}(\lambda x.x)) \underset{\mathcal{A}_R}{\rightarrow} \mathcal{A}(\lambda x.x) \rhd_{\mathcal{A}} \lambda x.x$$

En cada reducción el operador $\mathcal{A}$ prevalece junto con su argumento y va quitando las subexpresiones externas (estas subexpresiones corresponden a la continuación actual con respecto al subtérmino que contiene la $\mathcal{A}$-aplicación). Debemos tener en cuenta esta forma de operar porque la manera en la que la $\mathcal{C}$-aplicación *captura* la continuación actual es muy parecida.

Para diseñar correctamente las $\mathcal{C}$-aplicaciones debemos recordar la semántica deseada, es decir, la $\mathcal{C}$-aplicación debe capturar la continuación actual y suministrarla a su argumento. Primero hay que definir la continuación actual de la expresión $(\mathcal{C}M)N$, recordemos que una continuación representa el *resto* del cálculo o bien lo que aún hay que hacer luego de reducir M. Y lo que hay que hacer en este caso es pasar N cómo argumento. En otras palabras si f es la función a la que le pasaremos N entonces la continuación en forma de abstracción sería:

$$\lambda f.fN$$

Debemos luego pasar esta función al argumento de la $\mathcal{C}$-aplicación M, quedando así:

$$M(\lambda f.fN)$$

Es en este punto en el que debemos recordar la forma en la que las $\mathcal{A}$-aplicaciones logran interactuar con toda la continuación propagando el operador desde dentro de la subexpresión hacía afuera, paso a paso. Debido a que el operador debe prevalecer hasta el exterior del término (momento en el cual se usa la *regla de cálculo*) debemos modificar nuestra expresión agregando una $\mathcal{C}$-aplicación junto con una función lambda que toma una k como parámetro,

esta k actuará como un acumulador recolectando paso a paso la continuación actual, de forma que tenemos:

$$\mathcal{C}\lambda k.M(\lambda f.k(fN))$$

Sin embargo, aún hay un aspecto semántico de $\mathcal{C}$ que se está olvidando. Recordemos que las continuaciones y el operador $\mathcal{C}$ surgieron como una forma de manejar mejor el ambiente de ejecución como una respuesta en el ambiente funcional a los operadores de control. Esto significa darle al programador la libertad de utilizar la continuación actual una o múltiples veces y de detener el programa *de inmediato, sin cálculos o reducciones posteriores*. Esto se traduce a que cuando una continuación es invocada se debe descartar el contexto actual, así que para nuestras Λ_c-continuaciones la primera acción debe ser un *abortar* para olvidar el contexto actual. Por lo tanto la expresión a la que $(\mathcal{C}M)N$ debería reducirse es:

$$\mathcal{C}\lambda k.M(\lambda f.\mathcal{A}(k(fN)))$$

Definimos finalmente las reglas de reducción usando dos casos de manera análoga a los del operador $\mathcal{A}$ de la siguiente manera:

$$(\mathcal{C}M)N \underset{\mathcal{C}_L}{\to} \mathcal{C}\lambda k.M(\lambda f.\mathcal{A}(k(fN))) \qquad (\mathcal{C}_L)$$

$$M(\mathcal{C}N) \underset{\mathcal{C}_R}{\to} \mathcal{C}\lambda k.N(\lambda x.\mathcal{A}(k(Mx))) \qquad (\mathcal{C}_R)$$

Aún hay que examinar el caso en el que la continuación es vacía, esto es, no hay más operaciones por realizar pues la $\mathcal{C}$-aplicación está en la raíz del término. El $\mathcal{C}$-argumento M debe ser aplicado a una continuación que simula la ausencia de operaciones a calcular. La elección natural es la función identidad con una pequeña modificación $\lambda x.\mathcal{A}x$. De nuevo esta es una *regla de cálculo* para evitar inconsistencias:

$$\mathcal{C}M \rhd_{\mathcal{C}} M(\lambda x.\mathcal{A}x) \qquad (\mathcal{C}_T)$$

En conjunto todas las reglas necesarias para reducir un λ_c-término serían:

$$(\lambda x.M)N \quad \underset{\beta}{\to} \quad M[x := N] \qquad (\beta\text{-reducción})$$

$$(\mathcal{A}M)N \underset{\mathcal{A}_L}{\to} \mathcal{A}M \qquad (\mathcal{A}_L)$$

$$M(\mathcal{A}N) \underset{\mathcal{A}_R}{\to} \mathcal{A}N \qquad (\mathcal{A}_R)$$

$$\mathcal{A}M \rhd_{\mathcal{A}} M \qquad (\mathcal{A}_T)$$

$$(\mathcal{C}M)N \underset{\mathcal{C}_L}{\to} \mathcal{C}\lambda k.M(\lambda f.\mathcal{A}(k(fN))) \qquad (\mathcal{C}_L)$$

$$M(\mathcal{C}N) \underset{\mathcal{C}_R}{\to} \mathcal{C}\lambda k.N(\lambda x.\mathcal{A}(k(Mx))) \qquad (\mathcal{C}_R)$$

$$\mathcal{C}M \rhd_{\mathcal{C}} M(\lambda x.\mathcal{A}x) \qquad (\mathcal{C}_T)$$

Hemos definido todas las reducciones y reglas de cálculo que se necesitan para una función de reducción estándar[5] para Λ_c [26].

Definición 21: $\underset{c}{\to}$ Función de reducción para λ_c-términos [26]

$$\underset{c}{\to} \quad = \quad \underset{\beta}{\to} \cup \underset{\mathcal{C}_L}{\to} \cup \underset{\mathcal{C}_R}{\to} \cup \underset{\mathcal{A}_L}{\to} \cup \underset{\mathcal{A}_R}{\to} \cup \rhd_{\mathcal{C}} \cup \rhd_{\mathcal{A}}$$

4.2.3. Ejemplos de reducción con continuaciones

Un ejemplo muy útil para observar el comportamiento de las continuaciones es la reducción de la expresión $F((\mathcal{C}I)M)$ donde I es la función identidad $\lambda x.x$, este programa debe obtener la continuación de la subexpresión $\mathcal{C}I$ y pasarsela como argumento a I quedando solo la continuación, observemos como sucede esto:

[5]La función que se define podría contener abusos de notación, una correcta definición de cómo estás reglas se unen para formar una función de reducción necesita un conocimiento teórico más profundo que escapa al alcance de este trabajo.

$$F((\mathcal{C}I)M) \tag{1}$$

$$\underset{\mathcal{C}_L}{\rightarrow} F(\mathcal{C}\lambda k.I(\lambda f.\mathcal{A}(k(f\ M)))) \tag{2}$$

$$\underset{\mathcal{C}_R}{\rightarrow} \mathcal{C}\lambda k_1.(\lambda k.I(\lambda f.\mathcal{A}(k(f\ M))))(\lambda x.\mathcal{A}(k_1(F\ x))) \tag{3}$$

$$\rhd_{\mathcal{C}} (\lambda k_1.(\lambda k.I(\lambda f.\mathcal{A}(k(f\ M))))(\lambda x.\mathcal{A}(k_1(F\ x))))(\lambda x.\mathcal{A}x) \tag{4}$$

$$\underset{\beta}{\rightarrow} \lambda k.I(\lambda f.\mathcal{A}(k(f\ M)))(\lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)(F\ x))) \tag{5}$$

$$\underset{\beta}{\rightarrow} I(\lambda f.\mathcal{A}((\lambda x.\mathcal{A}((\lambda x.\mathcal{A}x)(F\ x)))(f\ M))) \tag{6}$$

$$\underset{\beta}{\rightarrow} I(\lambda f.\mathcal{A}((\lambda x.\mathcal{A}(\mathcal{A}(F\ x)))(f\ M))) \tag{7}$$

$$\underset{\beta}{\rightarrow} I(\lambda f.\mathcal{A}(\mathcal{A}(\mathcal{A}(F\ (f\ M))))) \tag{8}$$

$$\underset{\mathcal{A}_R}{\rightarrow} I(\lambda f.\mathcal{A}(\mathcal{A}(F\ (f\ M)))) \tag{9}$$

$$\underset{\mathcal{A}_R}{\rightarrow} I(\lambda f.\mathcal{A}(F\ (f\ M))) \tag{10}$$

$$\underset{\beta}{\rightarrow} \lambda f.\mathcal{A}(F\ (f\ M)) \tag{11}$$

Vale la pena notar algunos aspectos del ejemplo anterior, en primer lugar el resultado fue el término $\lambda f.\mathcal{A}(F\ (f\ M))$, que es justamente la continuación de la subexpresión $\mathcal{C}I$ en el término $F((\mathcal{C}I)M)$ ¿por qué?, porque al terminar de evaluar $\mathcal{C}I$ aún queda como *el resto del cálculo* pasarle M como argumento y luego aplicarle F que es de hecho lo que la función $\lambda f.\mathcal{A}(F\ (f\ M))$ haría con su parámetro f. Es posible afirmar entonces que en efecto el operador $\mathcal{C}$ logró capturar la continuación y pasarsela como argumento a la función identidad I, por lo cuál la semántica deseada tuvo efecto.

También cabe resaltar como desde el paso (2) hasta el paso (4) el operador $\mathcal{C}$ se propagó hacía afuera del término hasta finalmente desaparecer. Desde el paso (5) al (8) se utilizó k_1 y k como acumuladores para ir recolectando paso a paso la continuación actual hasta llegar a la expresión $I(\lambda f.\mathcal{A}(\mathcal{A}(\mathcal{A}(F\ (f\ M)))))$, que ya tenía una forma bastante parecida al resultado deseado, finalmente solo quedó reducir las $\mathcal{A}$-aplicaciones hasta solo quedar una y aplicar la función identidad en los pasos (9) a (11). Después de toda la reducción nos quedamos con la expresión $\lambda f.\mathcal{A}(F\ (f\ M))$ que, además de hacer el resto del cálculo, si se llega a utilizar corta abruptamente el flujo del programa haciendo uso del operador $\mathcal{A}$.

El siguiente ejemplo reduce un término haciendo uso de algunas constantes para una

mayor compresión del proceso, en este caso la continuación que se captura suma dos a su argumento y el argumento que recibe es cero:

$$
\begin{array}{ll}
+\ 2\ (\mathcal{C}\lambda k.(k\ 0)) & \text{usando } (\mathcal{C}_R) \\
\underset{c}{\rightarrow} \mathcal{C}\lambda k_0.(\lambda k.(k\ 0))(\lambda x.\mathcal{A}(k_0(+\ 2\ x))) & \text{usando } (\beta) \\
\underset{c}{\rightarrow} \mathcal{C}\lambda k_0.(\lambda x.\mathcal{A}(k_0(+\ 2\ x)))0 & \text{usando } (\beta) \\
\underset{c}{\rightarrow} \mathcal{C}\lambda k_0.\mathcal{A}(k_0(+\ 2\ 0)) & \text{usando } (\beta) \\
\underset{c}{\rightarrow} \mathcal{C}\lambda k_0.\mathcal{A}(k_0\ 2) & \text{usando } (\rhd_{\mathcal{C}}) \\
\underset{c}{\rightarrow} (\lambda k_0.\mathcal{A}(k_0\ 2))\ (\lambda x.\mathcal{A}\ x) & \text{usando } (\beta) \\
\underset{c}{\rightarrow} \mathcal{A}((\lambda x.\mathcal{A}\ x)\ 2) & \text{usando } (\beta) \\
\underset{c}{\rightarrow} \mathcal{A}(\mathcal{A}\ 2) & \text{usando } (\rhd_{\mathcal{A}}) \\
\underset{c}{\rightarrow} \mathcal{A}\ 2 & \text{usando } (\rhd_{\mathcal{A}}) \\
\underset{c}{\rightarrow} 2 &
\end{array}
$$

Hemos realizado una suma utilizando continuaciones, pero ¿qué pasa si la continuación que se pasa y utiliza es vacía?, debería olvidar todo el contexto al momento de utilizarse y regresar solo lo que su argumento evalúa. Este tipo de continuaciones son conocidas como *continuaciones de escape* [26]. Un ejemplo sería la siguiente reducción:

$$
\begin{array}{ll}
\mathcal{C}(\lambda k.+\ 2\ (k\ 0)) & \text{usando } (\rhd_{\mathcal{C}}) \\
(\lambda k.+\ 2\ (k\ 0))(\lambda x.\mathcal{A}\ x) & \text{usando } (\beta) \\
+\ 2\ ((\lambda x.\mathcal{A}\ x)\ 0) & \text{usando } (\beta) \\
+\ 2\ (\mathcal{A}\ 0) & \text{usando } (\mathcal{A}_R) \\
\mathcal{A}\ 0 & \text{usando } (\rhd_{\mathcal{A}}) \\
\underset{c}{\rightarrow} 0 &
\end{array}
$$

Al usarse la continuación la suma queda olvidada por completo. A continuación definimos la semántica operacional de la máquina CEK.

4.3. Semántica operacional

En esta sección desarrollaremos los elementos para definir las reglas de inferencia que nos permitirán implementar un evaluador que reduzca una expresión-λ_c a un *valor*.

Comenzamos definiendo el conjunto de valores a utilizar:

Definición 22: $\mathcal{V}_{CEK}$ Valores de la máquina CEK [26]

$$\begin{aligned} \mathcal{V}_{CEK} &= x \\ &\mid \quad \lambda x.M \end{aligned}$$

$$\text{donde } x \in \mathbb{X} \quad \text{y} \quad M \in \Lambda$$

Así pues los valores usados en la máquina CEK son: variables o bien alguna abstracción lambda que no incluya continuaciones.

Ahora presentamos las reglas de inferencia:

Definición 23: Semánica operacional para la máquina CEK [26]

$$\frac{M \underset{c}{\to} M'}{(M\ N) \underset{c}{\to} (M'\ N)} \tag{1}$$

$$\frac{v \in \mathcal{V}_{CEK} \quad N \underset{c}{\to} N'}{(v\ N) \underset{c}{\to} (v\ N')} \tag{2}$$

$$\frac{v \in \mathcal{V}_{CEK}}{(\lambda x.M)\ v \underset{c}{\to} M[x := v]} \tag{3}$$

$$\frac{}{(\mathcal{A}\ M)\ N \underset{c}{\to} \mathcal{A}\ M} \tag{4}$$

$$\frac{M \in \mathcal{V}_{CEK}}{M\ (\mathcal{A}\ N) \underset{c}{\to} \mathcal{A}\ N} \tag{5}$$

$$\frac{}{(\mathcal{C}\ M)\ N \underset{c}{\to} \mathcal{C}\lambda k.M(\lambda f.\mathcal{A}(k(f\ N)))} \tag{6}$$

$$\frac{M \in \mathcal{V}_{CEK}}{M(\mathcal{C}\ N) \underset{c}{\to} \mathcal{C}\lambda k.N(\lambda x.\mathcal{A}(k(M\ x)))} \tag{7}$$

Las primeras tres reglas representan un comportamiento muy similar al de la máquina SECD, ambas maquinas implementan paso por valor, sin embargo las reducciones se realizan de izquierda a derecha en este caso. Las reglas (4) y (5) capturan el comportamiento de $\mathcal{A}_L$ y $\mathcal{A}_R$ respectivamente, siendo la única restricción que en $\mathcal{A}_R$ el término M a la izquierda sea un valor. Las reglas (6) y (7) son análogas a $\mathcal{C}_L$ y $\mathcal{C}_R$ comportandose de la misma manera que en las dos reglas anteriores, es decir, la única restricción es que en $\mathcal{C}_R$ el término M a la izquierda sea un valor [26].

4.4. Estado interno de la máquina CEK

Antes de presentar formalmente el estado interno recordamos el concepto de *cerradura* y *ambiente*. Además definimos algunos conceptos clave para el desarrollo de la máquina.

4.4.1. Códigos de continuación y puntos de continuación

Un *ambiente* es un mapeo finito del conjunto de variables al conjunto de *valores semánticos*, representado con una lista de pares ordenados. Los *valores semánticos* se conforman de la unión de *cerraduras* y *puntos de continuación* donde las cerraduras son la representación de abstracciones funcionales vistas como valores y conformadas por una tripleta ordenada $< e, x, M >$ cuyos elementos corresponden al ambiente, argumento y cuerpo de la función respectivamente. Los *puntos de continuación* son estructuras que etiquetan continuaciones de la forma $< \mathbf{P}, k >$ donde $\mathbf{P}$ es la etiqueta y k es un *código de continuación*. Los códigos de continuación representan el *resto* del cálculo, esto es, lo que la máquina tendrá que hacer posteriormente a evaluar el término actual [26].
Definimos los códigos de continuación de la siguiente manera:

Definición 24: CC, Códigos de continuación [26]

Los códigos de continuación CC, se dividen en dos tipos, primero p-continuaciones de la forma:

$$\begin{aligned} p\text{-cont} = \ & \textbf{(stop)} \\ | \ & (k \ \textbf{cont}) \\ | \ & (k \ \textbf{arg} \ N\rho) \\ | \ & (k \ \textbf{fun} \ V) \end{aligned}$$

donde k es una p-continuación, V es un valor semántico, ρ un ambiente y $N \in \Lambda_c$. El segundo tipo son ret-continuaciones de la forma:

$$ret\text{-cont} = (k \ \textbf{ret} \ V)$$

donde k es una p-continuación y V es un valor semántico.

4.4.2. Estado interno

La máquina CEK tendrá un estado interno de la forma <C,E,K>. Definimos formalmente cada componente de la siguiente manera:

Definición 25: Estado interno de la máquina CEK

El estado interno con el cual opera la máquina CEK es una terna ordenada $< c, e, k >$ con $c \in C$, $e \in E$ y $k \in K$. Siendo los conjuntos C, E, K definidos de la siguiente manera:

1. $C = \Lambda_c \cup \{\updownarrow\}$
2. $E = L(\mathbb{X} \times (CPoint \ \cup \ Closure))$
3. $K = CC$

Donde

$$CPoint = \{< \mathbf{P}, k > : \ k \in CC\}$$

y

$$Closure = \{< e, x, M > : \ e \in E \ , \ x \in \mathbb{X} \ , \ M \in \Lambda_c\}$$

La primera componente C será la encargada de almacenar el término que se está evaluando así como el símbolo $\updownarrow$ que se utiliza para indicar a la máquina cuándo tiene que procesar el componente de continuación. La segunda componente guarda el ambiente representado como una lista de pares ordenados donde la primera coordenada es una variable y la segunda componente es un *valor semántico*, esto es, la unión de puntos de continuación y cerraduras. La tercera componente es un código de continuación, CC [26].

4.5. Reglas de Transición

Los estados de la máquina CEK son o bien una tripleta de la forma
$< \updownarrow, [\,], k >$ donde k es una *ret*-continuación o una tripleta de la forma
$< M, \rho, k >$ donde M es un λ_c-término, ρ es un ambiente y k es una *p*-continuación.
Los estados de la forma $< M, [\,], \textbf{(stop)} >$ son los *estados iniciales* además, para todos los valores semánticos V, $< \updownarrow, [\,], ((\textbf{stop}) \ \textbf{ret} \ V) >$ es un *estado final* [26].
La máquina CEK opera utilizando las siguientes reglas:

Definición 26: Máquina CEK [26]

	Estado actual	$\underset{CEK}{\mapsto}$	Estado siguiente
1)	$< x, \rho, k >$		$< \updownarrow , [\,] , (k \textbf{ ret } lookup(x, \rho)) >$
2)	$< \lambda x.M, \rho, k >$		$< \updownarrow , [\,] , (k \textbf{ ret } < \rho, x, M >) >$
3)	$< MN, \rho, k >$		$< M, \rho, (k \textbf{ arg } N \ \rho) >$
4)	$< \updownarrow , [\,] , ((k \textbf{ arg } N \ \rho) \textbf{ ret } F) >$		$< N, \rho, (k \textbf{ fun } F) >$
5)	$< \updownarrow , [\,] , ((k \textbf{ fun } < \rho, x, M >) \textbf{ ret } V) >$		$< M, \ ((x, V) : \rho) , k >$
6)	$< \mathcal{C}M, \rho, k >$		$< M, \rho, (k \textbf{ cont}) >$
7)	$< \updownarrow , [\,] , ((k \textbf{ cont}) \textbf{ ret } < \rho, x, M >) >$		$< M, \ ((x, < \mathbf{P}, k >) : \rho) , \textbf{(stop)} >$
8)	$< \updownarrow , [\,] , ((k \textbf{ cont}) \textbf{ ret } < \mathbf{P}, k_0 >) >$		$< \updownarrow , [\,] , (k_0 \textbf{ ret } < \mathbf{P}, k >) >$
9)	$< \updownarrow , [\,] , ((k \textbf{ fun } < \mathbf{P}, k_0 >) \textbf{ ret } V) >$		$< \updownarrow , [\,] , (k_0 \textbf{ ret } V) >$
10)	$< \mathcal{A}M, \rho, k >$		$< M, \rho, (\textbf{stop}) >$

Donde:

$$x \in \mathbb{X} \quad \rho \in E \quad k, k_0 \in CC \quad M, N \in \Lambda_c \quad F, V \in (CPoint \ \cup \ Closure)$$

Además se utiliza la siguiente función:

$$lookup : \mathbb{X} \times E \to (CPoint \ \cup \ Closure)$$

tal que dada una variable y un ambiente retorna el valor ligado a la variable.

Las reglas 1) a 5) definen un evaluador de paso por valor clásico del cálculo lambda puro, esto es la máquina SECD.

Los pasos 6) a 8) definen las operaciones necesarias para procesar una $\mathcal{C}$-aplicación.

En 6) la continuación se marca usando el código de continuación $(k \textbf{ cont})$ y se coloca el $\mathcal{C}$-argumento M, en el componente de control para su evaluación. Posteriormente en 7) el término M a evaluar en 6) se ha convertido ya en una cerradura $< \rho, x, M >$ y se procede a evaluar el cuerpo de la cerradura ingresando antes la continuación al ambiente en forma de punto de continuación y asociándola a la variable x. Esencialmente la regla 7) encapsula la continuación actual en un punto de continuación y se lo pasa como argumento al $\mathcal{C}$-argumento. El último paso de esta secuencia 8) reemplaza la continuación actual por la continuación inicial.

La regla 9) muestra que la invocación de una continuación remueve la continuación actual y utiliza la continuación anterior en su lugar. Esta regla implementa el comportamiento *inmediato* de las continuaciones que al ser usadas olvidan todo el contexto y se enfocan solo en su argumento.

Por último la regla 10) referente a las $\mathcal{A}$-aplicaciones ignora la continuación y comienza la evaluación usando el $\mathcal{A}$-argumento implementando así el comportamiento de terminar el cálculo restante [26].

La función de evaluación regresa valores semánticos. Una cerradura libre de continuaciones puede mapearse a un λ_c-término sustituyendo todas sus variables libres por los términos que corresponden en su ambiente asociado mientras que los puntos de continuación no tienen un significado obvio [26].

En este capítulo se mencionó la incapacidad del cálculo lambda simple para modelar *operadores de control* y se presentó el concepto de *continuación*, una manera de acceder al estado de control de un programa en un punto de su ejecución, como una posible forma de implementarlos. Sin embargo utilizar continuaciones en el cálculo lambda simple resulta en programas que no son muy fáciles de leer y además díficiles de construir, debido a esto definimos Λ_c el cálculo lambda extendido incluyendo primitivas nativas del lenguaje que permitan acceder a la continuación actual cuando sea necesario ($\mathcal{A}$-aplicaciones y $\mathcal{C}$-aplicaciones) simplificando el uso y manejo de las continuaciones. Se definió la sintaxis, semántica y función de reducción de Λ_c junto con algunos ejemplos de reducciones usando continuaciones. Finalmente se definió el estado interno de la máquina CEK así como sus reglas de transición incluyendo una breve explicación de las reglas y componentes de la misma.

Parte II

Implementación de las máquinas de reducción en Haskell

Capítulo 5

Implementación en Haskell

5.1. Máquina SECD

En este capítulo se implementan las definiciones y funciones correspondientes a la máquina SECD, dado que Haskell es un lenguaje de programación funcional las definiciones se traducen una a una directamente al código.

```
1 module SECD where  -- Definicion del modulo CEK a implementar
2 import Data.Maybe  -- Importacion del tipo Maybe
```

Código 5.1: Definición del módulo e importación del tipo **Maybe**.

5.1.1. El Cálculo Lambda

Codificamos los λ-términos usando el tipo `Term`, además serán extendidos para incluir el conjunto de constantes primitivas $\mathcal{C}_{\mathcal{P},SECD}$ que en este caso será el conjunto de números enteros $\mathbb{Z}$ representado por el tipo `Int`. El tipo `Ide` renombra al tipo `String` para codificar los identificadores de variables.

```
1 type Ide = String
2 data Term = Lit {num :: Int} -- Enteros
3         | Var {name :: Ide} -- Variables
4         | Lam {param::Ide, body::Term}  -- Abstracciones
5         | App {fun::Term, arg::Term} -- Aplicacion
6   deriving (Eq)
```

Código 5.2: Definición del tipo **Term** que representa a los λ-términos.

Su representación como cadena:

```
1 instance Show Term where
2   show (Lit n) = show n
```

```
3   show (Var x) = x
4   show (Lam x b) = "L " ++ x ++ "." ++ (show b)
5   show (App f p) = "(" ++ (show f) ++ " " ++ (show p) ++ ")"
```

Código 5.3: Representación en cadena del tipo **Term**.

Un ejemplo de λ-término construido usando el tipo `Term` sería el siguiente $(\lambda x.x\ x)(\lambda y.y\ y)$ que se traduce como:

```
1 ex1 = (App
2         (Lam "x"
3           (App
4             (Var "x")
5             (Var "x")))
6         (Lam "y"
7           (App
8             (Var "y")
9             (Var "y"))))
```

Código 5.4: Término $(\lambda x.x\ x)(\lambda y.y\ y)$ construido usando el tipo **Term**.

Otro ejemplo de construcción válido sería el término $(\lambda x.\ succ\ x)\ 8$ implementado como:

```
1 ex2 = (App
2         (Lam "x"
3             (App
4                 (Var "succ")
5                 (Var "x")))
6         (Lit 8))
```

Código 5.5: Término $(\lambda x.\ succ\ x)\ 8$ construido usando el tipo **Term**.

5.1.2. Ambientes

Los ambientes se construyen recursivamente utilizando un *tipo paramétrico*. Un tipo paramétrico es un *tipo* que puede recibir otros tipos como *paramétros* a través de *variables de tipo*. Observemos el siguiente código escrito en Haskell:

```
1 ghci> :t head
2 head :: [a] -> a
```

Código 5.6: Ejemplo de tipo paramétrico.

En el ejemplo se muestra el tipo de la funcion `head` que nos da el primer elemento de una lista, este tipo toma una lista de elementos de tipo a y nos regresa un resultado de tipo a. Esta a es la *variable de tipo* que representa a cualquier tipo según la lista que se construya

y se pase a la función `head`. Así pues si `head` recibe una lista de tipo `Char` regresará un elemento de tipo `Char`, si recibe una lista de tipo `Int` regresará un elemento de tipo `Int`, etc. Los tipos paramétricos son muy poderosos porque permiten escribir funciones más generales y nos ahorran la tarea de escribir un tipo para cada caso específico [15].

Un ambiente puede ser vacío o puede ser un identificador y su valor junto con otro ambiente, esta definición es muy parecida a la dada para listas en el capítulo dedicado a la teoría de la máquina SECD.

```
1 data Environment a = Empty
2         | Env Ide a (Environment a)
3          deriving (Eq, Show)
```

Código 5.7: Definición del tipo paramétrico **Environment**.

Adicionalmente se define la función *lookup* (buscar) la cual busca recursivamente valores en el ambiente usando un identificador. Esta definición hace uso del tipo `Maybe` para identificar cuando existe una variable libre en el término y arrojar un error.

```
1 elookup :: Ide -> Environment a -> Maybe  a
2 elookup x (Empty) = Nothing
3 elookup x (Env y a env)
4   | x == y = Just a
5   | x /= y = elookup x env
```

Código 5.8: Definición de la función de búsqueda en ambientes **elookup**.

Por ejemplo, el ambiente $[(x, 14), (y, 10)]$ utiliza como argumento del tipo paramétrico el tipo `Integer`:

```
1 ejemplo_ambiente = Env "x" 14 (Env "y" 10 Empty)
```

Código 5.9: Ambiente construido utilizando el tipo **Environment**.

La función *lookup* aplicada a y debería regresar un valor de tipo entero, 10:

```
1 *CEK> elookup "y" ejemplo_ambiente
2 Just 10
```

Código 5.10: Ejemplo de uso de la función de búsqueda en ambientes.

Mientras que la función *lookup* aplicada a z, una variable libre, deberia darnos un valor indefinido, representado por `Nothing` el valor de tipo `Maybe`:

```
1 *CEK> elookup "z" ejemplo_ambiente
2 Nothing
```

Código 5.11: Ejemplo de uso de la función de búsqueda en ambientes.

5.1.3. Valores

Definimos el tipo `Value` como el conjunto de valores de la máquina SECD, esto es constantes y operadores primitivos o bien alguna cerradura. En este caso en particular tomaremos como conjunto de constantes primitivas a los números enteros ($\mathbb{Z}$) y como operadores primitivos únicamente a la función *sucesor*. Para implementar a las cerraduras incluimos sus elementos: argumento de la función, cuerpo de la función y un ambiente. El ambiente es importante porque siendo el cuerpo de la función una expresión arbitraria es posible tener una definición de función anidada dentro de nuestra definición de función inicial por lo cual las funciones necesitan recordar sustituciones pasadas.

La implementación de $\mathcal{V}_{SECD}$ sería la siguiente:

```
1 data Value = Nat Int
2     | Succ
3     | Closure (Environment Value) Ide Term
4   deriving (Show)
```

Código 5.12: Definición del tipo **Value** que representa a los valores de la máquina SECD.

Algunos ejemplos de valores serían el número entero 8, la función sucesor y la cerradura que representa a la función identidad $< [\,], x, x >$. En Haskell se implementría así:

```
1 ex1 = Nat 8
2 ex2 = Succ
3 ex3 = Closure Empty "x" (Var "x")
```

Código 5.13: Ejemplos de valores construidos utilizando el tipo **Value**.

5.1.4. Cadena de control

El tipo `Directive` representa la subexpresión que estamos evaluando, también conocida como *cadena de control*. Además de λ-términos y constantes primitivas este tipo también permite una directiva especial $APPLY$ que actúa como una bandera que desencadena una serie de acciones para el cálculo de una aplicación de funciones.
Las directivas correspondientes a la cadena de control c:

```
1 data Directive = T Term | Apply  deriving (Show, Eq)
```

Código 5.14: Definición del tipo **Directive** la cadena de control de la máquina SECD.

5.1.5. Componentes de SECD

Dividiremos los componentes de la máquina SECD en sus cuatro partes correspondientes usando los tipos `S`, `E`, `C` y `D`. Recordemos brevemente las definiciones de los componentes:

1. $S = L(\mathcal{V}_{SECD})$
2. $E = L(\mathbb{X} \times \mathcal{V}_{SECD})$
3. $C = L(\Lambda \cup \mathcal{C}_{\mathcal{P},SECD} \cup APPLY)$
4. $D = L(S \times E \times C)$

Las partes que componen el estado interno son: `S` como una lista de valores, `E` como un ambiente, `C` como una lista de directivas y `D` como una lista de tripletas ordenadas que contiene una foto o *snapshot* de los componentes de la pila, ambiente y control.
En Haskell:

```
type S = [Value]
type E = Environment Value
type C = [Directive]
type D = [(S,E,C)]
```

Código 5.15: Definición de los tipos correspondientes a los componentes de la máquina SECD.

Tomemos como ejemplo el estado inicial de la máquina para evaluar la función identidad que toma la forma $< [\,], [(suc, SUC)], \lambda x.x, [\,] >$ y que representada en Haskell sería:

```
ex1 = ([],[(Env "suc" Succ Empty)],[(Lam "x" (Var "x"))], [])
```

Código 5.16: Ejemplo de implementación de un estado de la máquina SECD.

De la misma forma el estado final $< [8], [(suc, SUC)], [\,], [\,] >$ sería:

```
ex2 = ([Nat 8],[(Env "suc" Succ Empty)],[], [])
```

Código 5.17: Ejemplo de implementación de un estado de la máquina SECD.

5.1.6. Reglas de transición

Definimos las reglas que dictan el funcionamiento de la máquina SECD con la función `run` la cual toma un estado y va avanzando paso a paso hasta llegar a obtener un valor.

```
run :: S -> E -> C -> D -> Value
```

Código 5.18: Declaración de tipo de la función *run*.

Procedemos a implementar cada una de las reglas que integran la máquina SECD (Definición 14), las reglas de transición son muy semejantes a sus reglas teóricas correspondientes esto es por como se construyó cada uno de sus componentes.

La regla 1) especifica qué hacer si la cadena de control c y el componente de descarga d están vacíos, que corresponde a terminar el cálculo: el valor al tope de la pila s será el

resultado final y se alcanzará el estado final, en la implementación se regresa el valor que corresponde al resultado. La definición del estado final es:

$$< [v], e', [\,], [\,] >$$

la implementación correspondiente a 1) sería:

```
1 run (v:[]) e [] [] = v                                    -- (* Regla 1 *)
```

Código 5.19: Implementación de la regla de transición 1).

La regla 2) corresponde cuando la cadena de control c está vacía y el componente de descarga d no lo está, lo cual significa el retorno de una función: el cálculo continúa con los componentes guardados en el *snapshot* al inicio del componente de descarga d transfiriendo el valor al tope de la pila actual a la nueva pila, obsérvese que el ambiente anterior e' que corresponde a la función que se estaba evaluando es descartado por completo pues ya no será de utilidad. La definición es:

$$< [v], e', [\,], ((s, e, c) : d) > \underset{SECD}{\mapsto} < (v : s), e, c, d >$$

la implementación correspondiente a 2) sería:

```
1 run (v:[]) e' [] ((s,e,c):d) = run (v:s) e c d      -- (* Regla 2 *)
```

Código 5.20: Implementación de la regla de transición 2).

La regla 3) indica las acciones a realizar si al inicio de la cadena de control c se encuentra una constante primitiva (en este caso un entero pues $\mathcal{V}_{SECD} = \mathbb{Z}$), en cuyo caso simplemente transferimos la constante a la pila s ya que una constante es en sí un valor. La definición es:

$$< s, e, (num : c), d > \underset{SECD}{\mapsto} < (num : s), e, c, d >$$

la implementación correspondiente a 3) sería:

```
1 run s e ((T (Lit n)):c) d = run ((Nat n):s) e c d      -- (* Regla 3 *)
```

Código 5.21: Implementación de la regla de transición 3).

La regla 4) muestra la interacción si al inicio de la cadena de control c se encuentra una variable: el valor correspondiente deberá ser buscado en el ambiente actual e y transferido al tope de la pila s. La definición es:

$$< s, e, (x : c), d > \underset{SECD}{\mapsto} < (lookup(x, e) : s), e, c, d >$$

la implementación correspondiente a 4) sería:

```
run s e ((T (Var x)):c) d = if isJust (elookup x e)     -- (* Regla 4 *)
      then run ((fromJust (elookup x e)):s) e c d
      else error ("Failed to find variable name: " ++ x)
```

Código 5.22: Implementación de la regla de transición 4).

Esta regla busca valores en el ambiente e imprime un mensaje si es que no encuentra el identificador solicitado, usamos el tipo `Maybe` para el manejo de errores.

La regla 5), cuando al inicio de la cadena de control c se encuentra una abstracción lambda la cerradura correspondiente deberá ser transferida al tope de la pila s. La definición es:

$$< s, e, (\lambda x.M : c), d > \underset{SECD}{\mapsto} < ((e, x, M) : s), e, c, d >$$

la implementación correspondiente a 5) sería:

```
run s e ((T (Lam x t)):c) d =
         run ((Closure e x t):s) e c d            -- (* Regla 5 *)
```

Código 5.23: Implementación de la regla de transición 5).

La regla 6) ilustra el caso si al inicio de la cadena de control c se encuentra una aplicación de función: una directiva $APPLY$, el operador y el operando se tranfieren a la cadena de control c. La definición es:

$$< s, e, ((M\ N) : c), d > \underset{SECD}{\mapsto} < s, e, (N : (M : (APPLY : c))), d >$$

la implementación correspondiente a 6) sería:

```
run s e ((T (App t0 t1)):c) d =                    -- (* Regla 6 *)
        run s e ((T t1):(T t0):Apply:c) d
```

Código 5.24: Implementación de la regla de transición 6).

La regla 7) dice qué hacer si al inicio de la cadena de control c se encuentra una directiva $APPLY$, al tope de la pila s un operador primitivo (en este caso la función sucesor pues $\mathcal{O}_{\mathcal{P},SECD} = \{SUC\}$), y el segundo elemento de la pila s es una constante primitiva. Corresponde entonces a la aplicación del operador primitivo a sus respectivas constantes. La pila deberá eliminar tanto al operador como sus operandos y el resultado transferido al tope de la pila. La definición es:

$$< (SUC : (num : s), e, (APPLY : c), d > \underset{SECD}{\mapsto} < ((num + 1) : s), e, c, d >$$

la implementación correspondiente a 7) sería:

```
run (Succ:(Nat n):s) e (Apply:c) d =                    -- (* Regla 7 *)
        run ((Nat (succ n)):s) e c d
```

Código 5.25: Implementación de la regla de transición 7).

La regla 8) muestra los cambios en la máquina si al inicio de la cadena de control c se encuentra una directiva $APPLY$, al tope de la pila s una cerradura y existe un segundo elemento en la pila. Corresponde a una llamada de función: la pila eliminará sus primeros dos elementos y, junto con el ambiente actual e y el resto de la cadena de control serán transferidos al inicio del componente de descarga (tomando así el *snapshot* que guardará el estado de la máquina). La nueva pila será inicializada con la lista vacía, el nuevo ambiente con el que estaba contenido en la cerradura (extendido con los parámetros formales y reales de la abstracción), y la nueva cadena de control será el cuerpo de la cerradura. La definición es:

$$< ((e', x, M) : (v' : s)), e, (APPLY : c), d > \underset{SECD}{\mapsto} < [\,], ((x, v') : e'), [M], ((s, e, c) : d) >$$

la implementación correspondiente a 8) sería:

```
run ((Closure e' x t):v':s) e (Apply:c) d =          -- (* Regla 8 *)
        run [] (Env x v' e') [T t] ((s,e,c):d)
```

Código 5.26: Implementación de la regla de transición 8).

La regla 9) representa el estado inicial de la evaluación, comienza con la pila s vacía, el ambiente conteniendo todos los operadores primitivos vistos como variables ligados a sus respectivos valores (en este caso solo será la función sucesor), la expresión a evaluar como único elemento de la cadena de control y el componente de descarga vacío. La definición del estado inicial es:

$$< [\,], [(suc, SUC)], M, [\,] >$$

la implementación correspondiente a 9) seria:

```
eval :: Term -> Value                                -- (* Regla 9 *)
eval t = run [] (Env "succ" Succ Empty) [T t] []
```

Código 5.27: Implementación de la regla de transición 9).

La regla 9) crea una nueva función `eval` que utiliza `run` para realizar la reducción del término, `eval` es la función encargada de definir el estado inicial de nuestra ejecución almacenando en el ambiente los operadores primitivos y en el componente de control la expresión a evaluar.

En conjunto todas las reglas de transición de la máquina SECD serían:

```
run :: S -> E -> C -> D -> Value
run (v:[]) e [] [] = v                                          -- (* Regla 1 *)
run (v:[]) e' [] ((s,e,c):d) = run (v:s) e c d                  -- (* Regla 2 *)
run s e ((T (Lit n)):c) d = run ((Nat n):s) e c d               -- (* Regla 3 *)
run s e ((T (Var x)):c) d = if isJust (elookup x e)             -- (* Regla 4 *)
      then run ((fromJust (elookup x e)):s) e c d
      else error ("Failed to find variable name: " ++ x)
run s e ((T (Lam x t)):c) d =                                   -- (* Regla 5 *)
        run ((Closure e x t):s) e c d
run s e ((T (App t0 t1)):c) d =                                 -- (* Regla 6 *)
        run s e ((T t1):(T t0):Apply:c) d
run (Succ:(Nat n):s) e (Apply:c) d =                            -- (* Regla 7 *)
        run ((Nat (succ n)):s) e c d
run ((Closure e' x t):v':s) e (Apply:c) d =                     -- (* Regla 8 *)
        run [] (Env x v' e') [T t] ((s,e,c):d)

eval :: Term -> Value                                           -- (* Regla 9 *)
eval t = run [] (Env "succ" Succ Empty) [T t] []
```

Código 5.28: Implementación de todas las reglas de transición de la máquina SECD.

5.1.7. Reducción paso a paso

Considere la siguiente reducción paso a paso utilizando las reglas de transición presentes en la Definición 14:

$$
\begin{aligned}
& < [\,], (suc, SUC), (\lambda x.x)\ (\lambda z.z), [\,] > && \text{usando 6)} \\
\underset{SECD}{\mapsto} & < [\,], (suc, SUC), [(\lambda z.z), (\lambda x.x), APPLY], [\,] > && \text{usando 5)} \\
\underset{SECD}{\mapsto} & < ((suc, SUC), z, z), (suc, SUC), [(\lambda x.x), APPLY], [\,] > && \text{usando 5)} \\
\underset{SECD}{\mapsto} & < [((suc, SUC), x, x), ((suc, SUC), z, z)], (suc, SUC), APPLY, [\,] > && \text{usando 8)} \\
\underset{SECD}{\mapsto} & < [\,], [(x, ((suc, SUC), z, z)), (suc, SUC)], x, ([\,], (suc, SUC), [\,]) > && \text{usando 4)} \\
\underset{SECD}{\mapsto} & < ((suc, SUC), z, z), [(x, ((suc, SUC), z, z)), (suc, SUC)], [\,], ([\,], (suc, SUC), [\,]) > && \text{usando 2)} \\
\underset{SECD}{\mapsto} & < ((suc, SUC), z, z), (suc, SUC), [\,], [\,] > && \text{usando 1)} \\
\underset{SECD}{\mapsto} & \ estado\ final
\end{aligned}
$$

Ahora veamos como se ve esta reducción paso a paso en nuestra implementación con Haskell, para ello utilizaremos un par de funciones auxiliares que nos facilitarán ver el pro-

greso de la ejecución, o dicho de otra manera, como es que los estados de la máquina van cambiando conforme avanza la reducción y se van aplicando cada una de las reglas: la función `run_step :: (S,E,C,D) -> (S,E,C,D)` [1] que ejecuta un sólo paso de la transición y también la función `run_n_steps` que ejecuta n cantidad de pasos dados como argumentos en la expresión:

```
1 run_n_steps :: (S,E,C,D) -> Integer -> (S,E,C,D)
2 run_n_steps state 0 = state
3 run_n_steps state n = run_n_steps (run_step state) (n-1)
```

Código 5.29: Implementación de la función *run_n_steps* que nos ayudará a inspeccionar la reducción paso a paso.

Procedemos ahora implementar la expresión $(\lambda x.x)\ (\lambda z.z)$ en Haskell:

```
1 id1 = (Lam "x" (Var "x"))
2 id2 = (Lam "z" (Var "z"))
3 example = (App id1 id2)
4 -- Poner la expresion en un estado inicial
5 example_run = ([],(Env "succ" Succ Empty),[T example], [])
```

Código 5.30: Implementación de la expresión $(\lambda x.x)\ (\lambda z.z)$ usando el tipo *Term*

Ahora para la ejecución paso a paso, el principio refleja el estado inicial de la máquina ya que aún no se ha realizado ninguna transición. Para este término es el estado $< [\,], [(suc, SUC)], (\lambda x.x)\ (\lambda z.z), [\,] >$. En Haskell:

```
1 *SECD> run_n_steps example_run 0
2 ([],
3 Env "succ" Succ Empty,
4 [T (L x . x  L z . z )],
5 [])
```

Código 5.31: Estado de la máquina al no haber realizado transiciones.

Las lineas 2, 3, 4 y 5 del código representan a los componentes S, E, C y D de la máquina respectivamente. Según la reducción mencionada al principio de la sección el primer paso de la reducción sería aplicar la regla 6) lo cual significa reducir una aplicación de función de la siguiente manera:

$$< [\,], (suc, SUC), (\lambda x.x)\ (\lambda z.z), [\,] > \underset{SECD}{\mapsto} < [\,], (suc, SUC), [(\lambda z.z), (\lambda x.x), APPLY], [\,] >$$

En la implementación con Haskell el término resultante, esto es, el término del lado derecho sería:

[1] Omitimos la definición de esta función debido a su gran tamaño pero el lector podrá comprobar si intenta definirla que es muy parecida a la función *run*

```
*SECD> run_n_steps example_run 1
([],
Env "succ" Succ Empty,
[T L z . z ,T L x . x ,Apply],
[])
```

Código 5.32: Estado de la máquina al haber realizado un paso de la transición.

De nuevo, tomando en cuenta que las últimas cuatro líneas del código representan los componentes de la máquina SECD podemos confirmar que los términos se corresponden. El siguiente paso de la reducción sería tomar el resultado anterior y aplicar la regla 5) pues tenemos una abstracción funcional en la cadena de control:

$$\begin{aligned} &< [\,], (suc, SUC), [(\lambda z.z), (\lambda x.x), APPLY], [\,] > \\ \underset{SECD}{\mapsto} &< ((suc, SUC), z, z), (suc, SUC), [(\lambda x.x), APPLY], [\,] > \end{aligned}$$

En este punto es importante notar que la abstracción $(\lambda z.z)$ pasó de la cadena de control al componente S en forma de cerradura. El término resultante en Haskell:

```
*SECD> run_n_steps example_run 2
([Closure (Env "succ" Succ Empty) "z"  z ],
Env "succ" Succ Empty,
[T L x . x ,Apply],
[])
```

Código 5.33: Estado de la máquina al haber realizado dos pasos de la transición

Continuando con la reducción hay que aplicar la regla 5) de nuevo pues otra vez tenemos una abstracción funcional en la cadena de control:

$$\begin{aligned} &< ((suc, SUC), z, z), (suc, SUC), [(\lambda x.x), APPLY], [\,] > \\ \underset{SECD}{\mapsto} &< [((suc, SUC), x, x), ((suc, SUC), z, z)], (suc, SUC), APPLY, [\,] > \end{aligned}$$

Ahora es turno de la abstracción $(\lambda x.x)$ de pasar como cerradura al componente S. En Haskell:

```
*SECD> run_n_steps example_run 3
([Closure (Env "succ" Succ Empty) "x"  x,Closure (Env "succ" Succ Empty) "
    z" z],
Env "succ" Succ Empty,
[Apply],
```

```
5 [])
```

Código 5.34: Estado de la máquina al haber realizado tres pasos de la transición

El siguiente paso es aplicar la regla 8) pues en la pila se encuentra una cerradura seguida de un valor además de que en la cadena de control la directiva siguiente es un APPLY:

$$\begin{aligned} &< [((suc, SUC), x, x), ((suc, SUC), z, z)], (suc, SUC), APPLY, [\,] > \\ \underset{SECD}{\mapsto} &< [\,], [(x, ((suc, SUC), z, z)), (suc, SUC)], x, ([\,], (suc, SUC), [\,]) > \end{aligned}$$

Corresponde a una llamada de función: la pila eliminará sus primeros dos elementos y, junto con el ambiente actual y el resto de la cadena de control serán transferidos al inicio del componente de descarga (tomando así el *snapshot* que guardará el estado de la máquina). La nueva pila será inicializada con la lista vacía, el nuevo ambiente con el que estaba contenido en la cerradura extendido con el par $(x, ((suc, SUC), z, z))$, y la nueva cadena de control será el cuerpo de la cerradura. En Haskell el resultado de estos cambios sería:

```
1 *SECD> run_n_steps example_run 4
2 ([],
3 Env "x" (Closure (Env "succ" Succ Empty) "z"  z ) (Env "succ" Succ Empty),
4 [T  x ],
5 [([],Env "succ" Succ Empty,[])])
```

Código 5.35: Estado de la máquina al haber realizado cuatro pasos de la transición

Ahora es posible utilizar la regla 4), el motivo es que al inicio de la cadena de control se encuentra una variable:

$$\begin{aligned} &< [\,], [(x, ((suc, SUC), z, z)), (suc, SUC)], x, ([\,], (suc, SUC), [\,]) > \\ \underset{SECD}{\mapsto} &< ((suc, SUC), z, z), [(x, ((suc, SUC), z, z)), (suc, SUC)], [\,], ([\,], (suc, SUC), [\,]) > \end{aligned}$$

El valor correspondiente deberá ser buscado en el ambiente, en este caso la variable que debe buscarse es x y su valor correspondiente en el ambiente es la cerradura $((suc, SUC), z, z)$, así pues este valor es transferido al tope de la pila s. En Haskell:

```
1 *SECD> run_n_steps example_run 5
2 ([Closure (Env "succ" Succ Empty) "z"  z ],
3 Env "x" (Closure (Env "succ" Succ Empty) "z"  z ) (Env "succ" Succ Empty),
4 [],
```

```
5 [([],Env "succ" Succ Empty,[])])
```

Código 5.36: Estado de la máquina al haber realizado cinco pasos de la transición

Como no quedan elementos en la cadena de control, existe un elemento al tope de la pila y hay un elemento en el componente de descarga procedemos aplicando la regla 2), esto significa el retorno de una función:

$$\begin{aligned} & < ((suc, SUC), z, z), [(x, ((suc, SUC), z, z)), (suc, SUC)], [\,], ([\,], (suc, SUC), [\,]) > \\ \underset{SECD}{\mapsto} & < ((suc, SUC), z, z), (suc, SUC), [\,], [\,] > \end{aligned}$$

Reestablecemos la pila, el ambiente y la cadena de control guardadas en el componente de descarga agregando al inicio de la pila el valor de retorno de nuestra función, en este caso la cerradura $((suc, SUC), z, z)$. Implementado en Haskell:

```
1 *SECD> run_n_steps example_run 6
2 ([Closure (Env "succ" Succ Empty) "z"  z ],
3 Env "succ" Succ Empty,
4 [],
5 [])
```

Código 5.37: Estado de la máquina al haber realizado seis pasos de la transición

Por último notemos que por la regla 1) se ha llegado a un estado final de la forma $< [v], e, [\,], [\,] >$ con v siendo la cerradura que representa a $\lambda z.z$ que es el resultado esperado para la reducción de $(\lambda x.x)\ (\lambda z.z)$.

En conjunto todo el proceso de reducción de la expresión implementado en Haskell:

```
1  *SECD> run_n_steps example_run 0
2  ([],
3  Env "succ" Succ Empty,
4  [T (L x . x  L z . z )],
5  [])
6
7  *SECD> run_n_steps example_run 1
8  ([],
9  Env "succ" Succ Empty,
10 [T L z . z ,T L x . x ,Apply],
11 [])
12
13 *SECD> run_n_steps example_run 2
14 ([Closure (Env "succ" Succ Empty) "z"  z ],
```

```
Env "succ" Succ Empty,
[T L x . x ,Apply],
[])

*SECD> run_n_steps example_run 3
([Closure (Env "succ" Succ Empty) "x"  x,Closure (Env "succ" Succ Empty) "
   z" z],
Env "succ" Succ Empty,
[Apply],
[])

*SECD> run_n_steps example_run 4
([],
Env "x" (Closure (Env "succ" Succ Empty) "z"  z ) (Env "succ" Succ Empty),
[T  x ],
[([],Env "succ" Succ Empty,[])])

*SECD> run_n_steps example_run 5
([Closure (Env "succ" Succ Empty) "z"  z ],
Env "x" (Closure (Env "succ" Succ Empty) "z"  z ) (Env "succ" Succ Empty),
[],
[([],Env "succ" Succ Empty,[])])

*SECD> run_n_steps example_run 6
([Closure (Env "succ" Succ Empty) "z"  z ],
Env "succ" Succ Empty,
[],
[])
```

Código 5.38: Proceso de reducción de la expresión $(\lambda x.x)\ (\lambda z.z)$.

En esta sección se implementó el tipo `Term` que corresponde al cálculo lambda puro, se explicó que es y para que se usa un *tipo paramétrico* que se usó al implementar el tipo `Environment` correspondiente a los *ambientes* utilizados para la máquina SECD, después se implementó la función *lookup* para buscar valores en el ambiente. Se definió el tipo `Value` que representa a los valores de la máquina junto con el tipo `Directive` que puede ser tanto un $\lambda - término$ como la directiva APPLY emulando así a la cadena de control. Usando los tipos anteriormente descritos se implementaron nuevos tipos para cada uno de los componentes de la máquina SECD. Se explicaron una a una las implementaciones y definiciones de las reglas que componen la función de transición `run` y se dio un ejemplo específico de su uso. Para una mayor comprensión del proceso de reducción se explicó cómo es que la función `run` va cambiando el estado de la máquina conforme va avanzando en cada uno de los pasos hasta

llegar a un estado final. La exposición de todos estos aspectos de la implementación son de gran ayuda para que el lector pueda comprender el funcionamiento de la máquina viendo de forma transparente como se definen y transforman sus estados con el objetivo de reducir un término a un resultado.

5.2. Máquina de Krivine

En este capítulo se implementan las definiciones y funciones correspondientes a la máquina de Krivine.

```
1 module Krivine where  -- Definicion del modulo CEK a implementar
```

Código 5.39: Definición del módulo.

5.2.1. Notación De Bruijn

Recordemos de la definición 15 que los λ-términos en notación De Bruijn están definidos de forma inductiva como el mínimo conjunto que cumpla lo siguiente:

1. *Cualquier número natural (mayor que cero) es un término*
2. *Si M y N son términos entonces $(M\ N)$ es un término*
3. *Si M es un término entonces (λM) es un término*

Procedemos a implementar las tres partes correspondientes a la definición de λ-términos en notación De Bruijn: abstracciones, aplicaciones de función y los índices de Bruijn.

```
1 data Term = Index {num :: Int} -- Indices
2         | App {fun::Term, arg::Term} -- Aplicacion
3         | Lam {body::Term}  -- Abstracciones
4
5 instance Show Term where
6   show (Index n) = show n
7   show (App m n) = "(" ++ (show m) ++ " " ++ (show n) ++ ")"
8   show (Lam b) = "L." ++ (show b)
```

Código 5.40: Definición del tipo *Term* que representa λ-términos escritos en notación De Bruijn

Algunos ejemplos de λ-término en notación De Bruijn construidos usando el tipo de dato `Term` serían los siguientes

Estándar	Índices De Bruijn
$\lambda x.x$	$\lambda.1$
$\lambda x.\lambda y.x$	$\lambda.\lambda.2$
$\lambda x.\lambda y.\lambda s.\lambda z.xs(ysz)$	$\lambda.\lambda.\lambda.\lambda.4\ 2(3\ 2\ 1)$

Qué se traducen como:

```
*Krivine> ex1 = (Lam (Index 1))
*Krivine> ex1
L.1
```

Código 5.41: λ-término ($\lambda x.x$) escrito en notación De Bruijn como ($\lambda.1$) usando el tipo *Term*

```
*Krivine> ex2 = (Lam (Lam (Index 2)))
*Krivine> ex2
L.L.2
```

Código 5.42: λ-término ($\lambda x.\lambda y.x$) escrito en notación De Bruijn como ($\lambda.\lambda.2$) usando el tipo *Term*

```
*Krivine> ex3 = (Lam (Lam (Lam (Lam (App (App (Index 4) (Index 2)) (App (
    App (Index 3) (Index 2)) (Index 1)))))))
*Krivine> ex3
L.L.L.L.((4 2) ((3 2) 1))
```

Código 5.43: λ-término ($\lambda x.\lambda y.\lambda s.\lambda z.xs(ysz)$) escrito en notación De Bruijn como ($\lambda.\lambda.\lambda.\lambda.4\ 2(3\ 2\ 1)$) usando el tipo *Term*

5.2.2. Cerraduras

Sabemos por la semántica operacional que las funciones pueden ser un valor, el concepto de *cerradura* surgió ante la pregunta *¿qué valor representa una función?*. Para ver una función como un valor debemos conocer todo lo necesario para poder aplicarla a sus argumentos, empaquetando todos los componentes de la función en una estructura que indique como procesar los argumentos correctamente. Sus componentes son el nombre del argumento de la función, el cuerpo de la misma y el ambiente que recuerda las sustituciones previas. La máquina de Krivine hace uso de las cerraduras mediante su representación en notación De Bruijn dada en la definición 16. Las cerraduras en notación De Bruijn $\mathcal{R}$, representadas con el tipo `Closure` se definen recursivamente como un término `Term`, y una lista de cerraduras `[Closure]`.

```
-- Cerraduras
data Closure = C {term::Term, env::[Closure]}
```

```
3
4 instance Show Closure where
5   show (C t []) = (show t)
6   show (C (Index n) env) = (show n) ++ (show env)
7   show (C t env) = "(" ++ (show t) ++ ")" ++ (show env)
```

Código 5.44: Definición del tipo **Closure** que representa a las cerraduras.

Un ejemplo puede ser la cerradura $(1\ 2)[\lambda.1\ 1[\ ], \lambda.1[\ ]]$ que en su representación como `Closure` sería:

```
1 *Krivine> C (App (Index 1) (Index 2)) [C (Lam (App (Index 1) (Index 1)))
     [], C (Lam (Index 1)) []]
2 ((1 2))[L.(1 1),L.1]
```

Código 5.45: Cerradura construida usando el tipo **Closure**.

5.2.3. Reglas de transición

Los componentes del estado interno de la máquina de Krivine son $< \rho, M, s >$ siendo ρ un ambiente, M un λ-término en notación De Bruijn y s una lista de cerraduras. Para definir las transiciones de la máquina de Krivine no es necesario definir los componentes del estado interno de manera explícita creando un tipo en cada caso debido a que tanto el componente para el ambiente ρ, como el que se usa como pila de resultados intermedios s, son una lista de cerraduras y pueden representarse como `[Closure]`. En cuanto a la cadena de control, ya que contiene un λ-término en notación De Bruijn se usará el tipo `Term`.
La reducción de un término se lleva a cabo usando la codificación de las reglas de transición en la función `run`. Se puede observar que las reglas son muy parecidas a las presentes en la definición 19.
Definimos las reglas que dictan el funcionamiento de la máquina de Krivine con la función `run` la cual toma un estado y va avanzando paso a paso hasta llegar a un estado final.

```
1 run :: [Closure] -> Term -> [Closure] -> ([Closure] , Term , [Closure])
```

Código 5.46: Declaración de tipo de la función *run*.

Procedemos a implementar cada una de las reglas que integran la máquina de Krivine (Definición 19).

La primera regla indica que si la expresión a evaluar es una aplicación entonces se guarda en la pila la cerradura del argumento y se procede a evaluar la función correspondiente a la aplicación. La definición es:

$$< \rho, M\ N, S > \underset{kr}{\mapsto} < \rho, M, N[\rho] :: S >$$

la implementación correspondiente sería:

```
1 run env (App t0 t1) s = run env t0  ((C t1 env):s) -- (* 1 *)
```

Código 5.47: Implementación de la primera regla de transición.

La segunda regla establece la acción a tomar cuando se encuentra una abstracción en la cadena de control, en ese caso el cuerpo de la abstracción pasa a la cadena de control y el elemento en la cabeza de la pila pasa a extender el ambiente actual. Tanto esta regla como la primera se combinan para reducir una aplicación de función, lo primero que pasa es que al encontrar una aplicación de función el argumento pasa a la pila y la abstracción se reduce a una expresión lambda, luego en la segunda regla el argumento en la pila pasa a extender el ambiente de manera que el cuerpo de la función pueda usarlo al ser evaluado. La regla es:

$$< \rho, \lambda M, u :: S > \underset{kr}{\mapsto} < u{\cdot}\, \rho, M, S >$$

la implementación es:

```
1 run env (Lam t) (u:s) = run (u:env) t  s           -- (* 2 *)
```

Código 5.48: Implementación de la segunda regla de transición.

La tercera y cuarta regla buscan recursivamente en el ambiente el valor correspondiente al índice en la cadena de control. La tercera regla hace una llamada recursiva restando uno al índice y excluyendo el tope de la pila avanzando así un paso en el proceso de búsqueda. La regla es:

$$< u{\cdot}\, \rho, n+1, S > \underset{kr}{\mapsto} < \rho, n, S >$$

En Haskell:

```
1 run (u:env) (Index n) s = run env (Index (n-1)) s  -- (* 3 *)
```

Código 5.49: Implementación de la tercera regla de transición.

La cuarta regla representa el caso base de la recursión, cuando ya se ha encontrado el índice deseado se continúa la evaluación usando el ambiente y el cuerpo de la función que pertenecen a la cerradura en el tope de la pila:

$$M[v]{\cdot}\, \rho, 1, S > \underset{kr}{\mapsto} < v, M, S >$$

En Haskell:

```
1 run ((C t v):env) (Index 1) s = run v t s          -- (* 4 *)
```

Código 5.50: Implementación de la cuarta regla de transición.

Adicionalmente a las reglas que se encuentran en la definición 19 se implementan dos reglas más que corresponden a los posibles estados finales de la máquina de Krivine. Los estados finales de la máquina son de la forma $< \rho, \lambda M, [\,] >$ o bien de la forma $< [\,], n, S >$. Estados de la primera forma corresponden a λ-términos de la forma:

$$\lambda x.M$$

la implementación sería:

```
run env (Lam t) [] = ( env, (Lam t), []) -- (* Estado final Lx.M *)
```

Código 5.51: Implementación para estados finales de la forma $\lambda x.M$.

Y estados de la segunda forma corresponden a λ-términos de la forma:

$$xM_1...M_n$$

donde los términos a la derecha están en la pila S de la máquina. En Haskell:

```
run [] (Index n) s = ( [], (Index n), s) -- (* Estado final xM_1M_n *)
```

Código 5.52: Implementación para estados finales de la forma $xM_1...M_n$.

En conjunto todas las reglas de transición de la máquina de Krivine serían:

```
run :: [Closure] -> Term -> [Closure] -> ([Closure] , Term , [Closure])
run env (Lam t) [] = ( env, (Lam t), [])      -- (* Estado final Lx.M *)
run [] n s = ( [], n, s)                    -- (* Estado final xM_1M_n *)
run env (App t0 t1) s = run env t0  ((C t1 env):s)           -- (* 1 *)
run env (Lam t) (u:s) = run (u:env) t  s                     -- (* 2 *)
run ((C t v):env) (Index 1) s = run v t s                    -- (* 4 *)
run (u:env) (Index n) s = run env (Index (n-1)) s            -- (* 3 *)
```

Código 5.53: Implementación de todas las reglas de transición de la máquina de Krivine.

Debemos apuntar que la tercera y cuarta regla están cambiadas de orden para evitar que la cuarta regla que es más general impida que la tercera regla se ejecute ya que esto podría generar que se llegue a estados erróneos como por ejemplo en la siguiente ejecución:

```
*Krivine> t1 = App (Lam (App (Index 1) (Index 1))) (Lam (Index 1))
*Krivine> run [] t1 []
([],0,[1[L.1]])
```

Código 5.54: Ejemplo de una ejecución que llega a un estado erróneo.

El ejemplo anterior se ejecutó escribiendo la tercera y cuarta regla en el orden correspondiente y se llegó a un estado que contiene un índice igual a cero, esto es un error pues en la definición 15 se estipula que un índice debe ser mayor a cero.

5.2.4. Reducción paso a paso

Considere la siguiente reducción paso a paso utilizando las reglas de transición presentes en la Definición 15:

$$
\begin{aligned}
&< [\,], (\lambda.1\ 1)(\lambda.1), [\,] > && \text{usando 1)}\\
&\rightarrow < [\,], \lambda.1\ 1, \lambda.1 > && \text{usando 2)}\\
&\rightarrow < \lambda.1, 1\ 1, [\,] > && \text{usando 1)}\\
&\rightarrow < \lambda.1, 1, 1[\lambda.1] > && \text{usando 4)}\\
&\rightarrow < [\,], \lambda.1, 1[\lambda.1] > && \text{usando 2)}\\
&\rightarrow < 1[\lambda.1], 1, [\,] > && \text{usando 4)}\\
&\rightarrow < \lambda.1, 1, [\,] > && \text{usando 4)}\\
&\rightarrow < [\,], \lambda.1, [\,] >
\end{aligned}
$$

Ahora veamos como luce esta reducción paso a paso en nuestra codificación con Haskell, para ello utilizaremos un par de funciones auxiliares que nos facilitarán ver el progreso de la ejecución: la función `run_step` que va de un estado a otro de la máquina ejecutando un solo paso de la transición y también la función `run_n_steps` que ejecuta una cantidad n de pasos dados como argumento en la expresión. Su implementación:

```
1 run_step :: ([Closure], Term, [Closure]) -> ([Closure], Term, [Closure])
2 run_step (env, (Lam t), []) = (env, (Lam t), [])-- (* Estado final Lx.M *)
3 run_step ([], n, s) = ([], n, s)              -- (* Estado final xM_1M_n *)
4 run_step (env, (App t0 t1), s) = (env, t0, ((C t1 env):s))      -- (* 1 *)
5 run_step (env, (Lam t), (u:s)) = ((u:env), t, s)                -- (* 2 *)
6 run_step (((C t v):env), (Index 1), s) = (v, t, s)              -- (* 4 *)
7 run_step ((u:env), (Index n), s) = (env, (Index (n-1)), s)      -- (* 3 *)
```

Código 5.55: Definición de la función *run_step*.

```
1 run_n_steps :: ([Closure], Term, [Closure]) -> Integer -> ([Closure], Term
    , [Closure])
2 run_n_steps state 0 = state
3 run_n_steps state n = run_n_steps (run_step state) (n-1)
```

Código 5.56: Definición de la función *run_n_steps*.

Es importante señalar que `run_step` es muy similar a `run` siendo la única diferencia que la primera no ejecuta una llamada recursiva y la segunda sí, así es como avanza solo un paso en la reducción sin continuar hasta el final del proceso.

Procedemos ahora implementar la expresión $(\lambda.1\ 1)(\lambda.1)$ en Haskell:

```
t1 = App  (Lam
             (App
                 (Index 1)
                 (Index 1)))
          (Lam (Index 1))
```

Código 5.57: Implementación de la expresión $(\lambda.1\ 1)(\lambda.1)$ usando el tipo *Term*

Ahora para la ejecución paso a paso, el principio refleja el estado inicial de la máquina ya que aún no se ha realizado ninguna transición. Para este término es el estado $< [\,], (\lambda.1\ 1)(\lambda.1), [\,] >$. En Haskell:

```
*Krivine> run_n_steps ([],  t1, []) 0
([],  (L.(1 1) L.1),  [])
```

Código 5.58: Estado de la máquina al no haber realizado transiciones.

Según la reducción mencionada al principio de la sección el primer paso de la reducción sería aplicar la regla 1), esto significa reducir una aplicación. Se debe colocar el argumento en la pila y la función quedará en la cadena de control. Este paso luce de la siguiente manera:

$$< [\,], (\lambda.1\ 1)(\lambda.1), [\,] > \underset{kr}{\mapsto} < [\,], \lambda.1\ 1, \lambda.1 >$$

En la implementación con Haskell el término resultante, esto es, el término del lado derecho sería:

```
*Krivine> run_n_steps ([],t1,[]) 1
([],  L.(1 1),  [L.1])
```

Código 5.59: Estado de la máquina al haber realizado un paso de la transición.

El siguiente paso de la reducción sería tomar el resultado anterior y aplicar la regla 2) pues tenemos una abstracción funcional en la cadena de control:

$$< [\,], \lambda.1\ 1, \lambda.1 > \underset{kr}{\mapsto} < \lambda.1, 1\ 1, [\,] >$$

Ahora el argumento que antes estaba en la pila pasó a fomar parte del ambiente y el cuerpo de la función procede a ser la siguiente expresión a evaluar pues fue colocado en la cadena de control. En Haskell:

```
*Krivine> run_n_steps ([],t1,[]) 2
([L.1], (1 1),  [])
```

Código 5.60: Estado de la máquina al haber realizado dos pasos de la transición.

Continuamos utilizando la regla 1) nuevamente pues en la cadena de control se tiene una aplicación de función:

$$< \lambda.1, 1\ 1, [\,] > \underset{kr}{\mapsto} < \lambda.1, 1, 1[\lambda.1] >$$

La abstracción queda en la cadena de control mientras que el argumento, cargando consigo el ambiente actual correspondiente, pasa a formar parte de la pila. Implementado se vería de la siguiente manera:

```
*Krivine> run_n_steps ([],t1,[]) 3
([L.1], 1,  [1[L.1]])
```

Código 5.61: Estado de la máquina al haber realizado tres pasos de la transición.

Como en la cadena de control ahora se tiene un índice y además es el 1, es momento de usar la regla 4):

$$< \lambda.1, 1, 1[\lambda.1] > \underset{kr}{\mapsto} < [\,], \lambda.1, 1[\lambda.1] >$$

La cerradura correspondiente al índice que está en el tope del ambiente define la siguiente configuración del estado. El término de la cerradura pasa a la cadena de control mientras que el ambiente de la cerradura se convierte en el nuevo ambiente a utilizar, la pila se mantiene sin cambios. En nuestra implementación:

```
*Krivine> run_n_steps ([],t1,[]) 4
([],L.1,[1[L.1]])
```

Código 5.62: Estado de la máquina al haber realizado cuatro pasos de la transición.

Lo siguiente es aplicar la regla 2) ya que tenemos una abstracción funcional en la cadena de control:

$$< [\,], \lambda.1, 1[\lambda.1] > \underset{kr}{\mapsto} < 1[\lambda.1], 1, [\,] >$$

Ahora el argumento en la pila pasa al ambiente y el cuerpo de la función procede a ser la siguiente expresión a evaluar. En Haskell:

```
*Krivine> run_n_steps ([],t1,[]) 5
([1[L.1]],1,[])
```

Código 5.63: Estado de la máquina al haber realizado cinco pasos de la transición.

La regla 4) es la que se encarga de formar la siguiente expresión a evaluar usando la cerradura en el tope del ambiente pues en la cadena de control nos econtramos con el índice 1:

$$< 1[\lambda.1], 1, [\,] > \underset{kr}{\mapsto} < \lambda.1, 1, [\,] >$$

El término de la cerradura al inicio del ambiente pasa a la cadena de control y el ambiente de la cerradura sobrescribe al ambiente anterior. En Haskell:

```
*Krivine> run_n_steps ([],t1,[]) 6
([L.1],1,[])
```

Código 5.64: Estado de la máquina al haber realizado seis pasos de la transición.

Finalmente se utiliza la regla 4) al encontrarse nuevamente el índice 1 en la cadena de control:

$$< \lambda.1, 1, [\,] > \underset{kr}{\mapsto} < [\,], \lambda.1, [\,] >$$

El término de la cerradura al inicio del ambiente pasa a la cadena de control al ser vacío el ambiente de la cerradura es el que queda en el estado final. En Haskell:

```
*Krivine> run_n_steps ([],t1,[]) 7
([],L.1,[])
```

Código 5.65: Estado de la máquina al haber realizado seis pasos de la transición.

Notemos que se ha llegado a un estado final de la forma $< \rho, \lambda M, [\,] >$, más específicamente al estado $< [\,], \lambda.1, [\,] >$ que corresponde al λ-término $\lambda x.x$ que es la función identidad, el resultado esperado.

En conjunto todo el proceso de reducción de la expresión implementado en Haskell:

```
*Krivine> t1 = App (Lam (App (Index 1) (Index 1))) (Lam (Index 1))
*Krivine> run_n_steps ([],t1,[]) 0
([],(L.(1 1) L.1),[])
*Krivine> run_n_steps ([],t1,[]) 1
([],L.(1 1),[L.1])
*Krivine> run_n_steps ([],t1,[]) 2
([L.1],(1 1),[])
*Krivine> run_n_steps ([],t1,[]) 3
([L.1],1,[1[L.1]])
*Krivine> run_n_steps ([],t1,[]) 4
([],L.1,[1[L.1]])
*Krivine> run_n_steps ([],t1,[]) 5
([1[L.1]],1,[])
*Krivine> run_n_steps ([],t1,[]) 6
([L.1],1,[])
*Krivine> run_n_steps ([],t1,[]) 7
```

```
17 ([],L.1,[])
```

Código 5.66: Proceso de reducción de la expresión $(\lambda.1\ 1)(\lambda.1)$.

En esta sección se recapituló como se escriben los λ-términos usando la *notación De Bruijn*, se implementó el tipo `Term` que corresponde a los λ-términos usando dicha notación y se dieron algunos ejemplos. Se mencionó como surgió el concepto de *cerradura*, su utilización y sus componentes asi como su implementación en Haskell mediante el tipo `Closure` junto con un ejemplo, todo esto usando la notación De Bruijn. Se dió una breve explicación de cada uno de los componentes que conforman el estado interno de la máquina de Krivine y como se implementaría dicho estado usando el tipo `[Closure]` y el tipo `Term`. Se explicaron una a una las implementaciones y definiciones de las reglas que componen la función de transición `run`. Para una mayor comprensión del proceso de reducción se explicó como es que la función `run` va cambiando el estado de la máquina conforme va avanzando en cada uno de los pasos hasta llegar a un estado final. La exposición de todos estos aspectos de la implementación tienen como objetivo comprender el funcionamiento de la máquina viendo de forma transparente cómo se definen y transforman sus estados hasta llegar a un resultado.

5.3. Máquina CEK

En este capítulo se implementan las definiciones y funciones correspondientes a la máquina CEK.

```
1 module CEK where  -- Definicion del modulo CEK a implementar
2 import Data.Maybe -- Importacion del tipo Maybe
```

Código 5.67: Definición del módulo e importación del tipo **Maybe**.

5.3.1. El Cálculo-λ_c

El cálculo lambda modela funciones describiendo sus reglas en lugar de solamente sus gráficas sin embargo a veces resulta limitado pensar y modelar programas de esta manera. Por ejemplo si a mitad de una evaluación recursiva el programa encuentra su resultado o sucede algún error el programa debería ser capaz de reportarlo *de inmediato*. Este tipo de comportamiento es muy común y fácilmente implementado dentro del paradigma imperativo utilizando *operadores de control* tales como *catch*, *throw*, *return*, *goto* entre otros. El problema es que las funciones necesitan más *control* sobre su evaluación si van a ser usadas como modelo para programas computacionales. Las *continuaciones* pretenden solucionar este problema. Las continuaciones vuelven explícito y manejable el estado de control de un programa

en un punto de su ejecución para que así se pueda tener acceso a él en lugar de permanecer oculto en el ambiente de ejecución.
Presentamos una extensión del cálculo lambda que incorpora una funcionalidad capaz de manipular continuaciones más fácilmente. Los λ-términos extendidos para incluir continuaciones, es decir, el conjunto que nombramos como Λ_c están construidos en su mayoría de la misma forma que los λ-términos en el cálculo lambda simple, esto es variables, abstracciones y aplicaciones. Sin embargo además de estás primitivas básicas se añaden dos más que permitirán hacer uso explícito de las continuaciones en un término de la expresión, estás primitivas son las $\mathcal{C}$-aplicaciones y las $\mathcal{A}$-aplicaciones [26].
El significado de las construcciones derivadas del cálculo lambda puro se mantiene de manera aproximada, por ejemplo una variable representa un valor y una abstracción corresponde a una función. Sin embargo la aplicación además de invocar funciones en un argumento también puede invocar continuaciones en un argumento. Hablando de las nuevos tipos de aplicaciones la $\mathcal{A}$-aplicación aborta el cálculo actual y comienza uno nuevo con su argumento como punto de partida y la $\mathcal{C}$-aplicación captura la continuación actual del programa y la pasa como una abstracción funcional a su argumento [26].

Siguiendo la definición 20 la implementación de los λ_c-términos usando el tipo `Term` es la siguiente:

```
-- Lambda_c

type Ide = String
data Term = Var {name :: Ide} -- Variables
          | Lam {param::Ide, body::Term}  -- Abstracciones
          | App {fun::Term, arg::Term} -- Aplicacion
          | CApp {arg::Term} -- C-aplicacion
          | AbApp {arg::Term} -- A-aplicacion
  deriving (Eq, Show)
```

Código 5.68: Definición del tipo **Term** que representa a los λ_c-términos.

Un ejemplo de λ_c-término construido usando el tipo de dato `Term` sería $((\mathcal{C}x)(yz))$ que se traduce como:

```
(App                    --   (
    (CApp (Var "x"))    --    (C x)
    (App                --    (
      (Var "y")         --      y
      (Var "z")))       --      z))
```

Código 5.69: Implementación del λ_c-término $((\mathcal{C}x)(yz))$ usando el tipo **Term**.

Otro ejemplo más complejo es el siguiente $(\mathcal{C}(\lambda.x(\lambda.y((xy)(\mathcal{A}y)))))$ donde:

```
(CApp                        -- (C
  (Lam "x"                   --    (L.x
    (Lam "y"                 --       (L.y
      (App                   --         (
        (App                 --           (
          (Var "x")          --             x
          (Var "y")          --             y
        )                    --           )
        (AbApp               --           (A
          (Var "y"))))))     --             y)))))
```

Código 5.70: Implementación del λ_c-término $(\mathcal{C}(\lambda.x(\lambda.y((xy)(\mathcal{A}y)))))$ usando el tipo **Term**.

5.3.2. Ambientes

Los ambientes se construyen recursivamente utilizando un *tipo paramétrico*. Un ambiente puede ser vacío o puede ser un identificador y su valor junto con otro ambiente.

```
-- Ambientes

data Environment a = Empty
      | Env Ide a (Environment a)
      deriving (Eq, Show)
```

Código 5.71: Definición del tipo **Environment** que representa a los ambientes.

Adicionalmente se define la función *lookup* (buscar) la cual busca recursivamente valores en el ambiente usando un identificador. Esta definición hace uso del tipo `Maybe` para identificar cuando existe una variable libre en el término y arrojar un error.

```
elookup :: Ide -> Environment a -> Maybe  a
elookup x (Empty) = Nothing
elookup x (Env y a env)
  | x == y = Just a
  | x /= y = elookup x env
```

Código 5.72: Definición de la función de búsqueda en ambientes **elookup**.

Por ejemplo, el ambiente $[(x, 5), (y, 2), (z, 4)]$ utiliza como argumento del tipo paramétrico el tipo `Integer`:

```
ejemplo_ambiente = Env "x" 5 (Env "y" 2 (Env "z" 4 Empty))
```

Código 5.73: Ejemplo de definición de un ambiente usando el tipo **Environment**.

La función *lookup* aplicada a y deberia darnos un valor de 2:

```
*CEK> elookup "y" ejemplo_ambiente
Just 2
```

Código 5.74: Ejemplo de uso de la función **elookup**.

Mientras que la función *lookup* aplicada a w, una variable libre, deberia darnos un valor indefinido:

```
*CEK> elookup "w" ejemplo_ambiente
Nothing
```

Código 5.75: Ejemplo de uso de la función **elookup** en una variable libre.

5.3.3. Códigos de continuación

Las continuaciones representan el resto del cálculo que se debe hacer en un punto de la evaluación, la forma en que esto se representa en la máquina CEK es mediante lo que llamamos *códigos de continuación CC* (del inglés *continuation code*).

Los códigos de continuación dictan aquello que la máquina tendrá que hacer posteriormente a evaluar el término actual, de acuerdo con la definición 24 se conforman de dos partes las p-continuaciones y las ret-continuaciones.

Las p-continuaciones pueden ser la continuación vacía que termina el cálculo **(stop)** o bien contener únicamente otra p-continuación **(** k **cont)**, igualmente pueden representar una aplicación de función **(** k **arg** N ρ **)** o representar una abstracción funcional **(** k **fun F)**. En Haskell:

```
-- p-cont
data PCont a = STOP
    | CONT (PCont a)
    | ARG (PCont a) Term (Environment a)
    | FUN (PCont a) a
    deriving (Show, Eq)
```

Código 5.76: Definición del tipo **PCont** representando a las p-continuaciones.

Por ejemplo, la p-continuación (**(stop) arg** $\lambda z.z$ [ ]) implementada es:

```
*CEK> pc = ARG STOP (Lam "z" (Var "z")) Empty
```

Código 5.77: Ejemplo de p-continuación implementada usando el tipo **PCont**.

Las ret-continuaciones indican que se ha llegado a un valor y por lo tanto es posible proceder con la evaluación utilizando la continuación actual. Las ret-continuaciones son de la forma (k **ret** V) siendo k una p-continuación y V un valor. Su implementación es:

```haskell
-- ret-cont
data RetCont a = RET (PCont a) a
  deriving (Show ,Eq)
```

Código 5.78: Definición del tipo **RetCont** representando a las *ret*-continuaciones.

Un ejemplo es la *ret*-continuación $(((\textbf{stop})\ \textbf{fun}\ <[\,],x,x>)\ \textbf{ret}\ <[\,],y,y>)$. En Haskell:

```haskell
*CEK> rc = RET ( FUN (STOP) "<[ ], x, x>" ) "<[ ], y, y>"
```

Código 5.79: Ejemplo de *ret*-continuación implementada usando el tipo **RetCont**.

Ahora juntamos ambas partes, las *p*-continuaciones y las *ret*-continuaciones en un solo tipo. El tipo `CC` representa los códigos de continuación y se implementa como:

```haskell
-- CC
data CC a = PC (PCont a) | RC (RetCont a)
  deriving (Show, Eq)
```

Código 5.80: Definición del tipo **CC** representando a los códigos de continuación.

Un ejemplo de código de continuación es $(((\textbf{stop})\ \textbf{fun}\ <[\,],x,x>)\ \textbf{ret}\ <[\,],y,y>)$, nótese que esta expresión es a su vez una *ret*-continuación. En Haskell:

```haskell
*CEK> cc = RC (RET (FUN STOP "<[ ], x, x>") "<[ ], y, y>")
```

Código 5.81: Ejemplo de codigo de continuación usando el tipo **CC**.

Todas las definiciones utilizan tipos paramétricos debido a que los componentes en esta máquina se definen de manera circular, es decir, dependen unos de otros. Es importante aclarar que se utilizó el tipo cadena `[Char]`, como tipo paramétrico para representar valores en esta sección debido a que el tipo que corresponde a estos será definido en la próxima sección.

5.3.4. Valores semánticos

La máquina CEK utiliza el ambiente como un mapeo finito del conjunto de variables al conjunto conformado de la unión de *cerraduras* y *puntos de continuación*, esta unión define los valores que usará la máquina y es llamado el conjunto de *valores semánticos* [26]. Las *cerraduras* son la representación de abstracciones funcionales vistas como valores y conformadas por una tripleta ordenada $< e,\ x,\ M >$ cuyos elementos corresponden al ambiente, argumento y cuerpo de la función respectivamente. Los *puntos de continuación* son estructuras que etiquetan continuaciones de la forma $<\mathbf{P}, k>$ donde $\mathbf{P}$ es la etiqueta y k es un *código de continuación*. Los *códigos de continuación* representan el resto del cálculo, esto es, lo que la máquina tendrá que hacer posteriormente a evaluar el término actual [26].

Definimos los dos posibles valores semánticos: los *puntos de continuación* y las *cerraduras*, usando el tipo `Value`:

```
-- Valores
data Value = CPoint (CC Value)                -- Puntos de continuacion
    | Closure (Environment Value) Ide Term    -- Cerraduras
  deriving (Show, Eq)
```

Código 5.82: Definición del tipo **Value** representando a los valores semánticos.

Considere la *ret*-continuación **(((stop) fun** $< [\,], x, x >$ **) ret** $< [\,], y, y >$ **)** para ejemplificar los valores. Esta *ret*-continuación contiene dos *valores semánticos*, ambos son cerraduras representando la función identidad. Procedemos a construirlos usando el tipo apropiado `Value`:

```
-- Primer valor
v1 = Closure Empty "x" (Var "x")
-- Segundo valor
v2 = Closure Empty "y" (Var "y")
```

Código 5.83: Definición de las cerraduras $< [\,],x,x >$ y $< [\,],y,y >$ usando el tipo **Value**.

Utilizando estos valores podemos definir una *ret*-continuación que utiliza el tipo `RetCont` como argumento del tipo paramétrico:

```
-- ret-continuacion
rc = (RET (FUN STOP (Closure Empty "x" (Var "x"))) (Closure Empty "x" (Var
    "x")))
```

Código 5.84: Definición de la *ret*-continuación **(((stop) fun** $< [\,],x,x >$ **) ret** $< [\,],y,y >$ **)** usando el tipo **RetCont**.

Sabemos de la sección anterior que una *ret*-continuación puede ser también un código de continuación, así que lo implementamos usando el tipo `CC`:

```
-- Codigo de continuacion
cc = RC (RET (FUN STOP (Closure Empty "x" (Var "x"))) (Closure Empty "x" (
   Var "x")))
```

Código 5.85: Definición de la *ret*-continuación **(((stop) fun** $< [\,],x,x >$ **) ret** $< [\,],y,y >$ **)** usando el tipo **CC**.

Finalmente para que **(((stop) fun** $< [\,], x, x >$ **) ret** $< [\,], y, y >$ **)** sea un valor hay que recordar que los valores incluyen no solo cerraduras sino tambien *puntos de continuación* y estos a su vez usan un código de continuación. Implementamos el punto de continuación $<$ **P**, **(((stop) fun** $< [\,], x, x >$ **) ret** $< [\,], y, y >$ **)** $>$ usando el tipo `Value` como:

```haskell
-- Punto de continuacion
pc = CPoint (RC (RET (FUN STOP (Closure Empty "x" (Var "x"))) (Closure
    Empty "x" (Var "x"))))
```

Código 5.86: Definición de $<$ **P**,**(((stop) fun** $<[\,],x,x>$ **) ret** $<[\,],y,y>$ **)** $>$ usando el tipo **Value**.

Representando así a la expresión **(((stop) fun** $<[\,],x,x>$ **) ret** $<[\,],y,y>$ **)** como un valor dentro de nuestra implementación.

5.3.5. Componentes de la máquina CEK

Implementamos los tres componentes que conforman la máquina CEK según aparecen en la definición 25. C es un λ_c-término o bien puede ser el símbolo $\updownarrow$, por lo que se usará el tipo `Term` junto con el constructor `UPDOWN`:

```haskell
-- Control
data C = T Term | UPDOWN deriving (Show, Eq)
```

Código 5.87: Definición del componente de control de la máquina CEK usando el tipo **C**.

E representa el ambiente actual de la máquina CEK, usamos el tipo paramétrico `Environment` anteriormente definido para su implementación. Nótese que el tipo `Value` se usa como argumento de `Environment` pues el ambiente contiene *valores semánticos*.

```haskell
-- Ambiente
type E = Environment Value
```

Código 5.88: Definición del componente que contiene al ambiente de la máquina CEK usando el tipo **E**.

K es la continuación actual de la máquina CEK representada usando un *código de continuación*, usamos el tipo paramétrico `CC` anteriormente definido para su implementación. Nuevamente el tipo `Value` aparece como un argumento de un tipo paramétrico, en este caso `CC`.

```haskell
-- Continuaciones
type K = CC Value
```

Código 5.89: Definición del componente de continuaciones de la máquina CEK usando el tipo **K**.

En conjunto los tres componentes que conforman el estado interno de la máquina CEK en su implementación con Haskell son:

```haskell
-- Definicion de los componentes
```

```haskell
data C = T Term | UPDOWN deriving (Show, Eq)  -- Control
type E = Environment Value                    -- Ambiente
type K = CC Value                             -- Continuaciones
```

Código 5.90: Definición del estado interno de la máquina CEK usando los tipos **C,E y K**.

Si queremos implementar con este esquema el estado $< \updownarrow, [\,], ((\textbf{stop})\ \textbf{ret}\ < [\,], x, x >) >$ de la máquina CEK debemos escribir cada uno de los componentes del siguiente modo:

```haskell
-- Estado de ejemplo
componenteC = UPDOWN
componenteE = Empty
componenteK = RC (RET STOP (Closure Empty "x" (Var "x")))
```

Código 5.91: Definición del estado $< \updownarrow, [\,], ((\textbf{stop})\ \textbf{ret}\ < [\,], x, x >) >$ usando los tipos **C,E y K**.

Otro estado que podríamos representar sería $< \lambda x.x, [\,], \textbf{(stop)} >$ como:

```haskell
-- Estado de ejemplo
componenteC = T (Lam "x" (Var "x"))
componenteE = Empty
componenteK = PC STOP
```

Código 5.92: Definición del estado $< \lambda x.x, [\,], \textbf{(stop)} >$ usando los tipos **C,E y K**.

5.3.6. Reglas de transición

Definimos las reglas que dictan el funcionamiento de la máquina CEK con la función `run` que toma un estado y va avanzando paso a paso hasta llegar a obtener un valor.

```haskell
run :: C -> E -> K -> Value
```

Código 5.93: Declaración de tipo de la función *run*.

Procedemos a implementar cada una de las reglas que integran la máquina CEK (Definición 26), las reglas de transición son muy semejantes a sus reglas teóricas correspondientes esto es por como se construyó cada uno de sus componentes.

La regla `(* Estado final *)` especifica que hacer si la cadena de control c es la directiva `UPDOWN`, el ambiente está vacío y el código de continuación en k es una *ret*-continuación conteniendo la continuación **(stop)**. Corresponde a terminar el cálculo: el valor en la *ret*-continuación V será el resultado final y se alcanzará el estado final además en la implementación se regresa el valor que corresponde al resultado. La definición del estado final es:

$$< \updownarrow , [\,], ((\textbf{stop})\ \textbf{ret}\ V) >$$

la implementación correspondiente es:

```
run UPDOWN Empty (RC (RET STOP v)) = v                -- (* Estado final *)
```

Código 5.94: Implementación del estado final de CEK.

La regla 1) indica que cuando hay una variable en la cadena de control se construye una nueva *ret*-continuación, esta se conforma de la continuación actual y el valor correspondiente a la variable en el ambiente. La definición es:

$$< x, \rho, k > \underset{CEK}{\mapsto} < \updownarrow , [\,] , (k\ \textbf{ret}\ lookup(x, \rho)) >$$

la implementación correspondiente busca valores en el ambiente e imprime un error si es que no encuentra el que se pide, apoyandose del tipo `Maybe`:

```
run (T (Var x)) e (PC k) =                                    -- (* 1 *)
  if isJust (elookup x e)
    then run UPDOWN Empty (RC  (RET  k (fromJust (elookup x e))))
    else error ("Failed to find variable name: " ++ x)
```

Código 5.95: Implementación de la regla de transición 1).

La regla 2) maneja el caso en el que una abstracción se encuentra en la cadena de control. De manera similar al caso anterior se construte una *ret*-continuación con la continuación actual y como valor se utiliza una cerradura representando a la abstracción en la cadena de control. La definición es:

$$< \lambda x.M, \rho, k > \underset{CEK}{\mapsto} < \updownarrow , [\,] , (k\ \textbf{ret}\ < \rho, x, M >) >$$

la implementación correspondiente:

```
run (T (Lam x m)) e (PC k) =                                  -- (* 2 *)
  run UPDOWN Empty (RC  (RET  k (Closure e x m)))
```

Código 5.96: Implementación de la regla de transición 2).

La regla 3) describe el procedimiento a seguir cuando hay una aplicación en la cadena de control. En este caso el término correspondiente a la abstracción en la aplicacion pasa a la cadena de control como el siguiente término a evaluar mientras que la continuación actual es utilizada para construir una *p*-continuación. La *p*-continuación guarda el argumento de la aplicación junto con el ambiente actual para su posterior evaluación. La definición es:

$$< MN, \rho, k > \underset{CEK}{\mapsto} < M, \rho, (k\ \textbf{arg}\ N\ \rho) >$$

la implementación correspondiente:

```
run (T (App m n)) e (PC k) =                                    -- (* 3 *)
  run (T m) e (PC (ARG k n e))
```

Código 5.97: Implementación de la regla de transición 3).

La regla 4) muestra qué hacer cuando ya se ha evaluado la abstracción en una aplicación. Dicha abstracción se encuentra en forma de cerradura en el valor de la *ret*-continuación F. Lo siguiente será guardar F en una nueva p-continuación y evaluar el argumento de la función que había sido guardado anteriormente en la regla 3). La definición es:

$$< \updownarrow , \ [\,] \, , ((k \ \textbf{arg} \ N \ \rho) \ \textbf{ret} \ F) > \underset{CEK}{\mapsto} < N, \rho, (k \ \textbf{fun} \ F) >$$

la implementación correspondiente:

```
run UPDOWN Empty (RC (RET  (ARG k n e) f)) =                    -- (* 4 *)
  run (T n) e (PC (FUN k f))
```

Código 5.98: Implementación de la regla de transición 4).

La regla 5) actúa sobre un estado en el que tanto la abstracción como el argumento de una aplicación han sido reducidos a valores, la abstracción toma la forma de una cerradura. Corresponde evaluar el cuerpo de la abstracción y extender el ambiente con el par conformado por el argumento de la abstracción y el valor que representa el argumento de la aplicación. La continuación será la que existía al momento de evalúar la aplicación. La definición es:

$$< \updownarrow , \ [\,] \, , ((k \ \textbf{fun} \ < \rho, x, M >) \ \textbf{ret} \ V) > \underset{CEK}{\mapsto} < M, \ ((x, V) : \rho) \ , k >$$

la implementación correspondiente:

```
run UPDOWN Empty (RC (RET  (FUN k (Closure e x m)) v)) =        -- (* 5 *)
  run (T m) (Env x v e) (PC k)
```

Código 5.99: Implementación de la regla de transición 5).

Las reglas 1) a 5) definen un evaluador de paso por valor clásico del cálculo lambda puro, esto es la máquina SECD. Los pasos 6) a 8) definen las operaciones necesarias para procesar una $\mathcal{C}$-aplicación.

En 6) la continuación se marca usando el código de continuación $(k \ \textbf{cont})$ y se coloca el $\mathcal{C}$-argumento M, en el componente de control para su evaluación. La definición es:

$$< \mathcal{C}M, \rho, k > \underset{CEK}{\mapsto} < M, \rho, (k \ \textbf{cont}) >$$

la implementación correspondiente:

```
run (T (CApp m)) e (PC k) =                                    -- (* 6 *)
  run (T m) e (PC (CONT k))
```

Código 5.100: Implementación de la regla de transición 6).

Posteriormente en 7) el término M a evaluar en 6) se ha convertido ya en una cerradura $< \rho, x, M >$ y se procede a evaluar el cuerpo de la cerradura ingresando antes la continuación al ambiente en forma de punto de continuación y asociándola a la variable x. Esencialmente la regla 7) encapsula la continuación actual en un punto de continuación y se lo pasa como argumento al $\mathcal{C}$-argumento. La definición es:

$$< \updownarrow , \ [\,] \, , ((k \ \textbf{cont}) \ \textbf{ret} \ < \rho, x, M >) > \underset{CEK}{\mapsto} < M, \ ((x, < \mathbf{P}, k >) : \rho) \, , \textbf{(stop)} >$$

la implementación correspondiente:

```
run UPDOWN Empty (RC (RET  (CONT k) (Closure e x m))) =        -- (* 7 *)
  run (T m) (Env x (CPoint (PC k)) e) (PC STOP)
```

Código 5.101: Implementación de la regla de transición 7).

El paso 8) reemplaza la continuación actual por la continuación inicial. La definición es:

$$< \updownarrow , \ [\,] \, , ((k \ \textbf{cont}) \ \textbf{ret} \ < \mathbf{P}, k_0 >) > \underset{CEK}{\mapsto} < \updownarrow , \ [\,] \, , (k_0 \ \textbf{ret} \ < \mathbf{P}, k >) >$$

la implementación correspondiente:

```
run UPDOWN Empty (RC (RET  (CONT k) (CPoint (PC k0)))) =       -- (* 8 *)
  run UPDOWN Empty (RC (RET k0 (CPoint (PC k))))
```

Código 5.102: Implementación de la regla de transición 8).

La regla 9) muestra que la invocación de una continuación remueve la continuación actual y utiliza la continuación anterior en su lugar. Esta regla implementa el comportamiento *inmediato* de las continuaciones que al ser usadas olvidan todo el contexto y se enfocan solo en su argumento. La definición es:

$$< \updownarrow , \ [\,] \, , ((k \ \textbf{fun} \ < \mathbf{P}, k_0 >) \ \textbf{ret} \ V) > \underset{CEK}{\mapsto} < \updownarrow , \ [\,] \, , (k_0 \ \textbf{ret} \ V) >$$

la implementación correspondiente:

```
run UPDOWN Empty (RC (RET  (FUN k (CPoint (PC k0))) v)) =      -- (* 9 *)
  run UPDOWN Empty (RC (RET k0 v))
```

Código 5.103: Implementación de la regla de transición 9).

Pór último la regla 10) referente a las $\mathcal{A}$-aplicaciones ignora la continuación y comienza la evaluación usando el $\mathcal{A}$-argumento implementando así el comportamiento de terminar el cálculo restante. La definición es:

$$< \mathcal{A}M, \rho, k > \underset{CEK}{\mapsto} < M, \rho, (\textbf{stop}) >$$

la implementación correspondiente:

```
run (T (AbApp m)) e (PC k) = run (T m) e (PC STOP)          -- (* 10 *)
```

Código 5.104: Implementación de la regla de transición 10).

En conjunto todas las reglas de transición de la máquina CEK son:

```
-- Definicion de las reglas de transicion
run :: C -> E -> K -> Value
run UPDOWN Empty (RC (RET STOP v)) = v            -- (* Estado final *)
run (T (Var x)) e (PC k) =                                  -- (* 1 *)
  if isJust (elookup x e)
    then run UPDOWN Empty (RC  (RET  k (fromJust (elookup x e))))
    else error ("Failed to find variable name: " ++ x)
run (T (Lam x m)) e (PC k) =                                -- (* 2 *)
  run UPDOWN Empty (RC  (RET  k (Closure e x m)))
run (T (App m n)) e (PC k) =                                -- (* 3 *)
  run (T m) e (PC (ARG k n e))
run UPDOWN Empty (RC (RET  (ARG k n e) f)) =                -- (* 4 *)
  run (T n) e (PC (FUN k f))
run UPDOWN Empty (RC (RET  (FUN k (Closure e x m)) v)) =    -- (* 5 *)
  run (T m) (Env x v e) (PC k)
run (T (CApp m)) e (PC k) =                                 -- (* 6 *)
  run (T m) e (PC (CONT k))
run UPDOWN Empty (RC (RET  (CONT k) (Closure e x m))) =     -- (* 7 *)
  run (T m) (Env x (CPoint (PC k)) e) (PC STOP)
run UPDOWN Empty (RC (RET  (CONT k) (CPoint (PC k0)))) =    -- (* 8 *)
  run UPDOWN Empty (RC (RET k0 (CPoint (PC k))))
run UPDOWN Empty (RC (RET  (FUN k (CPoint (PC k0))) v)) =   -- (* 9 *)
  run UPDOWN Empty (RC (RET k0 v))
run (T (AbApp m)) e (PC k) = run (T m) e (PC STOP)          -- (* 10 *)
```

Finalmente se define la función `eval` cuyo propósito es tomar un término y comenzar las transiciones desde un estado inicial de la forma $< M, [\,], \textbf{(stop)} >$ hasta llegar a un valor.

```
eval :: Term -> Value
eval t = run (T t) Empty (PC STOP)
```

5.3.7. Reducción paso a paso

Considere la siguiente reducción paso a paso utilizando las reglas de transición presentes en la definición 26, la reducción muestra la expresión identidad aplicada a si misma:

$$
\begin{aligned}
& < (\lambda x.x)(\lambda z.z), [\,], \textbf{(stop)} > && \text{usando (3)} \\
\underset{CEK}{\mapsto} & < \lambda x.x, [\,], (\textbf{(stop) arg } \lambda z.z\ [\,]) > && \text{usando (2)} \\
\underset{CEK}{\mapsto} & < \updownarrow, [\,], (((\textbf{(stop) arg } \lambda z.z\ [\,]) \textbf{ ret } < [\,], x, x >) > && \text{usando (4)} \\
\underset{CEK}{\mapsto} & < \lambda z.z, [\,], (\textbf{(stop) fun } < [\,], x, x >) > && \text{usando (2)} \\
\underset{CEK}{\mapsto} & < \updownarrow, [\,], (((\textbf{(stop) fun } < [\,], x, x >) \textbf{ ret} < [\,], z, z >) > && \text{usando (5)} \\
\underset{CEK}{\mapsto} & < x, [(x, < [\,], z, z >)], \textbf{(stop)} > && \text{usando (1)} \\
\underset{CEK}{\mapsto} & < \updownarrow, [\,], (\textbf{(stop) ret} < [\,], z, z >>
\end{aligned}
$$

Ahora procedemos a examinar paso a paso cómo opera esta codificación con el lenguaje de programación Haskell, para ello utilizaremos un par de funciones auxiliares que nos facilitarán ver el progreso de la ejecución: la función `run_step :: (C,E,K) -> (C,E,K)` que va de un estado a otro de la máquina ejecutando un solo paso de la transición y también la función `run_n_steps` que ejecuta una cantidad n de pasos dados como argumento en la expresión. Su implementacíón:

```
1  run_step :: (C,E,K) -> (C,E,K)
2  run_step (UPDOWN, Empty, (RC (RET STOP v))) =           -- Estado final *)
3    (UPDOWN, Empty, (RC (RET STOP v)))
4  run_step ((T (Var x)), e, (PC k)) =                                -- 1 *)
5      if isJust (elookup x e)
6        then (UPDOWN, Empty, (RC  (RET  k (fromJust (elookup x e)))))
7        else error ("Failed to find variable name: " ++ x)
8  run_step ((T (Lam x m)), e, (PC k)) =                              -- 2 *)
9    (UPDOWN, Empty, (RC  (RET  k (Closure e x m))))
10 run_step ((T (App m n)), e, (PC k)) =                              -- 3 *)
11   ((T m), e, (PC (ARG k n e)))
12 run_step (UPDOWN, Empty, (RC (RET  (ARG k n e) f))) =              -- 4 *)
13   ((T n), e, (PC (FUN k f)))
14 run_step (UPDOWN, Empty, (RC (RET  (FUN k (Closure e x m)) v))) =  -- 5 *)
15   ((T m), (Env x v e), (PC k))
16 run_step ((T (CApp m)), e, (PC k)) =                               -- 6 *)
17   ((T m), e, (PC (CONT k)))
18 run_step (UPDOWN, Empty, (RC (RET  (CONT k) (Closure e x m)))) =   -- 7 *)
```

```
19   ((T m), (Env x (CPoint (PC k)) e), (PC STOP))
20 run_step (UPDOWN, Empty, (RC (RET  (CONT k) (CPoint (PC k0))))) =  -- 8 *)
21   (UPDOWN, Empty, (RC (RET k0 (CPoint (PC k)))))
22 run_step (UPDOWN, Empty, (RC (RET  (FUN k (CPoint (PC k0))) v))) = -- 9 *)
23   (UPDOWN, Empty, (RC (RET k0 v)))
24 run_step ((T (AbApp m)), e, (PC k)) =                          -- (* 10 *)
25   ((T m), e, (PC STOP))
```

Código 5.105: Definición de la función *run_step*.

```
1 run_n_steps :: (C,E,K) -> Integer -> (C,E,K)
2 run_n_steps state 0 = state
3 run_n_steps state n = run_n_steps (run_step state) (n-1)
```

Código 5.106: Definición de la función *run_n_steps*.

Es importante señalar que `run_step` es muy similar a `run` siendo la única diferencia que la primera no ejecuta una llamada recursiva y la segunda sí, así es como avanza solo un paso en la reducción sin continuar hasta el final del proceso.

Procedemos ahora implementar la expresión $(\lambda x.x)\ (\lambda z.z)$ en Haskell:

```
1 id1 = (Lam "x" (Var "x"))
2 id2 = (Lam "z" (Var "z"))
3 example = (App id1 id2)
4 -- Poner la expresion en un estado inicial
5 example_run = ((T example), Empty, (PC STOP))
```

Código 5.107: Implementación de la expresión $(\lambda x.x)\ (\lambda z.z)$ usando el tipo *Term*

Ahora para la ejecución paso a paso, el principio refleja el estado inicial de la máquina ya que aún no se ha realizado ninguna transición. Para este término es el estado $< (\lambda x.x)(\lambda z.z), [\,], \textbf{(stop)} >$. En Haskell:

```
1 *CEK> run_n_steps example_run 0
2 (T (App {fun = Lam {param = "x", body = Var {name = "x"}}, arg = Lam {
     param = "z", body = Var {name = "z"}}}),
3 Empty,
4 PC STOP)
```

Código 5.108: Estado de la máquina al no haber realizado transiciones.

Las lineas 2, 3 y 4 del código representan a los componentes C, E y K de la máquina respectivamente. Dado que el término en la cadena de control C es una aplicación de función continuamos utilizando la regla 3) de la siguiente forma:

$$< (\lambda x.x)(\lambda z.z), [\,], \textbf{(stop)} > \underset{CEK}{\mapsto} < \lambda x.x, [\,], (\textbf{(stop) arg } \lambda z.z\ [\,]) >$$

El argumento de la aplicación es guardado en forma de continuación, en cuanto a la abstracción de la aplicación pasa a la cadena de control para su evaluación. En la implementación con Haskell el término resultante, esto es, el término del lado derecho es:

```
*CEK> run_n_steps example_run 1
(T (Lam {param = "x", body = Var {name = "x"}}),
Empty,
PC (ARG STOP (Lam {param = "z", body = Var {name = "z"}}) Empty))
```

Código 5.109: Estado de la máquina al haber realizado un paso de la transición.

Si se toma en cuenta que las últimas tres líneas del código representan los componentes de la máquina CEK podemos confirmar que los términos se corresponden. Procedemos a aplicar la regla 2) pues hay una abstracción en la cadena de control:

$$< \lambda x.x, [\,], (\textbf{(stop) arg } \lambda z.z\ [\,]) > \underset{CEK}{\mapsto} < \updownarrow, [\,], (((\textbf{(stop) arg } \lambda z.z\ [\,])\ \textbf{ret}\ < [\,], x, x >) >$$

La continuación actual se extiende con la cerradura que representa a la abstracción funcional en la cadena de control. En Haskell:

```
*CEK> run_n_steps example_run 2
(UPDOWN,
Empty,
RC (RET (ARG STOP (Lam {param = "z", body = Var {name = "z"}}) Empty) (
    Closure Empty "x" (Var {name = "x"}))))
```

Código 5.110: Estado de la máquina al haber realizado dos pasos de la transición.

La siguiente regla a aplicar es la 4) ya que hay una *ret*-continuación que contiene una continuación de argumento, ademas se tiene la directiva `UPDOWN` y el ambiente vacío:

$$\begin{aligned} & < \updownarrow, [\,], (((\textbf{(stop) arg } \lambda z.z\ [\,])\ \textbf{ret}\ < [\,], x, x >) > \\ \underset{CEK}{\mapsto} & < \lambda z.z, [\,], (\textbf{(stop) fun}\ < [\,], x, x >) > \end{aligned}$$

La cerradura se guarda en una continuación de función y se procede a evaluar el argumento de la aplicación. En Haskell:

```
*CEK> run_n_steps example_run 3
(T (Lam {param = "z", body = Var {name = "z"}}),
Empty,
PC (FUN STOP (Closure Empty "x" (Var {name = "x"}))))
```

Código 5.111: Estado de la máquina al haber realizado tres pasos de la transición.

En este punto es posible utilizar la regla 2) pues hay nuevamente una abstracción en la cadena de control:

$$\begin{aligned} &< \lambda z.z,\ [\,]\ ,((\textbf{(stop) fun}\ < [\,], x, x >) > \\ \underset{CEK}{\mapsto} &< \updownarrow\ ,\ [\,]\ ,(((\textbf{(stop) fun}\ < [\,], x, x >)\ \textbf{ret} < [\,], z, z >) > \end{aligned}$$

La continuación actual se extiende con la cerradura que representa a la abstracción funcional en la cadena de control. En Haskell:

```
*CEK> run_n_steps example_run 4
(UPDOWN,
Empty,
RC (RET (FUN STOP (Closure Empty "x" (Var {name = "x"}))) (Closure Empty "
    z" (Var {name = "z"}))))
```

Código 5.112: Estado de la máquina al haber realizado cuatro pasos de la transición.

La siguiente regla a aplicar es la 5) ya que hay una *ret*-continuación que contiene una continuación de función:

$$\begin{aligned} &< \updownarrow\ ,\ [\,]\ ,(((\textbf{(stop) fun}\ < [\,], x, x >)\ \textbf{ret} < [\,], z, z >) > \\ \underset{CEK}{\mapsto} &< x\ ,[(x, < [\,], z, z >)]\ ,\textbf{(stop)}\ > \end{aligned}$$

Al tener ambos componentes de la aplicación reducidos a un valor se procede a evaluar el cuerpo de la cerradura y se extiende el ambiente con el valor en la *ret*-continuación, hay que notar que la continuación resultante es la que existía antes de la aplicación de función. En Haskell:

```
*CEK> run_n_steps example_run 5
(T (Var {name = "x"}),
Env "x" (Closure Empty "z" (Var {name = "z"})) Empty,
PC STOP)
```

Código 5.113: Estado de la máquina al haber realizado cinco pasos de la transición.

Al tener únicamente una variable en la cadena de control se usa la regla 1) para buscar su valor en el ambiente:

$$< x\ ,[(x, < [\,], z, z >)]\ ,\textbf{(stop)}\ > \underset{CEK}{\mapsto} < \updownarrow\ ,\ [\,]\ ,(\textbf{(stop) ret} < [\,], z, z >>$$

Utilizando la función *lookup* se busca el valor de la variable en el ambiente y se posiciona en una *ret*-continuación. En Haskell:

```
*CEK> run_n_steps example_run 6
(UPDOWN,
Empty,
RC (RET STOP (Closure Empty "z" (Var {name = "z"}))))
```

Código 5.114: Estado de la máquina al haber realizado seis pasos de la transición.

Notemos que se ha llegado a un estado final de la forma $< \updownarrow, [\,], ((\textbf{stop})\ \textbf{ret}\ V) >$, más específicamente al estado $< \updownarrow, [\,], ((\textbf{stop})\ \textbf{ret} < [\,], z, z >>$ que corresponde al λ-término $\lambda z.z$ que es la función identidad, el resultado esperado.

En conjunto todo el proceso de reducción de la expresión implementado en Haskell:

```
*CEK> run_n_steps example_run 0
(T (App {fun = Lam {param = "x", body = Var {name = "x"}}, arg = Lam {
    param = "z", body = Var {name = "z"}}}),
Empty,
PC STOP)

*CEK> run_n_steps example_run 1
(T (Lam {param = "x", body = Var {name = "x"}}),
Empty,
PC (ARG STOP (Lam {param = "z", body = Var {name = "z"}}) Empty))

*CEK> run_n_steps example_run 2
(UPDOWN,
Empty,
RC (RET (ARG STOP (Lam {param = "z", body = Var {name = "z"}}) Empty) (
    Closure Empty "x" (Var {name = "x"}))))

*CEK> run_n_steps example_run 3
(T (Lam {param = "z", body = Var {name = "z"}}),
Empty,
PC (FUN STOP (Closure Empty "x" (Var {name = "x"}))))

*CEK> run_n_steps example_run 4
(UPDOWN,
Empty,
RC (RET (FUN STOP (Closure Empty "x" (Var {name = "x"}))) (Closure Empty "
    z" (Var {name = "z"}))))

*CEK> run_n_steps example_run 5
(T (Var {name = "x"}),
```

```
28 Env "x" (Closure Empty "z" (Var {name = "z"})) Empty,
29 PC STOP)
30
31 *CEK> run_n_steps example_run 6
32 (UPDOWN,
33 Empty,
34 RC (RET STOP (Closure Empty "z" (Var {name = "z"}))))
35
36 *CEK> run_n_steps example_run 7
37 (UPDOWN,
38 Empty,
39 RC (RET STOP (Closure Empty "z" (Var {name = "z"}))))
```

Código 5.115: Proceso de reducción de la expresión $(\lambda x.x)\ (\lambda z.z)$.

El siguiente ejemplo muestra una ejecución mucho más elaborada, esto con el objetivo de ilustrar el comportamiento que tiene nuestra implementación al reducir una $\mathcal{C}$-aplicación. Considere esta reducción que utiliza $\underset{c}{\rightarrow}$ la función de reducción para λ_c-términos:

$$
\begin{aligned}
&\mathcal{C}(\lambda k.(\lambda x.\lambda y.y)(k(\lambda z.z))(\lambda w.ww)) && \text{usando } (\rhd_{\mathcal{C}})\\
&(\lambda k.(\lambda x.\lambda y.y)(k(\lambda z.z))(\lambda w.ww))\ (\lambda x.\mathcal{A}x) && \text{usando } (\beta)\\
&(\lambda x.\lambda y.y)((\lambda x.\mathcal{A}x)(\lambda z.z))(\lambda w.ww) && \text{usando } (\beta)\\
&(\lambda x.\lambda y.y)(\mathcal{A}(\lambda z.z))(\lambda w.ww) && \text{usando } (\mathcal{A}_R)\\
&(\mathcal{A}(\lambda z.z))(\lambda w.ww) && \text{usando } (\mathcal{A}_L)\\
&\mathcal{A}(\lambda z.z) && \text{usando } (\rhd_{\mathcal{A}})\\
&(\lambda z.z)
\end{aligned}
$$

Observemos que la expresión le aplica la función $(\lambda x.\lambda y.y)$ a dos argumentos, dicha expresión representa a una función que toma dos argumentos y regresa el segundo ignorando completamente el primero, no obstante, hay una $\mathcal{C}$-aplicación que captura la continuación vacía y la aplica al primer argumento. Las continuaciones vacías, es decir las que se originan de una $\mathcal{C}$-aplicación como *redex* externo, capturan la continuación cuando ya no hay más por hacer generando así una *continuación de escape*. En este caso la continuación de escape se aplica sobre el primer argumento de $(\lambda x.\lambda y.y)$ y hace que se olvide todo el contexto al momento de utilizarse y regrese solo lo que su argumento evalúa, es por eso que en lugar de ignorarse el primer argumento es de hecho el resultado final.

Para la reducción paso a paso de este ejemplo primero implementamos la expresión $\mathcal{C}(\lambda k.(\lambda x.\lambda y.y)(k(\lambda z.z))(\lambda w.ww))$ usando Haskell como:

```
arg1 = (Lam "z" (Var "z"))
arg2 = (Lam "w" (App (Var "w") (Var "w")))
false = (Lam "x" (Lam "y" (Var "y")))
example2 = CApp (Lam "k" (App (App false (App (Var "k") arg1 )) arg2 ))
```

Código 5.116: Implementación de la expresión $\mathcal{C}(\lambda k.(\lambda x.\lambda y.y)(k(\lambda z.z))(\lambda w.ww))$ usando el tipo *Term*

La reducción paso a paso de la expresión $\mathcal{C}(\lambda k.(\lambda x.\lambda y.y)(k(\lambda z.z))(\lambda w.ww))$ se muestra de manera análoga al ejemplo anterior, cada paso de la ejecución muestra los componentes de la máquina y como van cambiando. En Haskell:

```
*CEK> run_n_steps example2_run 0
(T (CApp {arg = Lam {param = "k", body = App {fun = App {fun = Lam {param
   = "x", body = Lam {param = "y", body = Var {name = "y"}}}, arg = App {
   fun = Var {name = "k"}, arg = Lam {param = "z", body = Var {name = "z"
   }}}}, arg = Lam {param = "w", body = App {fun = Var {name = "w"}, arg =
    Var {name = "w"}}}}}}),
Empty,
PC STOP)

*CEK> run_n_steps example2_run 1
(T (Lam {param = "k", body = App {fun = App {fun = Lam {param = "x", body
   = Lam {param = "y", body = Var {name = "y"}}}, arg = App {fun = Var {
   name = "k"}, arg = Lam {param = "z", body = Var {name = "z"}}}}, arg =
   Lam {param = "w", body = App {fun = Var {name = "w"}, arg = Var {name =
    "w"}}}}}),
Empty,
PC (CONT STOP))

*CEK> run_n_steps example2_run 2
(UPDOWN,
Empty,
RC (RET (CONT STOP) (Closure Empty "k" (App {fun = App {fun = Lam {param =
    "x", body = Lam {param = "y", body = Var {name = "y"}}}, arg = App {
   fun = Var {name = "k"}, arg = Lam {param = "z", body = Var {name = "z"
   }}}}, arg = Lam {param = "w", body = App {fun = Var {name = "w"}, arg =
    Var {name = "w"}}}}))))

*CEK> run_n_steps example2_run 3
(T (App {fun = App {fun = Lam {param = "x", body = Lam {param = "y", body
   = Var {name = "y"}}}, arg = App {fun = Var {name = "k"}, arg = Lam {
   param = "z", body = Var {name = "z"}}}}, arg = Lam {param = "w", body =
    App {fun = Var {name = "w"}, arg = Var {name = "w"}}}}),
Env "k" (CPoint (PC STOP)) Empty,
```

```
PC STOP)

*CEK> run_n_steps example2_run 4
(T (App {fun = Lam {param = "x", body = Lam {param = "y", body = Var {name
    = "y"}}}, arg = App {fun = Var {name = "k"}, arg = Lam {param = "z",
   body = Var {name = "z"}}}}),
Env "k" (CPoint (PC STOP)) Empty,
PC (ARG STOP (Lam {param = "w", body = App {fun = Var {name = "w"}, arg =
   Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty)))

*CEK> run_n_steps example2_run 5
(T (Lam {param = "x", body = Lam {param = "y", body = Var {name = "y"}}}),
Env "k" (CPoint (PC STOP)) Empty,
PC (ARG (ARG STOP (Lam {param = "w", body = App {fun = Var {name = "w"},
   arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty)) (App {fun
    = Var {name = "k"}, arg = Lam {param = "z", body = Var {name = "z"}}})
    (Env "k" (CPoint (PC STOP)) Empty)))

*CEK> run_n_steps example2_run 6
(UPDOWN,
Empty,
RC (RET (ARG (ARG STOP (Lam {param = "w", body = App {fun = Var {name = "w
   "}, arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty)) (App
   {fun = Var {name = "k"}, arg = Lam {param = "z", body = Var {name = "z"
   }}}) (Env "k" (CPoint (PC STOP)) Empty)) (Closure (Env "k" (CPoint (PC
   STOP)) Empty) "x" (Lam {param = "y", body = Var {name = "y"}}))))

*CEK> run_n_steps example2_run 7
(T (App {fun = Var {name = "k"}, arg = Lam {param = "z", body = Var {name
   = "z"}}}),
Env "k" (CPoint (PC STOP)) Empty,
PC (FUN (ARG STOP (Lam {param = "w", body = App {fun = Var {name = "w"},
   arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty)) (Closure
   (Env "k" (CPoint (PC STOP)) Empty) "x" (Lam {param = "y", body = Var {
   name = "y"}}))))

*CEK> run_n_steps example2_run 8
(T (Var {name = "k"}),
Env "k" (CPoint (PC STOP)) Empty,
PC (ARG (FUN (ARG STOP (Lam {param = "w", body = App {fun = Var {name = "w
   "}, arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty)) (
   Closure (Env "k" (CPoint (PC STOP)) Empty) "x" (Lam {param = "y", body
   = Var {name = "y"}}))) (Lam {param = "z", body = Var {name = "z"}}) (
```

```
   Env "k" (CPoint (PC STOP)) Empty)))
45
46 *CEK> run_n_steps example2_run 9
47 (UPDOWN,
48 Empty,
49 RC (RET (ARG (FUN (ARG STOP (Lam {param = "w", body = App {fun = Var {name
       = "w"}, arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty))
   (Closure (Env "k" (CPoint (PC STOP)) Empty) "x" (Lam {param = "y", body
       = Var {name = "y"}}))) (Lam {param = "z", body = Var {name = "z"}}) (
   Env "k" (CPoint (PC STOP)) Empty)) (CPoint (PC STOP))))
50
51 *CEK> run_n_steps example2_run 10
52 (T (Lam {param = "z", body = Var {name = "z"}}),
53 Env "k" (CPoint (PC STOP)) Empty,
54 PC (FUN (FUN (ARG STOP (Lam {param = "w", body = App {fun = Var {name = "w
   "}, arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty)) (
   Closure (Env "k" (CPoint (PC STOP)) Empty) "x" (Lam {param = "y", body
   = Var {name = "y"}}))) (CPoint (PC STOP))))
55
56 *CEK> run_n_steps example2_run 11
57 (UPDOWN,
58 Empty,
59 RC (RET (FUN (FUN (ARG STOP (Lam {param = "w", body = App {fun = Var {name
       = "w"}, arg = Var {name = "w"}}}) (Env "k" (CPoint (PC STOP)) Empty))
   (Closure (Env "k" (CPoint (PC STOP)) Empty) "x" (Lam {param = "y", body
       = Var {name = "y"}}))) (CPoint (PC STOP))) (Closure (Env "k" (CPoint (
   PC STOP)) Empty) "z" (Var {name = "z"}))))
60
61 *CEK> run_n_steps example2_run 12
62 (UPDOWN,
63 Empty,
64 RC (RET STOP (Closure (Env "k" (CPoint (PC STOP)) Empty) "z" (Var {name =
   "z"}))))
```

Después de 12 pasos se ha llegado a un estado final de la forma $< \updownarrow, [\,], ((\textbf{stop})\ \textbf{ret}\ V) >$, siendo V el valor $< [(k, < \textbf{P}, \textbf{(stop)} >)], z, z >$. Es importante notar que k es la única variable en el ambiente de esta cerradura y no aparece en el cuerpo, por lo cual puede ignorarse, debido a esto la cerradura anterior corresponde al λ-término $\lambda z.z$ que es la función identidad, el resultado esperado.

En esta sección se describió como el cálculo lambda simple resulta limitado si se quiere emular el comportamiento de los operadores de control en un paradigma imperativo, debido

a esto se presentó el concepto de *continuación* como una forma de otorgar más control sobre la evaluación haciendo explícito y manejable el estado de control de un programa en un punto de su ejecución para que así se pueda tener acceso a él en lugar de permanecer oculto en el ambiente de ejecución. Además se describió una extensión del cálculo lambda que incorpora una funcionalidad capaz de manipular continuaciones, el Cálculo-λ_c.
El Cálculo-λ_c añade dos primitivas más al cálculo lambda simple que permitirán hacer uso explícito de las continuaciones en un término de la expresión, estás primitivas son las $\mathcal{C}$-aplicaciones y las $\mathcal{A}$-aplicaciones. Se implementó el tipo `Term` como una representación de los λ-términos. De la misma manera se definió el tipo `Environment` como una implementación de los ambientes a utilizar. Se introdujo el tipo `CC` como la representación de los *códigos de continuación* que dictan aquello que la máquina tendrá que hacer posteriormente a evaluar el término actual, conformados de dos partes las p-continuaciones y las ret-continuaciones.

En cuanto a los valores que la máquina CEK utiliza se definió el concepto de *puntos de continuación* como una forma de etiquetar un código de continuación, esto para posteriormente definir el tipo `Value` que se conforma de cerraduras y puntos de continuación representando así los valores de la máquina.

Usando los tipos anteriormente descritos se implementaron nuevos tipos para cada uno de los componentes de la máquina CEK. Se explicaron una a una las implementaciones y definiciones de las reglas que componen la función de transición `run`. Para una mayor comprensión del proceso de reducción se explicó como es que la función `run` va cambiando el estado de la máquina conforme va avanzando en cada uno de los pasos hasta llegar a un estado final. La exposición de todos estos aspectos de la implementación son de gran ayuda para que el lector pueda comprender el funcionamiento de la máquina viendo de forma transparente cómo se definen y transforman sus estados con el objetivo de reducir un término a un resultado.

Capítulo 6

Conclusiones y trabajo futuro

Hemos explorado la construcción del cálculo lambda y el rol tan importante que cumple al proporcionar un formalismo para expresar funciones como reglas de correspondencia. Además pudimos ver cómo es que cada máquina abstracta se construye utilizando el cálculo lambda como base teórica para sus definiciones, semántica operacional y reglas de transición.

Ahora podemos construir una conexión más fuerte y clara entre las máquinas de reducción y el funcionamiento interno de los lenguajes de programación. También es posible establecer cómo es que el estudio de cada máquina puede contribuir al entendimiento general de los conceptos clave que forman la teoría de los lenguajes, conceptos como *valores, constantes y operadores primitivos, alcance, ambientes, cerraduras, continuaciones*, entre otros.

Este trabajo nació como un intento de comprender el trasfondo teórico de los lenguajes de programación funcionales con un interés particular en el proceso de reducción de una expresión. Cada máquina aporta de manera diferente a varios conceptos teóricos (como los anteriormente mencionados) y resuelve los problemas que su *implementación* representa usando un enfoque distinto para atacarlos, enfoques como el uso de continuaciones, el uso de ambientes o el uso de la notación De Bruijn. Es importante notar que las máquinas abstractas surgieron como un intento de formalizar la implementación de un algoritmo que paso a paso realiza la *reducción* de una expresión puramente funcional, dotando así al cálculo lambda con un nivel de abstracción intermedio entre la teoría pura y la implementación de bajo nivel de una máquina real (hardware).

Primero veamos algunos conceptos que todas las máquinas tocan y como es que resaltan su importancia y papel dentro del proceso de reducción. Cada máquina utiliza fuertemente el concepto de *valores* como expresiones con un significado que sea atómico y en correspondencia con algún objeto matemático que consideremos significativo, una expresión a la que

se desea llegar después de un proceso de reducción. Se denota su gran importancia ya que todas las funciones de transición son consideradas exitosas si y sólo si se llega a un valor y no se detiene la ejecución. Los valores se encuentran asociados a las *cerraduras* ya que son la forma en que las funciones son vistas como valores y que todas las máquinas aceptan como un resultado final exitoso.

A pesar de que cada máquina lleva un manejo diferente del *ambiente* todas ellas resaltan su importancia pues siempre existe un conjunto de reglas que se ocupa de llevar un control de éste, sin mencionar que las cerraduras siempre llevan consigo un ambiente.

Un caso parecido es el concepto de las *variables libres y ligadas* que se presentan al momento de buscar un identificador en un ambiente (usando la función *lookup* en el caso de SECD y CEK) y excluyendo la existencia de variables libres en el resultado final ya que de encontrarse una en la expresión siempre terminará en un error.

Hablando de *orden de reducción*, cuando se miran detenidamente las reglas de inferencia que conforman la semántica operacional y las reglas de transición que rigen el proceso de reducción podemos notar que existe en cada máquina un orden específico para reducir las expresiones, esto implica que el lector puede explorar los efectos que puede causar la elección de un cierto orden por sobre otro en la complejidad de una reducción.

Hablando más puntualmete de los aspectos que individualmente cada máquina resalta en su construcción está el caso de las *constantes y operadores primitivos* en la máquina SECD, a pesar de que el conjunto de constantes es tan simple como los números enteros y el único operador es la función unaria de sucesión esta máquina ejemplifica como podemos incluir elementos que están fuera del cálculo lambda simple y operar con ellos mediante el uso de una función δ que evalúa los operadores primitivos.
La máquina de Krivine al utilizar la *reducción de orden normal* introdujo en este trabajo la noción de *orden de reducción* resaltando las diferencias que existen y deben ser tomadas en cuenta para cada máquina, si se compara el orden normal de la máquina de Krivine con el orden aplicativo que se usa en SECD podemos explorar las ventajas y desventajas que cada orden presenta. Más importante aún es el rol que la máquina de Krivine tuvo al enriquecer el estudio del problema de *captura de variable*. Si bien este problema está presente en el cálculo lambda en general y cada máquina lidia con él de formas distintas esta actividad de renombrar variables involucra un gran esfuerzo tanto en la codificación misma (implementación) como en el tiempo de ejecución. Sin embargo la máquina de Krivine mostró que es posible

reducir el impacto que la captura de variables causa al apoyarse de la *notación De Bruijn*. A pesar de que esta notación puede resultar díficil de leer para un humano resulta eficiente en la implementación pues elimina la necesidad de renombrar variables del todo. Así pues en la máquina de Krivine se expresa que podemos reducir la complejidad de los problemas de implementación que el cálculo lambda puede contener, por ejemplo la captura de variables, usando una solución teórica, en este caso la notación De Bruijn.

La máquina CEK surgió como una solución al inconveniente que el cálculo lambda simple tiene al no poder modelar uno de los aspectos más poderosos que tienen los lenguajes imperativos, los *operadores de control*. La solución propuesta a este problema fue la noción de *continuación* que, si bien no es simple de comprender y operar, aporta al desarrollo del cálculo lambda y le otorga mucho más poder computacional al permitir construir los operadores de control bien conocidos en el paradigma imperativo, operadores como *catch*, *throw*, *backtracking*, *corrutinas*, entre otras [24].
Cabe resaltar que aunque somos conscientes de que existen versiones más simples y actualizadas de la máquina CEK que facilitan su comprensión eliminando elementos de la misma, decidimos implementar la primera versión de la máquina aunque fuera más compleja. El motivo es que este trabajo está enfocado en explorar lo más posible el proceso de reducción de una expresión lambda por lo cual es importante entender varios conceptos clave de los lenguajes de programación, conceptos como cerraduras, ambientes y continuaciones. Todos estos elementos se encuentran en la versión de la máquina presentada aquí y a pesar de que pudieran agregar un considerable nivel de dificultad a su comprensión es justo esta dificultad lo que hace que su potencial didáctico sea mayor. La versión que decidimos implementar es más detallada y minuciosa en su descomposición de la evaluación de las expresiones lo que puede ser útil para comprender la semántica de los lenguajes de programación funcional y cómo manejan el flujo de evaluación de las expresiones.

Hemos implementado las máquinas abstractas y éstas enriquecen la comprensión de algunos de los conceptos más fundamentales que comprenden el desarrollo de los lenguajes de programación al sacarlos de su abstracción pura, donde permanecen ocultos u obscuros, y hacer que el que los estudia los piense, trabaje y manipule de forma más explícita. Conceptos como *cerradura*, *ambiente*, *sustitución* o *continuaciones* toman una forma definida dentro de la implementación de modo que es posible reconocer su funcionamiento y componentes. Por ejemplo cuando se realiza una reducción en el cálculo lambda puro no es muy claro que una función tiene un contexto de evaluación, sin embargo dado que un componente de las cerraduras es precisamente el ambiente este hecho queda explícitamente señalado, más aún

al mirar como es que alguna máquina cambia el ambiente al avanzar paso a paso el lector puede ir construyendo una idea más concreta del contexto de evaluación, cómo se usa y para qué sirve.
El autor espera que al unificar las diversas máquinas y los conceptos que ellas cargan, estos resulten más accesibles al lector inexperto y despierte el interés para poder continuar con la investigación en este campo.

Hay que notar que las máquinas abstractas pueden explorar importantes ambitos de los lenguajes de programación muy diversos como es el caso de las desarrolladas por John Hannan y Dale Miller [11], que exploran la posibilidad de cambiar las reglas de inferencia de la semántica operacional para modificar el comportamiento de la máquina e incluir funcionalidades como tipos, recursión, condicionales, pares, entre otras.

Es importante mencionar que aunque aquí nos enfocamos en máquinas abstractas desarrolladas únicamente en el paradigma funcional también existen máquinas abstractas fuera de éste. Para más información en máquinas abstractas creadas para lenguajes imperativos, lenguajes orientados a objetos, procesamiento de cadenas, lenguajes lógicos, lenguajes híbridos, programación en paralelo entre otros ver el trabajo de Stephan Diehla, Pieter Hartel y Peter Sestoft [3].

Han quedado fuera del alcance de este trabajo la implementación de *reglas delta* en las máquinas de Krivine y CEK debido a que han sido incluidas en la máquina SECD y consideramos que esto es suficiente para denotar su importancia. Asimismo es deseable la inclusión de algunas otras máquinas como la CAM [9], CLS [11], CC [26], CK [26] y SK [10] que son variaciones de las máquinas aquí presentadas o máquinas completamente nuevas, algunas máquinas deben su nombre a un acrónimo que corresponde a un nombre más largo como en el caso de CAM: Categorical Abstract Machine. Otras al nombre de sus componentes como el caso de CLS: Control, List y Stack ó SECD: Stack, Environment, Control y Dump. Sin embargo en máquinas como la CC, CK y SK el origen del acrónimo no es tan claro. De igual forma aún es necesario diseñar e implementar una representación en Haskell de estas máquinas que sea más accesible para el programador e implementar un analizador sintáctico que pueda convertir expresiones lambda en sintaxis concreta, es decir, expresiones escritas usando el lenguaje de programación Haskell. Después de todo consideramos que el objetivo principal ha sido cumplido al mostrar la estrecha relación que las máquinas guardan con los lenguajes de programación y el potencial de aprendizaje que ellas otorgan a un estudiante poco experimentado.

Bibliografía

[1] Chris Hankin. (1994). *Lambda Calculi: A guide for computer scientists.* New York: Oxford University Press.

[2] Krivine, JL. *A call-by-name lambda-calculus machine.* Higher-Order Symb Comput 20, 199–207 (2007).

[3] Diehl, S., Hartel, P., and Sestoft, P. (2000). *Abstract machines for programming language implementation.* Future Generation Computer Systems, 16(7), 739-751.

[4] E. Börger, D. Rosenzweig, The WAM – Definition and Compiler Correctness, in: C. Beierle, L. Plümer (Eds.), *Logic Programming: Formal Methods and Practical Applications*, North-Holland, Amsterdam, 1995, pp. 22–90.

[5] D.M. Russinoff, *A verified Prolog compiler for the Warren abstract machine*, J. Logic Programming 13 (1992) 367–412.

[6] Van Horn, D., and Might, M. (2010, September). *Abstracting abstract machines.* In Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (pp. 51-62).

[7] Abelson, Harold; Sussman, Gerald Jay (1996). *Structure and Interpretation of Computer Programs.* MIT Press.

[8] Danvy, O. (2005). *A rational deconstruction of Landin's SECD machine.* In Implementation and Application of Functional Languages: 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004 Revised Selected Papers 16 (pp. 52-71). Springer Berlin Heidelberg.

[9] G. Cousineau, P.-L. Curien and M. Mauny. *The Categorical Abstract Machine.* Science of Computer Programming, 8 (1987), pp. 173-202

[10] D.A. Turner, *A new implementation technique for applicative languages*, Software Practice and Experience 9 (1979) 31-49.

[11] John Hannan and Dale Miller. *From operational semantics to abstract machines*. Mathematical Structures in Computer Science, 2(4):415–459, 1992

[12] *Partial Evaluation and Semantics-Based Program Manipulation*, volume 26. New York: ACM, September 1991.

[13] Hannan, J. *Staging transformations for abstract machines*. In Proceedings of the 1991 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, volume 26. New York: ACM, September 1991. (pp. 130-141).

[14] Simon Marlow et al. (2010). Haskell 2010 language report. *Available Online http://www.haskell.org* (May 2011).

[15] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. no starch press, 2011

[16] Bryan O'Sullivan, John Goerzen y Donald Bruce Stewart. *Real world Haskell: Code you can believe in*. O'Reilly Media, Inc. 2008

[17] Haskell. (2013, September 9). HaskellWiki, . Retrieved 01:46, April 23, 2021 from *https://wiki.haskell.org/index.php?title=Haskell&oldid=56799.*

[18] Murillo Albarrán, Diego. *Una implementación polimórfica de ISWIM*. Tesis para obtener el grado de licenciatura, Universidad Nacional Autónoma de México, Facultad de Ciencias. 2017

[19] J. B. Rosser S. C. Kleene. *The inconsistency of certain formal logics*. Annals of Mathematics, 36(3):630 - 636, 1934.

[20] Alonzo Church. *A set of postulates for the foundation of logic*. In *Annals of Mathematics*, 33(2):346-366, 1932

[21] A. Church, *The Calculi of Lambda Conversion*. Princeton, NJ, USA: Princeton Univ. Press, 1941.

[22] Krishnamurthi, S. (2012). *Programming languages: Application and interpretation*. Brown University.

[23] De Bruijn, N. G. (1972, January). *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem*. In Indagationes Mathematicae (Proceedings) (Vol. 75, No. 5, pp. 381-392). North-Holland.

[24] Fellesein, M., D.P. Friedman, E. Kohlbecker, B. Duba. *Reasoning With Continuations*, Proc. Symp. Logic in Computer Science, 1986, 131-141.

[25] Landin, P.J. *A correspondence between ALGOL 60 and Church's lambda notation* , Comm. ACM, 8(2), 1965. 89-101;158-165

[26] Felleisen, M., and Friedman, D.P. (1987). *Control operators, the SECD-machine, and the λ-calculus*. Formal Description of Programming Concepts.

[27] Haynes, C. T., Friedman, D. P., and Wand, M. (1984, August). *Continuations and coroutines*. In Proceedings of the 1984 ACM Symposium on LISP and functional programming (pp. 293-298).

[28] Sabry, A., and Felleisen, M. (1993). *Reasoning about programs in continuation-passing style*. Lisp and symbolic computation, 6, 289-360.

[29] Talcott, C., *The Essence of Rum–A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*, Ph.D. dissertation, Standford University, 1985.

[30] Landin, P.J., *The mechanical evaluation of expressions*, Computer Journal 6 (4), 1964.

[31] Clinger, W.D., et. al., The revised report on Scheme, *Joint Technical Report Indiana University 174 and MIT Laboratory for Computer Science 848, 1985*

[32] Carmen Gómez Laveaga, *Álgebra Superior: curso completo*. Primera edición. Editorial: Prensas de Ciencias, 2014.

[33] Henderson, Peter. *Functional Programming: Application and Implementation.* Prentice Hall, 1980.

[34] Padget, Julian. *Three Uncommon Lisps.* First International Workshop on Lisp Evolution and Standardization, 1988.

[35] Graham, Brian. *SECD: DESIGN ISSUES.* University of Calgary, 1989.

[36] Alonzo Church. A proof of freedom from contradiction. Proceedings of the National Academy of Sciences of the United States of America, 21(5), 275-281, 1935.

[37] Church, A., and Rosser, J. B. (1936). *Some properties of conversion.* Transactions of the American Mathematical Society, 39(3), 472-482.

[38] Sussman G.J., G. Steele. *Scheme: An interpreter for extended lambda calculus.* Memo 349, MIT AI-Lab, 1975.