UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Verificación formal del cálculo λSF en Coq

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Mateo Alberto Torres Ruiz

DIRECTOR DE TESIS

Dr. Favio Ezequiel Miranda Perea

Ciudad Universitaria, Cd. Mx., 2019

Agradecimientos

Quiero agradecer en primer lugar a mi madre, por todo su apoyo a lo largo de los años, así como a mis dos hermanos, Julián y Sofía. A mi familia en general, por el buen tiempo compartido conmigo y la educación que inculcaron en mí.

Me siento también obligado a agradecer a gran cantidad de amigas y amigos, para mi suerte imposibles de enumerar aquí, por todos los días compartidos.

A Clarisa por su apoyo incondicional durante los últimos años.

A Favio por mostrarme, desde los cursos más básicos que da en la Facultad de Ciencias hasta las conferencias que imparte, lo bella e interesante que llega a ser la computación teórica, los lenguajes de programación y la verificación formal.

A Iván por guiarme en los últimos meses y ayudarme tanto.

A Barry Jay y su equipo, tanto por la pronta respuesta a dudas e inquietudes como por todo el material desarrollado.

Índice general

Capítulo 1

Introducción

La enorme capacidad del ser humano para conocer y modificar su ambiente se encuentra fuertemente ligada a su facilidad de abstracción. Toda teoría, por más elaborada que sea, toma en sus conceptos más básicos, y como bloque inicial para la construcción de todos sus postulados, ideas que relacionan un término con otro; se revela así la capacidad de abstracción como pieza fundamental de todo razonamiento. Puede parecer que, como cada término que se define es a su vez definido por medio de otros, el conocimiento humano debería contentarse con aceptar ciertas ideas con una convicción similar a la de la fe para así lograr definir conceptos útiles. El mismo supuesto sobre la existencia de las formas de la intuición obliga la aceptación de cierta constitución mental (ininteligible para nosotros) necesaria para todo proceso de cognición [1], el cual estará hecho sobre las bases de ciertas abstracciones que relacionan conceptos cada vez más complejos con aquellos primigenios. En este sentido, la programación de una computadora se devela como un proceso que muestra de forma clara este mecanismo del razonamiento: por un lado, el acto de programar necesariamente está ligado con el de diseñar. Sería inútil sentarse a programar sin tener de antemano algún problema que resolver. Por otro lado, al programar se necesita tener en cuenta las distintas abstracciones disponibles o que uno podría requerir para resolver el problema para el cual se diseñó una solución de antemano. Las abstracciones que uno encuentra en la computación son particularmente numerosas. La misma computadora es un sistema complejo hecho de partes mecánicas, uno o varios sistemas operativos, herramientas de software, middleware, etc. Todos estos componentes son resultado del trabajo de miles de personas a lo largo de muchas décadas y están interconectadas de forma compleja, trabajando de manera sumamente organizada para resultar en las máquinas funcionales que conocemos actualmente [2].

La naturaleza misma de la computación hace posible estudiarla desde

muchas y muy diversas trincheras. Uno de los acercamientos que uno podría dar a su estudio es a través de los lenguajes de programación. Dichos lenguajes forman un conjunto considerablemente grande hoy en día y, aunque cada uno de los miembros de dicho conjunto funciona como paso intermedio entre la especificación de un sistema y el proceso que lo implementa (i.e., un programa), varían en forma sustancial en tanto que estos están basados en diversos modelos computacionales que definen formalmente cómo los cómputos son llevados a cabo [2]. La semántica y sintaxis de los lenguajes están estrechamente conectadas con los conceptos fundacionales sobre los cuales se idearon. Las diversas formas de ver a la programación, hoy llamadas "paradigmas", son el resultado de distintas escuelas de pensamiento que adoptaron una filosofía determinada respecto a cómo deberían funcionar estos lenguajes.

Dentro de una computadora son necesarios procesos de lo más diversos para llevar a cabo las tareas que cotidianamente realizamos con ellas. La robustez y debilidad de estos procesos depende, de forma llana y general, de dos cosas: el modelado, es decir, lo buena que pudiera ser la descripción de la tarea a realizar por la computadora, la facilidad o posibilidad de dividir la tarea en una serie de procesos más sencillos, la claridad de las estructuras de datos utilizadas, etc.; y, de mayor interés para este trabajo, del lenguaje que es utilizado para modelar lo anteriormente dicho.

Tal como las estructuras de datos, que nos permiten almacenar y organizar datos de forma que el acceso y modificación de estos sea sencillo, no pueden otorgarnos una estructura coherente para todo propósito, y por lo mismo se vuelve relevante el estudio sobre las fortalezas y limitaciones de una colección considerable de estas estructuras [3], es importante así conocer los modelos de computación que funcionan como base de los distintos lenguajes de programación, para, de manera similar, conocer qué paradigmas facilitan el uso de las estructuras requeridas para determinadas tareas, la facilidad de abstracción de los objetos requeridos, etc.

Surge de forma natural la pregunta sobre la necesidad de tener tantos estilos, a veces tan diversos entre ellos, de programación. Muchas veces dentro de un mismo hardware, tenemos ejecutándose programas que están hechos de forma relacional, funcional, imperativa y orientada a objetos. Más interesantes resultan los proyectos de gran magnitud: los programas prácticos que se generan en la industria alcanzan fácilmente el tamaño de unas miles de líneas de código y hasta millones a veces, donde distintos paradigmas existen en interacción y aportan de diversas abstracciones y sintaxis distintas. No es, claro, algo deliberado. Estos programas no se hacen buscando abarcar el mayor número de paradigmas disponibles. La razón es un tanto sencilla: uno no programa con los paradigmas existentes, sino con diversos conceptos, cada uno de los cuales es más fácil utilizar y comprender desde un punto

de vista en específico que brindan los paradigmas. Nace necesariamente la intención de mezclar varios conceptos en la solución de un problema y, así, la aparición de diversos paradigmas dentro de un mismo programa surge como algo accidental. Las formas de razonar sobre un problema que llevan a la implementación de un programa en particular son siempre específicas al programa en cuestión.

La interacción entre los estilos que tienen los distintos paradigmas no sólo se vuelve compleja conforme nuevas capacidades son añadidas a estos por parte de sus respectivas comunidades, sino que también se vuelve cada vez más costoso mantener los frágiles hilos que, mediante capas de abstracción, permiten la interacción entre las distintas partes de los programas.

Las preguntas que ahora corresponde hacernos son: ¿qué tan compatibles son entre sí los distintos conceptos centrales que se tienen en las diversas formas de programar?, ¿pueden coexistir de manera natural estos? Preguntas más abstractas sobre la naturaleza de las funciones que abarcan las teorías que sirven de pie a los distintos paradigmas fueron ya respondidas a principios del siglo XX, siendo los trabajos de Alonzo Church [4] y Alan Turing [5] los más destacados. Es ahora un resultado bien conocido el que ambos modelos computacionales, tanto el cálculo-λ desarrollado por Church (centrado en las funciones y el cual podríamos decir es base de la programación funcional) como las máquinas de Turing (centradas en estados y a las cuales podríamos ver como el fundamento de la programación imperativa) tienen la misma capacidad computacional: ambos son capaces de expresar la familia de las funciones recursivas. La equivalencia establecida entre ambas concepciones nos habla únicamente sobre la capacidad que tienen de realizar las mismas funciones sobre el conjunto de los números naturales, mas no sobre la similitud de estilos ni la capacidad de reúso de los programas que pudieran resultar de sus vistas particulares respecto a cómo computar. La distinción entre los lenguajes inspirados tanto por el modelo de Turing como por el modelo de Church no sólo se establece en la práctica de forma estilística, también tienen amplias diferencias teóricas. Un vistazo rápido a las comunidades dedicadas al estudio de estas distintas formas de cómputo basta para notar los tópicos que son de crucial interés para cada uno de los modelos, enfocándose el de Turing en la complejidad de los algoritmos y el de Church en su significado [6].

Las implicaciones prácticas del desarrollo de un estilo de programación que permita la interacción natural de los estilos de cada uno de los distintos paradigmas hasta ahora utilizados en la programación podrían ser realmente amplias. La unificación de los distintos paradigmas se ha intentado ya con anterioridad [7, 8], mas nunca ha prosperado ni ha logrado encontrar un nicho permanente en la práctica o tenido la oportunidad de cambiar de forma

sustancial el ecosistema de los lenguajes de programación.

Si llevamos a sus puntos extremos los distintos estilos que hoy se establecen sobre los paradigmas que han resultado útiles, veremos que por un lado tenemos lenguajes que facilitan el tratamiento de todas las cosas como funciones (programación funcional) y, por otro, lenguajes que consideran que toda abstracción está encima de una estructura, la cual se conforma de un estado que puede mutar por medio de asignaciones a los componentes que lo conforman (programación imperativa). En el medio, modificaciones y/o mezcla de estas dos concepciones dan espacio a diversos estilos (programación orientada a objetos, lenguajes de queries, etc.).

El presente trabajo explora un modelo propuesto en años recientes [9, 10] que logra caracterizar de manera sintáctica a los objetos dentro suyo como formas factorizables y de esta forma dar cuenta de su estructura interna, aumentando el espectro de expresividad en comparación con los modelos clásicos de la computación, en tanto que permite la introducción de nociones intensionales que, en conjunto con el soporte extensional que brindan ya los paradigmas comunes, otorga la capacidad tanto de expresar todas las funciones y estructuras de datos comúnmente halladas en los distintos lenguajes de programación, como de representar funciones intensionales como la igualdad estructural de los programas representados en este formalismo. Asimismo, permite un tratamiento uniforme de todos los objetos del lenguaje a través del emparejamiento de patrones, tal y como se formaliza en el cálculo de patrones y [6], a través de sus diversos mecanismos de reducción, en conjunto con los hallados en el cálculo λ y de la capacidad de identificar a programas con estructuras de datos manipulables de forma extensional e intensional, posibilita una provechosa línea de trabajo futuro en el ámbito del análisis y optimización de programas, pues facilita la remoción de dificultosos procesos implicados en ello, como son el uso de un metalenguaje para el etiquetado externo o aritmetización de las abstracciones, pudiendo dar cuenta de ellas desde dentro suyo.

1.1. Lenguajes imperativos

Los programas que son construidos en los lenguajes de programación "tradicionales" (como C o Fortran) se conforman de un conjunto de instrucciones que indican cómo deben hacerse los cambios sobre un estado inicial, el cual se constituye de distintos valores σ que cambian reiteradamente a lo largo de la ejecución del programa para finalizar con nuevos valores σ':

$$\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n = \sigma'$$

El modelo en el que se basan estos programas abstrae una noción de las funciones computables que las ve como un conjunto de cambios de estado. El estado inicial de estos programas es típicamente modificado a través de comandos de asignación. A los programas que siguen este modelo son a los que solemos llamar como escritos en un estilo "imperativo", mismo que es pilar del paradigma más popular en la actualidad, el orientado a objetos.

Ejemplo 1.1.1.

```
int i, j;
for (i = 0; i < n - 1; i++)
   for (j = 0; j < n - i - 1; j++)
      if (arr[j] > arr[j+1]) {
         int t = arr[j];
         arr[j] = arr[j+1];
         arr[j+1] = t;
      }
```

Para posteriormente poder contrastar programas que siguen el estilo imperativo con programas hechos en otros modelos (en particular, el funcional), aquí se encuentra el código que implementa el algoritmo de ordanamiento "*bubble sort*". En este se puede observar de forma clara el uso de asignaciones para lograr el ordenamiento de una colección de números.

Los primeros lenguajes de programación que se desarrollaron tenían un objetivo bastante claro: dotar de un vehículo mediante el cual se pudiera controlar el comportamiento de una computadora digital. De esta forma, los primeros lenguajes reflejaron de una manera bastante fiel la estructura de las computadoras en las cuales operaban. Así, la arquitectura que tienen las computadoras digitales ha tenido un fuerte impacto a lo largo de la historia del diseño de lenguajes de programación, siendo la mayoría de los lenguajes más populares de los últimos cincuenta años diseños encaminados a mejorar la eficiencia de los procesos que se realizan al nivel más bajo en las computadoras de arquitectura von Neumann. En esta arquitectura, datos y programas son almacenados en una misma memoria y existe una unidad de procesamiento central, también conocida como CPU, en una memoria independiente, que se encarga de la ejecución de instrucciones; los datos de memoria en conjunto con las instrucciones se transmiten a través de un búfer desde la memoria de almacenamiento a la CPU, donde los resultados, una vez obtenidos, corren de vuelta a la memoria. Esta forma de funcionamiento se ve materializada en las características que definen a los lenguajes imperativos: las celdas de memoria son representadas por las distintas variables que se puedan hallar en el programa; las sentencias de asignación representan la transmisión que corre por el búfer, y las instrucciones iterativas se erigen como la forma más básica

para repetir una instrucción, dejando de lado a la recursión, en tanto que las instrucciones en la arquitectura von Neumann son almacenadas en celdas de memoria adyacentes, lo que hace que esta forma de repetición requiera únicamente de un salto a una instrucción anterior [11, 12, 13].

Viéndolos como un conjunto ordenado en el tiempo, estos lenguajes pueden verse como una enorme progresión de desarrollos en torno a la mejora de un modelo básico de programación imperativa establecido en 1957 por el lenguaje Fortran 1 [14]; todos diseñados para hacer un uso eficiente de la arquitectura von Neumann, y fuertemente relacionados en el análisis de sus programas con las máquinas de Turing, en tanto que esta formalización, al estar estrechamente ligada al concepto de "estados", permite mimetizar de manera bastante fiel la forma en la que son realizados los cómputos en dicha arquitectura, logrando dar cuenta del número de operaciones requeridas para realizar distintas rutinas, lo que a su vez posibilita predecir la eficiencia tanto en tiempo como en memoria de ellas.

1.2. Lenguajes funcionales

Aunque el estilo imperativo logró asentarse en un nicho especial en la comunidad de programadores, depender de forma tan fuerte en la arquitectura subyacente ha sido visto por varios como una restricción innecesaria respecto a los acercamientos que pudiera tener el desarrollo de software.

No tardó mucho en cambiar la forma tradicional de ver el diseño de los lenguajes de programación, pues no es difícil darse cuenta de que aquellas abstracciones sobre las cuales una máquina logra operar de forma óptima no son necesariamente sobre las que resulta más fácil razonar a un ser humano. La programación funcional, nacida con Lisp en 1958 [15], representa una ruptura radical respecto del modelo que sustenta a los lenguajes imperativos.

En esencia, los lenguajes de programación funcionales logran codificar a sus programas como una única expresión, que es conceptualizada como una función, haciendo la ejecución de un programa la evaluación de esta. Aunque bien es posible asociar a programas hechos en los lenguajes imperativos comunes con una función f que determina el estado final a partir del incial $\sigma' = f(\sigma)$, las características principales de los lenguajes funcionales los distancian de los imperativos en tanto que prescinden de cualquier tipo de variables, sentencias de asignación y formas iterativas como sentencias de repetición primitivas, lo que permite escribir programas en un nivel más abstracto, similar a las funciones empleadas en las matemáticas, facilitando así el razonamiento formal y análisis en torno al significado de sus programas. Asimismo, la inexistencia de estados permite la ejecución en arquitecturas

paralelas de forma natural y la incapacidad de asignar valores a memoria evita cualquier efecto secundario.

Ejemplo 1.2.1.

```
bubbleSort :: Ord a => [a] -> [a]
bubbleSort = foldr swapbs []

swapbs x [] = [x]
swapbs x (y:xs) = min x y : swapbs (max x y) xs
```

Nuevamente mostramos el algoritmo de ordenamiento burbuja, esta vez en Haskell, un lenguaje funcional puro.

La idea de desarrollar lenguajes de programación que se separen de forma abrupta de la tradición imperativa ha sido defendida numerosas veces. En su lectura del premio Turing en 1978 [16], John Backus, quien trabajó en el desarrollo de Fortran, defendió el uso y desarrollo de los lenguajes puramente funcionales (vistos en la época como artificios académicos de poco valor en aplicaciones reales) en tanto que estos resultan más "legibles, confiables y probables a ser correctos".

El desarrollo que han seguido esta clase de lenguajes se ha visto influenciado por un vasto número de teorías, sin embargo, ninguna llegó a repercutir en estos de forma tan radical como lo hizo el trabajo de Alonzo Church en torno al cálculo λ, que fue el primer tratamiento de los aspectos funcionales de la computación; su naturaleza libre de tipos, así como el corto número de reglas, permitieron un cálculo simple y conciso con una propiedad sumamente interesante: poder expresar y razonar acerca de las funciones en el entorno más general posible. Es sobre este formalismo, en conjunto con la lógica combinatoria, equivalente a él, y desarrollos posteriores que giran alrededor de estos dos, que elaboramos en el presente trabajo una exposición de los fundamentos que abren camino para el diseño de nuevos lenguajes de programación capaces de mantener la fácilidad de análisis y optimización en torno a sus programas, tal como en los lenguajes imperativos, así como permitirnos examinar su significado semántico, como ocurre en los lenguajes funcionales, y dotar de una capacidad expresiva ausente en los paradigmas popularizados hoy día.

1.3. Plan de trabajo

En el segundo capítulo de este trabajo se revisará la teoría fundacional de Alonzo Church, el cálculo λ, misma que funge como fundamento de los lenguajes de programación basados en el paradigma funcional. Posteriormente,

en el capítulo tercero, se formalizarán las ideas de Moses Schönfinkel y Haskell Curry en torno a la lógica combinatoria, que serán ligadas al cálculo λ, y se mostrará la forma en la que es posible establecer un homomorfismo entre estos formalismos, permitiéndonos en todo momento hacer uso del lenguaje que mejor se acomode al escenario que encontremos, sabiendo que existen mecanismos por los cuales podemos pasar de los términos de uno a los del otro. En el capítulo cuarto se revisarán los conceptos establecidos por Barry Jay, referentes al cálculo $\boldsymbol{SF}$, se explicarán diversas definiciones y resultados respecto a este, como la contención de la lógica combinatoria dentro suyo; se expondrá la diferencia que existe entre las funciones extensionales e intensionales dentro de los modelos de la computación, y se definirá la igualdad intensional dentro del cálculo que exponemos. En el capítulo quinto expondremos el cálculo $\lambda\boldsymbol{SF}$, sus propiedades y diversos resultados, para cada uno de los cuales se ofrece su demostración. Se mostrarán también los conceptos más básicos del análisis y optimización de programas, trazando el escenario existente en esta área en torno a los dos formalismos que forman parte de este último cálculo: el cálculo λ y el cálculo $\boldsymbol{SF}$, con la finalidad de sentar el panorama sobre el que nos encontramos hoy en día para facilitar el trabajo futuro en torno a estos temas dentro del cálculo $\lambda\boldsymbol{SF}$. El último capítulo concluirá y establecerá diversas posibilidades de trabajo futuro relacionado con los temas expuestos en esta tesis, en partícular con los cálculos $\boldsymbol{SF}$ y $\lambda\boldsymbol{SF}$.

Para las definiciones y resultados relevantes expuestos en los capítulos segundo y tercero, se ofrecerá su respectiva formalización en el asistente de pruebas Coq, mientras que los resultados mostrados en el capítulo quinto, para los cuales existe formalización dentro de Coq, mas no demostraciones en el común sentido de la palabra, se ofrecerán, como ya mencionamos, estas; mismas que se extienden a las definiciones y resultados mencionados en el cuarto capítulo.

Capítulo 2

Cálculo λ

El uso de funciones dentro de las diversas áreas relacionadas con la matemática es actualmente tan extenso y visto con tanta normalidad que interesarnos por un análisis respecto a su naturaleza podría parecer carente de sentido a primera vista. La notación con la que contamos, tanto para la abstracción de funciones como para la substitución de variables dentro de ellas, se remonta por lo menos al año de 1889, cuando Peano establece en su *Arithmetices Principia Nova Methodo Exposita* la forma en la que se había de denotar las funciones de variable x que son determinadas por un término α, misma que se usa hoy día: $\alpha(x)$.

Las veces en que las mismas funciones son a su vez argumentos de otras ocurren en muchas y muy diversas ramas de la matemática con gran frecuencia y tienden a ser de una relevancia considerable. Si bien no siempre se explicita, e incluso podría costar algo de trabajo notar en un primer acercamiento que una función esté siendo utilizada como parámetro de otra, esta clase de funciones son presentadas desde los primeros pasos que se dan en cursos básicos de matemáticas. Como ejemplo común de funciones actuando en otras tenemos al operador diferencial del cálculo, el cual convierte una función que da como resultado valores numéricos en una función que da cuenta del ritmo de cambio que están teniendo estos valores.

Dentro del ámbito de la computación no es extraña la idea de las funciones como miembros de primera clase, es decir, las funciones son vistas como objetos capaces de ser pasados como parámetros a otras funciones, de ser evaluadas, y de ser regresadas como resultados. La facilidad o dificultad con lo que esto se hace estriba, en la mayoría de los casos, en las tensiones que existen entre los diversos paradigmas de la computación (tal como se discutió en el capítulo primero), siendo el funcional el que, precisamente, se erige sobre la concepción de las funciones como los objetos centrales del cómputo y permite sin el mayor esfuerzo el uso de estas para un amplísimo abanico

de usos.

Ejemplo 2.0.1.

```
int mcd(int a, int b) {
   if (b == 0)
      return a;
   return mcd(b, a%b);
}
```

La función para computar al mínimo común divisor de dos números enteros puede expresarse dentro de C++ como una función recursiva, donde se regresa a la misma hasta llegar al caso base de la recursión.

Ejemplo 2.0.2.

```
dosVeces :: (a -> a) -> a -> a
dosVeces f x = f (f x)
```

Dentro de Haskell, un lenguaje puramente funcional, es posible definir funciones que toman como argumento a otras. En este ejemplo, se reciben un valor y una función, la cual se aplicará dos veces a este.

Como veremos en este capítulo, la maquinaria funcional logra representar tanto a los números naturales como a cualquier función que actúa sobre estos y sea Turing computable, mostrando así que el punto de vista puramente funcional de la computación no es sólo compacto, sino que también es capaz de dar cuenta de todo cuanto sea computable.

La teoría de las funciones sobre la cual desarrollaremos es una teoría libre de tipos que versa sobre funciones como reglas, oponiéndose a la concepción más común que se tiene, atribuida por lo general a Dirichlet, de las funciones consideradas como gráficas, idea que es, por lo general, la imagen mental que se nos da de lo que es una función en los cursos más básicos de matemáticas: que las funciones pueden y deben ser vistas como conjuntos de parejas argumento-valor.

Apartándonos de esta concepción tan útil en diversas áreas de las matemáticas, nos enfocamos en el cálculo λ, que conceptualiza a las funciones ya no bajo el panorama enormemente popularizado de parejas que facilita su "visualización", sino como reglas, de forma que el estudio de los aspectos computacionales de estas sea facilitado.

El cálculo λ, concebido y estudiado alrededor de 1928 por Alonzo Church y publicado por vez primera en 1932, se construyó con dos objetivos en mente: el primero, desarrollar una teoría general de las funciones; el segundo, extender dicha teoría con nociones de la lógica, proveyendo así una fundación

para la matemática. Desafortunadamente, los intentos de proveer a través de este cálculo una fundación sólida a la matemática fallaron. El sistema planteado por Church era inconsistente, como mostraron Kleene y Rosser, alumnos suyos, en 1935 [17].

El punto de partida de la teoría desarrollada por Church inicia con las funciones como objetos de estudio de esta e incluye como sus miembros primitivos a la abstracción ($\lambda x.M$) y a la aplicación (MN). En particular, dentro del cálculo λ, las funciones pueden ser aplicadas a ellas mismas (así, son también los "argumentos" de toda función, funciones). Esto fue en su época un tratamiento bastante novedoso, pues la noción común de lo que entonces se consideraba que era una función, tal como se establece en la teoría de los conjuntos de Zermelo-Fraenkel (teoría que surge también de la necesidad de tener una maquinaria fundacional de la matemática), no lo permite, consecuencia del axioma de buena fundación de la misma teoría.

Definición 2.0.1. (Función dentro de ZFC). "Una función es una relación F con la propiedad de que para todo $x \in Dom(F)$ hay una única y tal que $< x, y > \in F$. Es usual llamar a tal única y, el valor de F en x y denotarla $y = F(x)$."[18]

Lema 2.0.1. *Dada la noción común de función (tal como aparece en ZFC) no es posible aplicar una función a sí misma.*

Demostración: Supongamos que existe una función F tal que se aplica a sí misma. Entonces existe el par $(F, x) \in F$. Dado que las funciones son conjuntos de pares ordenados, tenemos que $(F, x) = \{\{F\}, \{F, x\}\}$, y así: $F \in \{F\} \in (F, x) \in F$. De modo que tenemos un conjunto

$$\gamma = \{F, \{F\}, \{\{F\}, \{F, x\}\}\}$$

Por el axioma de buena fundación:

$$\forall x(x \neq \emptyset \rightarrow \exists y \in x(y \cap x = \emptyset))$$

Debería cumplirse que $\exists y \in \gamma(y \cap \gamma = \emptyset)$, pero no hay tal y en γ, lo cual contradice al axioma de buena fundación. Así, no es posible, dentro de ZFC, tener a una función que se aplique a sí misma. $\square$

Pese a su fracaso como fundación matemática, la parte del cálculo λ que se encarga de lidiar con las funciones resultó un proyecto sumamente exitoso: fue por medio de esta teoría que Church pudo por vez primera ofrecer una formalización de la noción de lo que es "efectivamente computable" a través del concepto de la λ-definibilidad. En 1936, Turing publicó un famoso artículo en el que estableció que sus máquinas y la noción de computabilidad resultante de ellas (lo hoy conocido como Turing-computable) es, de hecho, equivalente a la noción de computabilidad antes establecida por Church [19].

Definición 2.0.2. (Efectivamente computable)[20]. Diremos que una función f es *efectivamente calculable* o *efectivamente computable* si y sólo si existe un procedimiento efectivo que, dada una entrada x, logre producir, tal y como se espera, $f(x)$.

A partir del análisis que Turing realizó, se sigue que, a pesar de tener una sintaxis tan mínima, el cálculo λ puede describir a todas las funciones computables. Así, podría incluso ser visto como un lenguaje de programación. Consideremos una función

$$f(x) = 3 * x$$

tal que $f(1)$ resulta en 3, $f(2)$ en 6, etc. La declaración de esta función tanto la describe como una que multiplica por tres a cualquier $x \in \mathbb{N}$ que le pasemos, como la nombra (es la función f). Contrastemos esta forma de expresarla con la manera en la que la misma función es expresada dentro del cálculo λ, que trabaja con funciones anónimas a través de los conceptos de abstracción y aplicación, detallados más adelante en este mismo capítulo: nuestra función, asumiendo que la multiplicación fue previamente definida y su operador es infijo, se vería de la siguiente forma en nuestro lenguaje:

$$\lambda x.3 * x$$

Ahora nos es posible separar el nombre de la función de lo que esta hace:

$$f = \lambda x.3 * x$$

La aplicación de un valor numérico a nuestra función resulta en la evaluación por medio de reducciones de esta:

$$(\lambda x.3 * x)3$$

Pues reemplaza las ocurrencias de x por 3 en $3 * x$, resultando en $3 * 3$, lo que evalúa a 9.

2.1. Cálculo λ puro

El cálculo λ, lenguaje cuyas nociones básicas esbozaremos en este capítulo, es universal en el sentido en que cualquier función computable puede ser expresada y evaluada a través de este formalismo. *Grosso modo*, el cálculo λ consiste en una única regla de transformación y en un único esquema de definición de funciones. Las expresiones válidas del lenguaje son dadas por la sintaxis de términos que definimos a continuación:

Definición 2.1.1. (Términos lambda). Los términos lambda o λ-términos son expresiones definidas sobre un alfabeto compuesto por un número infinito de variables $x_0, x_1, \ldots$, un abstractor (λ) y signos impropios como paréntesis y corchetes $(,), [,]$. El conjunto de todos los λ-términos, Λ, representa a todas las expresiones válidas de nuestro lenguaje y se define de forma inductiva como:

1. $x \in \Lambda$,
2. $M \in \Lambda \Rightarrow (\lambda x.M) \in \Lambda$,
3. $M, N \in \Lambda \Rightarrow (MN) \in \Lambda$,

donde x es una variable arbitraria.

Algunas anotaciones pueden ser hechas sobre los λ-términos: En primer lugar, se puede notar, a partir del punto uno, que la expresión válida más sencilla del cálculo λ es aquella que consta únicamente de una variable. En segundo lugar, las expresiones similares al punto número dos (es decir, aquellas que son precedidas por el abstractor λ y que contienen a un λ-término después del punto) son conocidas como "abstracciones", y ligan a una variable (x en este caso) con un término válido del lenguaje (M). Como un primer ejemplo, la expresión que representa a la función identidad:

$$\lambda x.x$$

consta solamente de la abstracción de un término que es ligado a sí mismo. El λ-término que aparece inmediatamente después de la λ (en este caso, la variable x) representa a la variable ligada que funciona como argumento de la función, mientras que la expresión que sigue del punto (en este caso, nuevamente, la variable x) es lo que conocemos como "el cuerpo" de la expresión. Como tercera y última anotación sobre la defnición de los términos de nuestro lenguaje, en el punto número tres podemos ver la forma general de la *aplicación*, que aplica N al término M. La aplicación es asociativa por la izquierda. Es decir, dada una expresión

$$M_0 M_1 M_2 \ldots M_n$$

ella se evalúa aplicando las expresiones de la siguiente manera:

$$(\ldots((M_0 M_1)M_2)\ldots M_{n-1})M_n$$

Además de ser asociativa por la izquierda, la aplicación tiene mayor precedencia que la abstracción, de forma que $\lambda x.MN$ debe leerse e interpretarse

como $\lambda x.(MN)$ y no $(\lambda x.M)N$. De cualquier forma, debe notarse que cada una de estas expresiones es válida, siendo expresiones equivalentes las dos primeras, en tanto que la tercera cambia el significado; sirva este ejemplo para hacer hincapié en la jerarquía y asociatividad tanto de la abstracción como de la aplicación.

Tómese en considereación que todas las variables del cálculo λ son anónimas, es decir, las variables de ligadura que utilizamos no poseen significado alguno y, así, sus nombres carecen de importancia. Retomando el ejemplo de la función identidad, podemos decir que tanto la expresión que utilizamos, $\lambda x.x$, como $\lambda y.y$ sirven para referirse a la misma función y por tanto son equivalentes. Definiremos los términos del cálculo λ como clases de equivalencia sobre los λ-términos bajo la reescritura de las variables de ligadura una vez que se introduzcan las definiciones necesarias para esto [6]. Esta relación se generará por medio de los axiomas. Para formular estos, es necesario definir previamente un operador para la *substitución de términos*: $M[x := N]$, el cual denota el resultado de substituir al término N por x dentro de M.

Definición 2.1.2. (Axiomas y reglas). El cálculo λ tiene por fórmulas

$$M = N$$

donde $M, N \in \Lambda$ y es axiomatizado por los siguientes axiomas y reglas:

1. $(\lambda x.M)N = M[x := N]$ (β-conversión),
2. $M = M$,
3. $M = N \Rightarrow N = M$,
4. $M = N, N = L \Rightarrow M = L$,
5. $M = N \Rightarrow MZ = NZ$,
6. $M = N \Rightarrow ZM = ZN$,
7. $M = N \Rightarrow \lambda x.M = \lambda x.N$ (regla ξ)

Nótese que el cálculo λ es libre de nociones lógicas: es una teoría ecuacional. Los conectivos y cuantificadores serán utilizados en el metalenguaje informal que ocupemos para hablar de nuestra teoría.

Centremos por un momento nuestra atención sobre la β-conversión, que es la regla más básica de reducción dentro del cálculo λ. Esta establece que la aplicación nos devuelve la substitución de un término por la variable ligada del término sobre el que funciona la aplicación. Por ejemplo:

$$(\lambda x.x)a = x[x := a] = a$$

El primer paso en esta forma de reducción consiste en determinar cuáles son las ocurrencias del término ligado por la abstracción que habrán de ser reemplazadas. Es necesario entonces tener una forma por medio de la cual podamos determinar el alcance de las ligaduras que imponen las abstracciones. Con este fin, definimos las variables libres de un λ-término, que son aquellas disponibles para ser substituidas en la β-conversión.

Definición 2.1.3. (Variable libre). Una variable x ocurre libre dentro de un λ-término M si x no existe en alcance de λx; de otra forma, decimos que x ocurre de forma *ligada*.

Ejemplo 2.1.1. La expresión

$$(\lambda x.x(\lambda x.x))a$$

Se reduce, por medio de la β-conversión a la siguiente:

$$a(\lambda x.x)$$

Pues la primer abstracción liga únicamente a la "primera" variable x, y no al cuerpo del λ-término que aparece en aplicación a la ocurrencia de esta misma x. Una expresión equivalente al ejemplo es:

$$(\lambda x.x(\lambda y.y))a$$

Definición 2.1.4. (FV). Denominamos como $FV(M)$ al conjunto de las variables libres de M, que puede ser definido de forma inductiva como:

- $FV(x) = \{x\}$,
- $FV(\lambda x.M) = FV(M) - \{x\}$,
- $FV(MN) = FV(M) \cup FV(N)$.

Decimos que M es cerrado o un *combinador* si $\mathrm{FV}(M) = \emptyset$. Podemos ahora también denotar al conjunto de todos los combinadores: $\Lambda^0 = \{M \in \Lambda \mid M \text{ es combinador}\}$.

Ejemplo 2.1.2. Retomemos el último ejemplo para mostrar cómo trabaja la función FV:

$$\begin{aligned} FV((\lambda x.x(\lambda x.x))a) &= FV((\lambda x.x(\lambda x.x))) \cup FV(a) && (2.1)\\ &= FV(\lambda x.x) \cup FV(\lambda x.x) \cup \{a\} && (2.2)\\ &= (FV(x) - \{x\}) \cup (FV(x) - \{x\}) \cup \{a\} && (2.3) \end{aligned}$$

$$= (\{x\} - \{x\}) \cup (\{x\} - \{x\}) \cup \{a\} \quad (2.4)$$
$$= \emptyset \cup \emptyset \cup \{a\} \quad (2.5)$$
$$= \{a\} \quad (2.6)$$

Para contrastar, veamos cómo actúa FV sobre un combinador. Tomemos para nuestro ejemplo el famoso combinador $\boldsymbol{K} = \lambda x.\lambda y.x$ del cálculo combinatorial:

$$FV(\boldsymbol{K}) = FV(\lambda x.\lambda y.x) \quad (2.1)$$
$$= FV(\lambda y.x) - \{x\} \quad (2.2)$$
$$= (FV(x) - \{y\}) - \{x\} \quad (2.3)$$
$$= (\{x\} - \{y\}) - \{x\} \quad (2.4)$$
$$= \{x\} - \{x\} \quad (2.5)$$
$$= \emptyset \quad (2.6)$$

Como era de esperarse, es vacío el conjunto de variables libres dentro de $\boldsymbol{K}$, pues sabíamos de antemano que $\boldsymbol{K}$ es un combinador.

Definición 2.1.5. (Subtérmino). Decimos que M es un subtérmino de N (notación $M \subset N$) si $M \in \text{Sub}(N)$, donde $\text{Sub}(N)$ es la colección de subtérminos de N, definida inductivamente como:

- $\text{Sub}(x) = \{x\}$,
- $\text{Sub}(\lambda x.N) = \text{Sub}(N) \cup \{\lambda x.N\}$,
- $\text{Sub}(NM) = \text{Sub}(N) \cup \text{Sub}(M) \cup \{NM\}$.

Definición 2.1.6. (Cambio de variables ligadas). El cambio de variables ligadas en un λ-término M es un reemplazo de una facción $\lambda x.N$ de M por $\lambda y.(N[x := y])$, donde y no ocurre ligada en N (es decir, y es variable libre). En caso contrario, al ser y ligada dentro de N, se generaría un conflicto de nombramiento que cambiaría el sentido de la expresión.

Decimos también que M es α-congruente con N, y lo denotamos como $M \equiv_\alpha N$ si N resulta de M por una serie de cambios de variables ligadas.

Esta relación forma clases de equivalencia sobre los λ-términos bajo el renombramiento de las variables ligadas, es decir, $\equiv_\alpha$ cumple con ser reflexiva, simétrica y transitiva:

- $\forall x(x \equiv_\alpha x)$,
- $\forall x, y(x \equiv_\alpha y \to y \equiv_\alpha x)$,

- $\forall x, y, z(x \equiv_\alpha y \wedge y \equiv_\alpha z \rightarrow x \equiv_\alpha z)$.

Es además congruente en tanto que $\equiv_\alpha$ es cerrada bajo las reglas de formación de nuestra sintaxis:

- Si $r_1 \equiv_\alpha r_2$ y $u_1 \equiv_\alpha u_2$ entonces $(r_1 u_1) \equiv_\alpha (r_2 u_2)$.
- Si $s_1 \equiv_\alpha s_2$ entonces $(\lambda x.s_1) \equiv_\alpha (\lambda x.s_2)$.

Definición 2.1.7. (Substitución). El resultado de substituir N por las ocurrencias libres de x en M, denotado por $M[x := N]$, se define de la siguiente manera:

- $x[x := N] \equiv N$,
- $y[x := N] \equiv y$, si $x \not\equiv y$,
- $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$,
- $(MN)[x := P] \equiv (M[x := P])(N[x := P])$.

Lema 2.1.1. *Para todo término t, existe un término s tal que este es α-congruente con t.*

Demostración: Por inducción en t: Para el caso de la abstracción, i.e., cuando $t = \lambda x.M$ tenemos, por hipótesis, un término s tal que $t \equiv_\alpha s$. De modo que s resultó de una serie de cambios de variables ligadas en t, por lo que es necesario y suficiente que las substituciones hechas hayan sido únicamente por variables que apareciesen libres en M. Los casos en los que t es una variable o una aplicación son inmediatos. □

Lema 2.1.2. *Si t y s son términos α-congruentes, entonces $FV(t) = FV(s)$.*

Demostración: Tenemos por hipótesis que $t \equiv_\alpha s$. Procedemos por inducción. Para el caso en el que t y s son abstracciones, tomamos a $t = \lambda x.y$ y $s = \lambda x'.y'$, y denotamos a las variables libres en el cuerpo de cada abstracción: $FV(y) = Y, FV(y') = Y'$. Tenemos también que $y[x := X] \equiv_\alpha y'[x' := X]$ tomando a X libre tanto en y como en y'. Si ahora $Z = FV(y[x := X])$, tenemos que $Z = Y$ si y sólo si $x \notin Y$, en tanto que esto implica que $X \notin Z$, por lo que no hay nada que substituir en y; en forma contraria, se tiene que $Z = (Y \backslash \{x\}) \cup \{X\}$. Para s se procede en forma análoga y, así, tenemos por hipótesis que $Z = Z'$ con $Z' = FV(y'[x' := X])$.

En el caso en el que se efectúan las substituciones, es decir, cuando $x \in Y$, tenemos que $X \in Z$, y análogo, $x' \in Y' \Rightarrow X \in Z'$, de donde $FV(y) = Y \backslash \{x\} = Z \backslash \{X\}$ y $FV(y') = Y' \backslash \{x'\} = Z' \backslash \{X\}$, y nuevamente, $Z = Z'$.

En los casos en los que tenemos una aplicación o variable, la igualdad en los conjuntos de variables libres es inmediato, en tanto que, por definición, $FV(MN) = FV(M) \cup FV(N)$ y $M_1N_1 \equiv_\alpha M_2N_2 \Leftrightarrow M_1 \equiv_\alpha M_2 \wedge N_1 \equiv_\alpha N_2$. □

2.2. β-reducción

Antes de definir a la β-reducción y establecer su relevancia en el contexto de una discusión introductoria al cálculo λ puro, es pertinente definir las relaciones de reducción y sus propiedades en un entorno más general.

Definición 2.2.1. (Relación compatible). Decimos que una relación $\mathcal{R}$ en Λ es compatible con las operaciones si:

$$\forall M, N, Z \in \Lambda[M\mathcal{R}N \Rightarrow (ZM)\mathcal{R}(ZN), (MZ)\mathcal{R}(NZ), (\lambda x.M)\mathcal{R}(\lambda x.N)]$$

Definición 2.2.2. (Relación de congruencia). Decimos que una relación $\mathcal{R}$ en Λ es de congruencia cuando es una relación compatible de equivalencia.

Definición 2.2.3. (Relación de reducción). Decimos que una relación $\mathcal{R}$ en Λ es una relación de reducción cuando es compatible, reflexiva y transitiva.

Definición 2.2.4. A continuación definimos de forma inductiva las relaciones $\to_\mathcal{R}$ ($\mathcal{R}$-reducción de un solo paso), $\twoheadrightarrow_\mathcal{R}$ ($\mathcal{R}$-reducción) y $=_\mathcal{R}$ ($\mathcal{R}$-equivalencia o $\mathcal{R}$-convertibilidad):

1. $\to_\mathcal{R}$ es la cerradura compatible de las relaciones binarias $\mathcal{R}$ de reducción en Λ:
   - i $(M, N) \in \mathcal{R} \Rightarrow M \to_\mathcal{R} N$,
   - ii $M \to_\mathcal{R} N \Rightarrow ZM \to_\mathcal{R} ZN$,
   - iii $M \to_\mathcal{R} N \Rightarrow MZ \to_\mathcal{R} NZ$,
   - iv $M \to_\mathcal{R} N \Rightarrow \lambda x.M \to_\mathcal{R} \lambda x.N$.
2. $\twoheadrightarrow_\mathcal{R}$ es la cerradura reflexiva transitiva de $\to_\mathcal{R}$:
   - i $M \to_\mathcal{R} N \Rightarrow M \twoheadrightarrow_\mathcal{R} N$,
   - ii $M \twoheadrightarrow_\mathcal{R} M$,
   - iii $M \twoheadrightarrow_\mathcal{R} N, N \twoheadrightarrow_\mathcal{R} L \Rightarrow M \twoheadrightarrow_\mathcal{R} L$.
3. $=_\mathcal{R}$ es la relación de equivalencia generada por $\twoheadrightarrow_\mathcal{R}$:

i $M \twoheadrightarrow_{\mathcal{R}} N \Rightarrow M =_{\mathcal{R}} N$,

ii $M =_{\mathcal{R}} N \Rightarrow N =_{\mathcal{R}} M$,

iii $M =_{\mathcal{R}} N, N =_{\mathcal{R}} L \Rightarrow M =_{\mathcal{R}} L$.

Es fácil notar, a partir de las definiciones dadas, que la relación de igualdad que establece la β-conversión (definición 2.1.2) puede ser también analizada como una reducción:

Definición 2.2.5. ($\boldsymbol{\beta}$-reducción).

$$\boldsymbol{\beta} = \{((\lambda x.M)N, M[x := N]) | M, N \in \Lambda\}$$

Ejemplo 2.2.1. A continuación analizamos un ejemplo breve que ilustra a la β-reducción:

$$\begin{aligned}(\lambda x.xx)(\lambda y.y)z &\rightarrow_{\boldsymbol{\beta}} (\lambda y.y)(\lambda y.y)z && (2.1)\\ &\rightarrow_{\boldsymbol{\beta}} (\lambda y.y)z && (2.2)\\ &\rightarrow_{\boldsymbol{\beta}} z && (2.3)\end{aligned}$$

Recordando que la aplicación asocia a la izquierda, el primer término que hallamos al hacer la reducción es $(\lambda x.xx)(\lambda y.y)$, de forma que la $\boldsymbol{\beta}$-reducción funciona de la siguiente manera:

$$(\lambda x.xx)(\lambda y.y) \rightarrow (xx)[x := (\lambda y.y)]$$

Efectuando la substitución, obtenemos la parte derecha en 2.1:

$$(\lambda y.y)(\lambda y.y)z$$

Siguiendo la cadena de reducciones, llegamos finalmente a z, lo que significa tanto que $(\lambda x.xx)(\lambda y.y)z \twoheadrightarrow_{\boldsymbol{\beta}} z$ como que $(\lambda x.xx)(\lambda y.y)z =_{\boldsymbol{\beta}} z$.

Definición 2.2.6. ($\mathcal{R}$-reducto). Un $\mathcal{R}$-reducto es un término M tal que $(M, N) \in \mathcal{R}$ para algún término N. En este caso, decimos que N es un $\mathcal{R}$-contractum de M.

Definición 2.2.7. ($\mathcal{R}$-fn). Decimos que un término M es $\mathcal{R}$-fn (de "forma normal") si M no contiene como subtérmino de sí ningún $\mathcal{R}$-reducto. En el contexto de la $\boldsymbol{\beta}$-reducción, diremos que un término está en forma normal cuando este ya no puede ser reducido.

Cabe ahora hacer algunas preguntas que surgen de forma natural una vez que se han introducido las definiciones más básicas relacionadas a la principal forma de reducción de nuestra teoría: ¿todos los términos que son reducidos por medio de la $\boldsymbol{\beta}$-reducción llegan a una forma normal? Dado un λ-término, es a veces posible tener múltiples términos que pueden ser sustituidos. ¿Afecta de alguna manera el orden en que se efectúa la reducción, o se alcanza siempre una forma normal única? La primera de nuestras preguntas puede ser fácilmente respondida de forma negativa mediante un contraejemplo. Considerse la siguiente expresión:

$$(\lambda x.xx)(\lambda x.xx)$$

Reduciendo esta expresión conforme las reglas establecidas, obtenemos

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\to (xx)[x := (\lambda x.xx)]\\ (\lambda x.xx)(\lambda x.xx) &\to (xx)[x := (\lambda x.xx)]\\ (\lambda x.xx)(\lambda x.xx) &\to \dots\end{aligned}$$

Incluso es posible que los términos no se reduzcan sino que crezcan:

$$\begin{aligned}(\lambda x.xxx)(\lambda x.xxx) &\to (xxx)[x := (\lambda x.xxx)]\\ (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) &\to (xxx)[x := (\lambda x.xxx)]\\ (\lambda x.xxx)\dots(\lambda x.xxx) &\to \dots\end{aligned}$$

Para responder a nuestra segunda pregunta, usemos la función constante $\lambda x.y$, la cual, independiente del argumento que recibe, devuelve siempre a y, y apliquémosle la expresión recién utilizada, $(\lambda x.xx)(\lambda x.xx)$. Si bien no hay un orden explícito en el que deban ser evaluados los términos al emplear la $\boldsymbol{\beta}$-reducción, el hecho de que la aplicación tenga mayor precedencia que la abstracción y asocie por la derecha, así como que la abstracción lo haga por izqueirda, establece un orden implícito al que llamaremos *orden aplicativo*, que se caracteriza por reducir los argumentos de las funciones a una forma normal antes de evaluarlos en el $\mathcal{R}$-reducto de mayor nivel en la expresión.

Nombremos a $\lambda x.xx$ como $\boldsymbol{\omega}$. Ahora, haciendo uso del orden aplicativo de reducción, es fácil ver que pronto nos atoramos en un ciclo infinito al evaluar:

$$\lambda x.y(\boldsymbol{\omega\omega}) \to \lambda x.y(\boldsymbol{\omega\omega}) \to \cdots \to \lambda x.y(\boldsymbol{\omega\omega}) \to \dots$$

La evaluación por medio del orden aplicativo, al reducir primero los argumentos para posteriormente sustituirlos, no encuentra una forma normal para las expresiones que no son fuertemente normalizables, como ocurre en el ejemplo.

Definición 2.2.8. (Fuertemente normalizable). Decimos que t es fuertemente normalizable si existe una sequencia de reducción finita $t \to t_1 \to t_2 \to \cdots \to t_n$.

Es importante mencionar que el orden aplicativo no es el único por el que se pueda optar. En particular, el *orden normal* reduce primero a los $\mathcal{R}$-reductos antes de aplicar cualquier argumento. De forma que evaluando la misma expresión, ahora usando el orden normal, obtenemos:

$$\lambda x.y(\boldsymbol{\omega\omega}) \to y$$

pues la función constante ignora a cualquier argumento. Así, el orden normal, al sustituir argumentos antes de reducir cualquier cosa, llega a una forma normal, mientras que el orden aplicativo nunca termina. A lo largo de este trabajo, optaremos por el uso del orden aplicativo. Siempre que el orden de reducción utilizado sea otro esto se hará explícito.

2.3. Puntos fijos

En matemáticas, el término "punto fijo" se usa para referirse a los elementos del dominio de una función tales que al ser aplicados a esta se obtiene como resultado a estos mismos [21]. Por ejemplo, si definimos la función f en $\mathbb{R} : f(x) = x^3$, esta tiene tres puntos fijos: $0, 1$ y -1. En el contexto del cálculo λ, el punto fijo de una función arbitraria f es a su vez una función x que no cambia bajo la aplicación de la función f. Es decir: $x = fx$.

El siguiente teorema es uno de los más importantes dentro del cálculo λ, pues establece la existencia de puntos fijos para cualquier función dentro de Λ. Es además, por medio de los puntos fijos, que podremos definir las funciones recursivas en el ámbito del cálculo λ puro.

Teorema 2.3.1. *Teorema del punto fijo:*

$$\forall F \exists X (FX = X)$$

Demostración: Sea $W \equiv \lambda x.F(xx)$ y $X \equiv WW$. De tal forma, tenemos que:

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) = FX. \quad \square$$

Este extraordinario teorema, tanto más cuando vemos que tiene una demostración sorprendentemente simple, lleva a uno de los resultados más significativos para el cálculo λ: que cualquier función recursiva puede ser escrita de una forma no-recursiva. A continuación definimos conceptos relacionados antes de dar ejemplos puntuales de esto.

Definición 2.3.1. (Combinador de punto fijo). Un combinador de punto fijo es un término M tal que

$$\forall F(MF = F(MF))$$

(Es decir, MF es un punto dijo de F).

Corolario 2.3.2. *Sea* $\boldsymbol{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. *Entonces es* $\boldsymbol{Y}$ *un combinador de punto fijo.*

Demostración: Sea $W = \lambda x.F(xx)$, entonces tenemos:

$$\boldsymbol{Y}F = WW = F(WW) = F(\boldsymbol{Y}F). \quad \square$$

El término $\boldsymbol{Y}$ fue expuesto por Haskell Curry como el "combinador paradójico" [22].

Dado que las funciones del cálculo λ son todas anónimas, no es concebible una forma en la que podríamos construir una función que se referencíe a sí; de tener una función f tal, ésta tendría la forma:

$$f \equiv \ldots f \ldots$$

En lugar de esto, podemos definir a una función F que tome a f como argumento

$$F \equiv \lambda f. \ldots . f \ldots$$

Ahora, el problema se reduce a encontrar una función x que pueda aplicarse a F de forma tal que $Fx \equiv x$, pues siendo x lo que se aplica a F, es en verdad lo que toma el valor de nuestra f original. Así, queremos que Fx se comporte de la misma forma en la que x lo haría (i.e., al igual que hubiese hecho f de ser posible en el cálculo λ tener expresiones autorreferenciales). De esta manera, nuestro problema no es otro que el de encontrar un punto fijo de F. Como demostramos en 2.3.2, $\boldsymbol{Y}$ es un combinador de punto fijo, lo que quiere decir que para cualquier función $F \in \Lambda$, $\boldsymbol{Y}F = F(\boldsymbol{Y}F)$. De esta forma, tenemos que $x = \boldsymbol{Y}F$ es la función que buscamos. Contamos ahora con una función recursiva válida dentro del cálculo λ que no incurre en una autorreferencia directa.

A continuación, ilustramos el uso de los puntos fijos para la formación de funciones recursivas dentro del cálculo λ con ejemplos concretos. Asúmase que las sentencias condicionales así como la comparación booleana y las operaciones aritméticas han sido definidas con anterioridad y cuentan con la interpretación común.

Ejemplo 2.3.1. Suma de $x, y \in \mathbb{N}$:

$$f \equiv \lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } f(x+1)(y-1))$$

$$F \equiv \lambda f.\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } f(x+1)(y-1))$$

Finalmente, la suma de dos números naturales toma la forma de $F(\boldsymbol{Y}F)$. Como ejemplo, realicemos la suma $2 + 2$:

$$\begin{aligned}
&F(\boldsymbol{Y}F)(2\ 2) \rightarrow \\
&\lambda f.\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } f(x+1)(y-1))(\boldsymbol{Y}F)(2\ 2) \rightarrow \\
&(\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } (\boldsymbol{Y}F)(x+1)(y-1)))(2\ 2) \rightarrow \\
&(\lambda y.(\texttt{if } y == 0 \texttt{ then } 2 \texttt{ else } (\boldsymbol{Y}F)(2+1)(y-1)))(2) \rightarrow \\
&(\texttt{if } 2 == 0 \texttt{ then } 2 \texttt{ else } (\boldsymbol{Y}F)(2+1)(2-1)) \rightarrow \\
&(\boldsymbol{Y}F)(3\ 1) \rightarrow \\
&F(\boldsymbol{Y}F)(3\ 1) \rightarrow \\
&\lambda f.\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } f(x+1)(y-1))(\boldsymbol{Y}F)(3\ 1) \rightarrow \\
&(\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } (\boldsymbol{Y}F)(x+1)(y-1)))(3\ 1) \rightarrow \\
&(\lambda y.(\texttt{if } y == 0 \texttt{ then } 3 \texttt{ else } (\boldsymbol{Y}F)(3+1)(1-1)))(1) \rightarrow \\
&(\text{if } 1 == 0 \texttt{ then } 3 \texttt{ else } (\boldsymbol{Y}F)(3+1)(1-1)) \rightarrow \\
&(\boldsymbol{Y}F)(4\ 0) \rightarrow \\
&F(\boldsymbol{Y}F)(4\ 0) \rightarrow \\
&\lambda f.\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } f(x+1)(y-1))(\boldsymbol{Y}F)(4\ 0) \rightarrow \\
&(\lambda x.\lambda y.(\texttt{if } y == 0 \texttt{ then } x \texttt{ else } (\boldsymbol{Y}F)(x+1)(y-1)))(4\ 0) \rightarrow \\
&(\lambda y.(\texttt{if } y == 0 \texttt{ then } 4 \texttt{ else } (\boldsymbol{Y}F)(4+0)(y-1)))(0) \rightarrow \\
&(\texttt{if } 0 == 0 \texttt{ then } 4 \texttt{ else } (\boldsymbol{Y}F)(3+1)(0-1)) \rightarrow \\
&4.
\end{aligned}$$

Ejemplo 2.3.2. La función factorial:

$$f \equiv \lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * f(x-1)))$$

$$F \equiv \lambda f.\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * f(x-1)))$$

De igual forma que en el ejemplo anterior, concluimos que la función que computa el factorial de un número $x \in \mathbb{N}$ toma la forma de $F(\boldsymbol{Y}F)$. Como ejemplo, computemos el factorial de 2:

$$\begin{aligned}
&F(\boldsymbol{Y}F)(2) \rightarrow \\
&(\lambda f.\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * f(x-1))))(\boldsymbol{Y}F)(2) \rightarrow
\end{aligned}$$

$$
\begin{aligned}
(\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * (\boldsymbol{Y}F)(x-1))))(2) &\rightarrow \\
(\texttt{if } 2 == 0 \texttt{ then } 1 \texttt{ else } (2 * (\boldsymbol{Y}F)(2-1))) &\rightarrow \\
2 * (\boldsymbol{Y}F)(1) &\rightarrow \\
2 * (F(\boldsymbol{Y}F))(1) &\rightarrow \\
2 * (\lambda f.\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * f(x-1))))(\boldsymbol{Y}F)(1) &\rightarrow \\
2 * (\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * (\boldsymbol{Y}F)(x-1))))(1) &\rightarrow \\
2 * (\texttt{if } 1 == 0 \texttt{ then } 1 \texttt{ else } (1 * (\boldsymbol{Y}F)(1-1))) &\rightarrow \\
2 * 1 * (\boldsymbol{Y}F)(0) &\rightarrow \\
2 * 1 * (F(\boldsymbol{Y}F))(0) &\rightarrow \\
2 * 1 * (\lambda f.\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * f(x-1))))(\boldsymbol{Y}F)(0) &\rightarrow \\
2 * 1 * (\lambda x.(\texttt{if } x == 0 \texttt{ then } 1 \texttt{ else } (x * (\boldsymbol{Y}F)(x-1))))(0) &\rightarrow \\
2 * 1 * (\texttt{if } 0 == 0 \texttt{ then } 1 \texttt{ else } (0 * (\boldsymbol{Y}F)(0-1))) &\rightarrow \\
2 * 1 * 1 &\rightarrow \\
2.
\end{aligned}
$$

La posibilidad de definir funciones recursivas sin recurrir a la autorreferencia es en gran medida lo que confiere al cálculo λ su practicidad (y popularidad) para ser modelo del cómputo.

En los ejemplos desarrollados para ejemplificar el uso de los puntos fijos en la formulación de funciones recursivas dentro del cálculo λ, asumimos por practicidad la validez del uso de las sentencias condicionales, expresiones booleanas y operaciones aritméticas dentro de nuestro lenguaje. Antes de exponer y desarrollar el tratamiento de los números naturales dentro del cálculo λ, codificaremos dentro de este los valores booleanos de verdadero y falso, los operadores lógicos más comunes y las sentencias condicionales, posponiendo las operaciones aritméticas hasta la siguiente sección.

Definición 2.3.2. (Valores booleanos). Definimos los valores booleanos de verdadero y falso como:

$$\mathcal{V} = \lambda x.\lambda y.x$$
$$\mathcal{F} = \lambda x.\lambda y.y$$

De forma tal que $\mathcal{V}$ queda definida como una función que requiere de dos argumentos, de los cuales regresará siempre el primero, en tanto que $\mathcal{F}$ es una función análoga que regresará siempre el segundo.

Definición 2.3.3. (Sentencias condicionales). Contando ya con una forma en la que podemos expresar la validez o falsedad dentro del cálculo λ, podemos ahora definir a las sentencias condicionales:

$$\texttt{if } b \texttt{ then } s \texttt{ else } t$$

como bst, pues b podrá reducirse únicamente a nuestros valores booleanos primeramente definidos, $\mathcal{V}$ y $\mathcal{F}$, y como consecuencia de ello, se reducirá a s siempre que se tenga $\mathcal{V}st$ y a t cuando $\mathcal{F}st$.

Definición 2.3.4. ($\wedge$.): La función para la conjunción de dos argumentos se define como:

$$\wedge \equiv \lambda x \lambda y.xy\mathcal{F}$$

Definición 2.3.5. ($\vee$). La función para la disyunción de dos argumentos se define como:

$$\vee \equiv \lambda x \lambda y.x\mathcal{V}y$$

Definición 2.3.6. ($\neg$). La función para la negación se define como:

$$\neg \equiv \lambda x.x\mathcal{F}\mathcal{V}$$

La interpretación de estas últimas tres funciones es la común de la lógica proposicional.

x	y	$\wedge$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{V}$	$\mathcal{F}$
$\mathcal{V}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{V}$	$\mathcal{V}$	$\mathcal{V}$

x	y	$\vee$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{F}$	$\mathcal{V}$	$\mathcal{V}$
$\mathcal{V}$	$\mathcal{F}$	$\mathcal{V}$
$\mathcal{V}$	$\mathcal{V}$	$\mathcal{V}$

x	$\neg$
$\mathcal{F}$	$\mathcal{V}$
$\mathcal{V}$	$\mathcal{F}$

2.4. Números naturales

Como ya se ha mencionado en esta brevísima introducción al cálculo λ, todos los objetos dentro de este son funciones, y los números no son excepción. La representación de los números naturales dentro del cálculo λ, tal como fue expuesta por Alonzo Church en *The Calculi of Lambda-Conversion* [23], se presenta de la siguiente forma:

Definición 2.4.1. (Números naturales). Definimos a los números naturales de la siguiente manera:

$$\bar{1} \equiv \lambda s.\lambda z.sz$$
$$\bar{2} \equiv \lambda s.\lambda z.s(sz)$$
$$\bar{3} \equiv \lambda s.\lambda z.s(s(sz))$$
$$\vdots$$

Así, todos los números se definen como funciones de dos argumentos que aplican un s número de veces la primera función a la segunda. En su libro, Church no define al $\bar{0}$, lo cual es fácil de hacer una vez que dotamos de interpretación a lo que son las funciones que representan a los números naturales dentro del cálculo λ:

$$\bar{0} \equiv \lambda s.\lambda z.z$$

Es importante notar el uso de la barra sobre los numerales para evitar ambigüedades, como podría ser interpretar 11 como el número once en lugar de $\bar{1}\bar{1} \equiv (\lambda s.\lambda z.sz)(\lambda s.\lambda z.sz)$.

La primer función que definimos sobre los números naturales es la función sucesor:

Definición 2.4.2. (Sucesor).

$$\mathcal{S} \equiv \lambda a.\lambda b.\lambda c.b(abc)$$

Ejemplo 2.4.1. Apliquemos la función sucesor a nuestra representación del cero:

$$\begin{aligned}
\mathcal{S}\bar{0} \equiv & (\lambda a.\lambda b.\lambda c.b(abc))(\lambda s.\lambda z.z) \\
\to & \lambda b.\lambda c.b((\lambda s.\lambda z.z)bc) \\
\to & \lambda b.\lambda c.b(\lambda z.z)c \\
\to & \lambda b.\lambda c.bc \\
\equiv_\alpha & \lambda s.\lambda z.sz \\
\equiv & \bar{1}
\end{aligned}$$

De forma similar definimos la suma, multiplicación y exponenciación:

Definición 2.4.3. (Suma).

$$M + N \equiv \lambda a.\lambda b(((Ma)(Na))b)$$

Ejemplo 2.4.2. Ejemplificamos la suma computando la sencilla operación de $\bar{1} + \bar{1}$:

$$\begin{aligned}
\bar{1} + \bar{1} \equiv & \lambda a.\lambda b.(((\bar{1}a)(\bar{1}a))b) \\
\lambda a.\lambda b.(((\bar{1}a)(\bar{1}a))b) \to & \lambda a.\lambda b.((((\lambda s.\lambda z.sz)a)((\lambda s.\lambda z.sz)a))b) \\
\lambda a.\lambda b.(((\lambda z.az)(\lambda z.az))b) \to & \lambda a.\lambda b.((a(\lambda z.az)b) \\
\lambda a.\lambda b.((a(\lambda z.az)b) \to & \lambda a.\lambda b.((a(ab)) \\
\lambda a.\lambda b.a(ab) \equiv & \bar{2}
\end{aligned}$$

Definición 2.4.4. (Multiplicación).

$$M * N \equiv \lambda a.(M(Na))$$

Ejemplo 2.4.3. Ejemplificamos la multiplicación realizando la operación de $\bar{2} * \bar{2}$:

$$\begin{aligned}
\bar{2} * \bar{2} &\equiv \lambda a.(\bar{2}(\bar{2}a)) \\
\lambda a.(\bar{2}(\bar{2}a)) &\to \lambda a.(\lambda s.\lambda z.s(sz)([\lambda s.\lambda z.s(sz)]a)) \\
\lambda a.(\lambda s.\lambda z.s(sz)([\lambda s.\lambda z.s(sz)]a)) &\to \lambda a.(\lambda s.\lambda z.s(sz)(\lambda z.a(az))) \\
\lambda a.(\lambda s.\lambda z.s(sz)(\lambda z.a(az))) &\to \lambda a.(\lambda z.(\lambda z.a(az))((\lambda z.a(az))z)) \\
\lambda a.(\lambda z.(\lambda z.a(az))((\lambda z.a(az))z)) &\to \lambda a.(\lambda z.(\lambda z.a(az))(a(az))) \\
\lambda a.(\lambda z.(\lambda z.a(az))(a(az))) &\to \lambda a.((\lambda z.a(a(a(az))))) \\
\lambda a.((\lambda z.a(a(a(az))))) &\to \lambda a.\lambda z.a(a(a(az))) \\
\lambda a.\lambda z.a(a(a(az))) &\equiv_\alpha \bar{4}
\end{aligned}$$

Definición 2.4.5. (Exponenciación).

$$M^N \equiv NM$$

Ejemplo 2.4.4. Como última ejemplificación de operaciones sobre los números naturales dentro del cálculo λ, mostramos la exponenciación de $\bar{2}^{\bar{2}}$:

$$\begin{aligned}
\bar{2}^{\bar{2}} &\equiv (\lambda s.\lambda z.s(sz))(\lambda s.\lambda z.s(sz)) \\
(\lambda s.\lambda z.s(sz))(\lambda s.\lambda z.s(sz)) &\to_\alpha \lambda z.[\lambda s.\lambda x.s(sx)]([\lambda s.\lambda x.s(sx)]z) \\
\lambda z.[\lambda s.\lambda x.s(sx)]([\lambda s.\lambda x.s(sx)]z) &\to \lambda z.[\lambda s.\lambda x.s(sx)](\lambda x.z(zx)) \\
\lambda z.[\lambda s.\lambda x.s(sx)](\lambda x.z(zx)) &\to_\alpha \lambda z.[\lambda x.(\lambda y.z(zy))((\lambda y.z(zy))x)] \\
\lambda z.[\lambda x.(\lambda y.z(zy))((\lambda y.z(zy))x)] &\to \lambda z.[\lambda x.(\lambda y.z(zy))(z(zx))] \\
\lambda z.[\lambda x.(\lambda y.z(zy))(z(zx))] &\to \lambda z.\lambda x.z(z(z(zx))) \\
\lambda z.\lambda x.z(z(z(zx))) &\equiv_\alpha \bar{4}
\end{aligned}$$

2.5. Índices de de Bruijn

Dentro del cálculo λ es posible renombrar en cualquier momento las variables ligadas de una expresión, ya sea para evitar la colisión de símbolos dentro de esta, o bien porque un distinto nombramiento de las variables pudiese resultar más conveniente para los fines buscados.

Ejemplo 2.5.1. Desarrollando el último ejemplo de la subsección anterior absteniéndonos de utilizar la α-conversión, es fácil ver las repercusiones semánticas que derivan de la colisión de símbolos dentro de una expresión:

$$\begin{aligned}
\bar{2}^{\bar{2}} \equiv & (\lambda s.\lambda z.s(sz))(\lambda s.\lambda z.s(sz)) \\
(\lambda s.\lambda z.s(sz))(\lambda s.\lambda z.s(sz)) \rightarrow & \lambda z.[\lambda s.\lambda z.s(sz)]([\lambda s.\lambda z.s(sz)]z) \\
\lambda z.[\lambda s.\lambda z.s(sz)]([\lambda s.\lambda z.s(sz)]z) \rightarrow & \lambda z.[\lambda s.\lambda z.s(sz)](\lambda z.z(zz)) \\
\lambda z.[\lambda s.\lambda z.s(sz)](\lambda z.z(zz)) \rightarrow & \lambda z.[\lambda s.\lambda z.[\lambda z.z(zz)]([\lambda z.z(zz)]z)] \\
\lambda z.[\lambda s.\lambda z.[\lambda z.z(zz)]([\lambda z.z(zz)]z)] \rightarrow & \lambda z.[\lambda s.\lambda z.[\lambda z.z(zz)](z(zz))] \\
\lambda z.[\lambda s.\lambda z.[\lambda z.z(zz)](z(zz))] \rightarrow & \lambda z.[\lambda s.\lambda z.(z(zz))[(z(zz))(z(zz))]]
\end{aligned}$$

El nombramiento de las variables del cálculo λ no afecta en forma alguna el significado de los diversos términos que podamos expresar. Si bien el uso de algunos símbolos es más frecuente que el de otros a lo largo de la literatura, ya sea por convención o facilidad en torno a la discusión de diversos conceptos o demostraciones, los resultados que puedan derivarse dentro del cálculo λ son independientes de la denominación que pudiera tener cualquier símbolo empleado en sus expresiones [24]. Tomando esto en cuenta, sería conveniente establecer alguna convención en torno al nombramiento de nuestras variables, pues no sólo facilitaría la mecanización del cálculo λ, sino que también podría servir para facilitar la constante labor del renombramiento de símbolos.

La principal causa del renombramiento de las variables dentro de una expresión deriva de la necesidad que existe de establecer la equivalencia entre dos expresiones cuya única diferencia estriba en el nombramiento de estas (es decir, son α-equivalentes) [25]. El matemático holandés de Bruijn (1918 - 2012) notó que la α-equivalencia permite, dentro del cálculo λ, establecer una notación que sea capaz de prescindir de la α-conversión abandonando el uso de las variables y optando por la utilización de índices para referir a la abstracción a la que las variables son ligadas [26].

Algunas de las opciones que tenemos para la representación de las variables dentro del cálculo λ son:

- Representar las variables de forma simbólica y continuar utilizando la α-conversión siempre que sea necesaria, como hemos hecho hasta este momento.
- Utilizar la convensión de Barendregt: introducir la condición de que los símbolos de todas las variables ligadas deben ser todos diferentes unos de otros, así como de toda variable libre que se utilice.
- Optar por una representación de las variables en términos que no requieran del renombramiento de los símbolos que se utilizan para estas.

El mismo de Bruijn enumera en su famoso artículo "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem"[1] los criterios que debiesen utilizarse para elegir a una notación sobre otra:

1. Es fácil de escribir y leer para un humano.
2. Es fácil de manipular en discusiones metalingüísticas.
3. Es fácil de utilizar para los programadores, así como de manipular para las computadoras.

Y apunta que su propuesta es especialmente útil para los puntos dos y tres.

La idea de de Bruijn es representar a las variables con números que den cuenta de cuál es la abstracción a la que están ligadas, en lugar de utilizar cualquier otro símbolo. Por ejemplo, el λ-término que denota la función identidad, $\lambda x.x$ cambia a $\lambda 1$ en tanto que nuestro combinador $\boldsymbol{K} \equiv \lambda x.\lambda y.x$ se expresa en notación de de Bruijn como $\lambda\lambda 2$.

La notación de de Bruijn tiene dos principales ventajas sobre la que hemos usado hasta ahora. La primera y más notable es que la α-equivalencia se reduce a la igualdad sintáctica de los términos, la segunda es que no existe ningún riesgo en términos de la colisión de símbolos dentro de las expresiones. Como vimos ya, las variables ligadas dentro de una expresión son representadas por los índices de de Bruijn que toman valores de números naturales. Esta notación hace que sea innecesario nombrar algún símbolo que represente a la variable abstraida.

Los términos que contienen variables libres pueden ser también tratados por esta nueva notación introduciendo *contextos de nombramiento*, Γ, que establezcan un conjunto de números que habrán de usarse para denotar las variables libres [24].

En todo el código desarrollado dentro del verificador de pruebas Coq en el marco de este trabajo, establecemos de forma arbitraria y como variables libres a los índices con un valor mayor o igual a 100. A continuación reescribimos las definiciones de las funciones sucesor, suma y multiplicación, esta vez utilizando la sintaxis expuesta en esta subsección, misma que continuaremos usando en los capítulos posteriores.

Definición 2.5.1. (Sucesor).

$$\mathcal{S} \equiv \lambda\lambda\lambda 2(3\ 2\ 1)$$

[1] Una notación para el cálculo lambda utilizando marcadores sin nombre, una herramienta para la manipulación automática con aplicación al teorema de Church-Rosser.

Definición 2.5.2. (Suma).

$$M + N \equiv \lambda\lambda(((M2)(N2))1)$$

Definición 2.5.3. (Multiplicación).

$$M * N \equiv \lambda(M(N1))$$

2.6. Confluencia

Es menester, como última anotación de esta breve introducción al cálculo λ, establecer la unicidad de la forma normal de los λ-términos, sospecha que salta a la vista con la presentación de la α-equivalencia y que se ve confirmada por la notación sintáctica dada por de Bruijn. Los términos aquí enunciados de forma intuitiva serán formalizados más adelante, en el capítulo quinto, donde daremos una prueba de la confluencia para los términos del cálculo $\lambda\boldsymbol{SF}$ bajo su regla de reducción.

Definición 2.6.1. (Relación confluente). Decimos que una relación $\mathcal{R}$ es confluente si para cualquier término t y par de sucesión de reducciones $t \to \cdots \to t_1$ y $t \to \cdots \to t_2$ existe un término t_3 y par de sucesión de reducciones $t_1 \to \cdots \to t_3$ y $t_2 \to \cdots \to t_3$.

Esta definición da pie al siguiente teorema que presentamos aquí sin su demostración, debido a la extensión de la misma. Sin embargo, el siguiente teorema, uno de los más importantes en el contexto del cálculo λ, ha sido demostrado varias veces y formalizado tantas más. El lector puede referirse a [6] y [27] para un tratamiento detallado de su demostración.

Teorema 2.6.1. *Confluencia del cálculo λ: Para cualquier expresión M del cálculo λ, si $M \to \cdots \to K$ y $M \to \cdots \to L$, entonces existe una expresión N tal que $K \to \cdots \to N$ y $L \to \cdots \to N$ módulo α-conversión.*

El anterior teorema establece un resultado importante en tanto que implica que el orden de reducción que se tome para los reductos dentro de las expresiones del cálculo λ no afecta en forma alguna al producto de las mismas reducciones. Asimismo, es importante resaltar que este teorema no implica en forma alguna que las reducciones que se hagan sobre las expresiones deban llevar a formas normales, sino que de hacerlo, estas serán únicas.

Capítulo 3

Lógica combinatoria

"In the beginning, the forest gods started the forest with just two birds —the starling $\boldsymbol{S}$ *and the kestrel* $\boldsymbol{K}$*. (...)"*

Raymond Smullyan [28]

El término "lógica combinatoria" designa a la parte de la lógica matemática que tiene una relación esencial con los combinadores, incluyendo todo lo necesario para una fundación adecuada de las teorías lógicas más usuales. La lógica aquí expuesta tiene que ver con los combinadores y las relaciones formales que guardan tanto entre ellos como consigo mismos en referencia con la relación de igualdad. Es a esto a lo que llamaremos *lógica combinatoria pura*, única que expondremos en este capítulo.

3.1. Breve marco histórico

Es importante recordar y tener presente el paisaje que existía en torno a la lógica y los fundamentos de la matemática a principios del siglo XX para entender mejor el contexto en el que se realizaron las investigaciones que desembocaron en las propuestas de los sistemas formales que hoy día estudiamos bajo el nombre de cálculo λ y lógica combinatoria, ambos desarrollados en la tercera década del siglo XX y enfocados en la descripción de diversas nociones de caracter sumamente básico que fue (y sigue siendo) obviado pese a constituir conceptos sumamente complicados que forman parte de una "prelógica" y que, como hemos visto en el capítulo anterior, requieren de un análisis profundo, como son los procesos de substitución, generalmente

indicados por el uso de las variables, la aplicación y la abstracción en un entorno general.

En el tiempo en el que surgían las investigaciones de Church y Schönfinkel, la paradoja de Russell era algo bastante reciente, los teoremas de Gödel no eran todavía publicados, las obras de Frege empezaban a ser discutidas en la comunidad más amplia de matemáticos dedicados a la lógica, y existía un amplio grupo de personas dedicadas a la construcción de sistemas que pretendían ser fundaciones consistentes para todo el aparato matemático con el que se trabajaba. Algunas de estas fundaciones (las más famosas), se basaron en el concepto de conjunto; es su evolución la que hoy día reina como la herramienta fundacionista más popular.

La falta de un consenso general en esta época respecto a cuál sistema sirviese mejor a las aspiraciones fundacionistas de la matemática incentivó la propuesta de diversos sistemas fundamentales basados en conceptos distintos al de conjunto, como aquellos que retomaron a las funciones, tal como la lógica combinatoria, que en sus principios surge no como un sistema elaborado de forma aislada, sino como parte de un proyecto fundacionista más grande a partir de las intuiciones de lo que es una función y la formalización de esta.

La lógica combinatoria se establece a partir de una serie de conferencias dadas por Moses Schönfinkel en Göttingen, Alemania, las cuales fueron posteriormente preparadas y redactadas para su publicación en forma de artículo por Heinrich Behmann en el año de 1924 [29]. En dicho artículo se discute la atractiva posibilidad de la eliminación de variables cuantificadas dentro de la lógica de predicados a partir de la introducción de la idea de lo que hemos denominado hasta el momento como aplicación, mostrando que las funciones de un número n de variables pueden a su vez ser sustituidas al introducir la fructífera idea, vista ya en el ámbito del cálculo λ, de que las funciones pueden ser aplicadas como argumento a sí mismas o a otras, cosa que, como demostramos en el anterior capítulo (2.0.1) no era posible en otras teorías fundacionistas como la teoría de conjuntos. El mecanismo aplicativo para las funciones usado por Schönfinkel había sido ya mencionado en los trabajos de Frege y su idea principal era la de tomar una función F de dos variables ($F(x, y)$) y reconstruirla como Fxy, donde esta debe tomarse no como $F(x, y)$, sino como $(Fx)y$, haciendo a F una función de aridad unitaria cuyo valor es una función de misma aridad (comentarios de Quine a [29]). La misma idea se aplica a funciones de cualquier número de variables.

Si bien la eliminación de variables cuantificadas en el contexto de la lógica matemática había sido ya explorada a partir de álgebras, no fue sino hasta la publicación de las investigaciones hechas en la Sociedad Matemática de Göttingen por Schönfinkel que se construye un sistema lo suficientemente sólido para abarcar a todos los cuantificadores y las variables ligadas a estos

[30, 31].

En su artículo fundacional para la lógica combinatoria, "On the building blocks of mathematical logic", Moses Schönfinkel, siguiendo el método axiomático establecido por Hilbert (con quien trabajó en Göttingen), caracterizado por la búsqueda por mantener al conjunto de axiomas consistentes de una teoría, así como las nociones fundamentales no definidas de la misma, como el mínimo posible, parte de examinar resultados contemporáneos de la lógica proposicional con este mismo propósito.

Dentro de la lógica, es un resultado común la imposibilidad de definir todas las conectivas empleadas para relacionar las fórmulas bien formadas (a saber: la negación, disyunción, conjunción, implicación y doble implicación) a partir de una sola de ellas.

Si bien es posible reducirlas únicamente al par de la negación en dupla con la disyunción, conjunción o implicación, como en trabajos de finales del siglo XIX y principios del siglo XX hicieron ya Whitehead, Russell o Frege, el que la reducción a un conectivo fundamental sea imposible a partir de los conectivos clásicos condujo al análisis de otras formas de conectivos que no guardan el mismo significado de los conectivos lógicos clásicos. Fue de esta manera como primero Pierce [32], sin publicación, y más tarde Sheffer, en 1913 [33], tomando como base a la conectiva lógica cuya interpretación semántica es la de "no a y no b" (hoy conocida como el operador "`NOR`" o, en el ámbito de la ingeniería eléctrica y electrónica, como la compuerta lógica universal) lograron reducir todos los conectivos lógicos a uno solo. Denotaremos el operador de Sheffer como $\downarrow$ y daremos la siguiente interpretación:

x	y	$\downarrow$
$\mathcal{F}$	$\mathcal{F}$	$\mathcal{V}$
$\mathcal{F}$	$\mathcal{V}$	$\mathcal{F}$
$\mathcal{V}$	$\mathcal{F}$	$\mathcal{F}$
$\mathcal{V}$	$\mathcal{V}$	$\mathcal{F}$

Es fácil ver que se puede usar a este operador lógico como a un conectivo fundacional dentro de la lógica, es decir, podemos a partir de éste definir a todos los conectivos clásicos, pues tenemos:

$$\neg x = x \downarrow x$$
$$x \vee y = (x \downarrow x) \downarrow (y \downarrow y)$$

Y al tener la disyunción y negación es fácil mostrar que se tienen las tres conectivas lógicas clásicas restantes.

Lema 3.1.1. *Bastan la negación y disyunción para describir en sus términos cualquier conectivo lógico binario.*

Demostración: Nótese que el número de conectivos binarios posibles para la lógica es de tantos como distintas combinaciones de $\mathcal{V}$ y $\mathcal{F}$ existan como resultado para una expresión de la forma $p\square q$, en donde $\square$ es nuestro conectivo binario. Así, hay 2^4 posibles conectivos binarios. Primero que nada, mostremos que podemos expresar todos los conectivos lógicos con los que contamos (es decir, la conjunción, implicación y doble implicación) en términos de la disyunción y negación:

$$\begin{aligned} p \wedge q &\equiv \neg(\neg p \vee \neg q) \\ p \Rightarrow q &\equiv \neg p \vee q \\ p \Leftrightarrow q &\equiv (p \Rightarrow q) \wedge (q \Rightarrow p) \\ &\equiv (\neg p \vee q) \wedge (\neg q \vee p) \\ &\equiv \neg(\neg(\neg p \vee q) \vee \neg(\neg q \vee p)) \end{aligned}$$

Ahora, bástenos mostrar que es posible formular expresiones únicamente en términos de disyunción y negación, tales que tengamos una expresión e_1 que sea $\mathcal{V}$ únicamente en el caso en que las dos variables son $\mathcal{F}$; una expresión e_2 que sea $\mathcal{V}$ únicamente en el caso en que la primera variable, p, sea $\mathcal{F}$ y la segunda, q, $\mathcal{V}$, etc. De forma que para cualquier conectivo binario $\square$, sea suficiente poner en disyunción a las expresiones e_1, e_2, e_3, e_4 necesarias para obtener la tabla de verdad que le represente.

Para la primera expresión, e_1, que resulta en $\mathcal{V}$ únicamente en el caso en que tanto p como q son $\mathcal{F}$, tenemos:

$$\neg(p \vee q)$$

Para e_2, caso que resulta en $\mathcal{V}$ únicamente cuando p es $\mathcal{F}$ y q es $\mathcal{V}$, tenemos:

$$\neg(p \vee \neg q)$$

Para e_3, caso que resulta en $\mathcal{V}$ únicamente cuando p es $\mathcal{V}$ y q es $\mathcal{F}$, tenemos:

$$\neg(\neg p \vee q)$$

Y finalmente, para e_4, caso que resulta en $\mathcal{V}$ únicamente cuando tanto p como q son $\mathcal{V}$, tenemos:

$$\neg(\neg p \vee \neg q)$$

De forma que cualquier conectivo binario es representable utilizando únicamente disyunción y negación. □

Este proceso de reducción que hacemos de la lógica, mediante el cual buscamos conseguir los bloques más fundamentales, suficientes y necesarios para la formulación de todos sus enunciados, nos lleva, desde el punto de vista de la axiomatización, no sólo a la revaloración (que bien pudiera ser superficial) de los conectivos, sino también, nos conduce a la examinación de los conceptos existentes en torno a la cuantificación de las variables que utilizamos en las expresiones de la lógica, a saber, tanto la cuantificación universal ($\forall$) como la existencial ($\exists$).

Similar a la reducción de los conectivos por medio del operador `NOR`, podemos introducir el siguiente enunciado de cuantificación como un conectivo fundamental:

$$\forall x(\neg F(x) \vee \neg G(x))$$

Al que denotaremos como:

$$F(x) \uparrow^x G(x)$$

Y a partir del cual podemos reducir todavía más nuestro arsenal de definiciones. Tomando a las constantes como predicados de un único argumento, tenemos que:

$$\begin{aligned}
\neg a &= a \uparrow^x a \\
a \vee b &= \forall x(a \vee b) \\
&= \neg a \uparrow^x \neg b \\
&= (a \uparrow^y a) \uparrow^x (b \uparrow^y b) \\
\forall x(F(x)) &= \forall x(\neg\neg F(x) \vee \neg\neg F(x)) \\
&= \neg F(x) \uparrow^x \neg F(x) \\
&= (F(x) \uparrow^y F(x)) \uparrow^x (F(x) \uparrow^y F(x))
\end{aligned}$$

En su exposición, Schönfinkel apunta que las variables cuantificadas dentro de la lógica no son más que simples fichas que caracterizan el que argumentos y operadores corresponden a una y la misma frase. De forma que

estas variables pueden ser vistas como una noción meramente auxiliar de la lógica, innecesaria cuando se contrasta con las nociones esenciales de esta y, así, pugna por removerlas. Para la remoción de las variables es necesario extender la noción general de las funciones, vistas como reglas de correspondencia entre dos elementos de un mismo dominio, permitiendo, al igual que en los trabajos de Church, que aparezcan tanto como argumentos como valores de las mismas funciones. Es a partir de los combinadores que se elabora una teoría de las funciones capaz de esto, como expondremos a continuación.

3.2. Combinadores

Los combinadores son, de forma similar a los λ-términos en el cálculo λ, los objetos de los cuales versa nuestra teoría, y pueden ser tanto definidos en términos de una operación de abstracción, como postulados como primitivos y utilizados para definir a otros.

Semejante al cálculo λ expuesto en el capítulo anterior, la lógica combinatoria permite no sólo la definición en términos propios del conjunto de los números naturales, sino también la construcción de cualquier función sobre los números naturales que sea recursiva general [34] y así, cualquier función efectivamente calculable puede ser definida únicamente por medio de combinadores, que hacen las veces de los elementos primigenios de nuestra teoría.

Definición 3.2.1. (Función recursiva general). Una función f de k argumentos es recursiva general o recursiva primitiva si:

1. f es una función constante $\mathbf{0}$ que para cualquier lista de argumentos que reciba, devuelve siempre 0.

2. f es la función sucesor $\mathcal{S}$ de aridad k, que para cada elemento i de sus argumentos, le suma uno.

3. f es la función proyección $\mathcal{P}_k^n$ que devuelve al k-ésimo argumento de su lista de argumentos.

4. f es la función de aridad n que resulta de la composición de dos funciones primitivas recursivas g y h, de aridad n y m, respectivamente, tal que:

$$f(x_1, \ldots, x_m) = g(h_1(x_1, \ldots, x_m), \ldots, h_n(x_1, \ldots, x_m))$$

5. f está definida por recursión en dos funciones recursivas primitivas f y h de aridad $n-1$ y $n+1$, respectivamente, de forma que:

$$f(x_1, \ldots, x_{n-1}, 0) = g(x_1, \ldots, x_{n-1})$$

$$f(x_1, \ldots, x_{n-1}, m+1) = h(x_1, \ldots, x_{n-1}, m, f(x_1, \ldots, x_{n-1}, m))$$

Definición 3.2.2. (Función calculable). Una función $f : \mathbb{N}^n \to \mathbb{N}$ es calculable si y sólo si existe un procedimiento efectivo tal que, dada cualquier eneada $\bar{a}$ de números naturales, nos da $f(\bar{a})$.

Los combinadores que se definen en el artículo de Schönfinkel, son los siguientes: $\boldsymbol{I}$ (función identidad o *Identitätsfunktion*), $\boldsymbol{C}$ (función constante o *Konstantzfunktion*), $\boldsymbol{T}$ (función de intercambio o *Vertauschungsfunktion*), $\boldsymbol{Z}$ (función de composición o *Zusammensetzungsfunktion*) y $\boldsymbol{S}$ (función de fusión o *Verschmelzungsfunktion*), mismos que aquí mostramos sólo para dar cuenta del origen de sus nombres. Por practicidad, en lugar de continuar con la exposición de los combinadores en los términos usados por la escuela de Göttingen, emplearemos las definiciones expuestas en *Combinatory Logic* [22], por el matemático estadounidense Haskell Curry (1900 - 1982) y el belga Robert Feys (1889 - 1961), mismas que posteriormente se popularizaron, así como su notación, que se perpetuó en el estudio de la teoría relacionada a esta área de la lógica matemática.

De manera similar a Schönfinkel, Curry y Feys parten de la definición de combinadores simples, los cuales expondremos a continuación en conjunto con las reglas de reducción asociadas a ellos. Estas reglas establecen la reducción que resulta de la aplicación de cada combinador a una serie finita de variables. Al igual que en el cálculo λ, la aplicación en la lógica combinatoria asocia por la izquierda.

Definición 3.2.3. (Identificador elemental). El más sencillo de los combinadores es aquel que nos permite expresar una variable como función de sí misma. Este combinador, que empata con la definición de la *Identitätsfunktion* de Schönfinkel, se denota, al igual que este, como $\boldsymbol{I}$, cuya regla de reducción es:

$$\boldsymbol{I}x \to x$$

Ejemplo 3.2.1.

$$\boldsymbol{II} \to \boldsymbol{I}$$
$$\boldsymbol{ISKK} \to \boldsymbol{SKK}$$
$$\boldsymbol{I}w \to w$$

Definición 3.2.4. (Permutador elemental). Dada una función f de dos argumentos, el permutador $\boldsymbol{C}f$, equivalente a la *Vertauschungsfunktion* dada

por Schönfinkel, devuelve el converso, es decir, la regla de reducción de este es:

$$\boldsymbol{C}fxy \to fyx$$

Ejemplo 3.2.2. Suponiendo que hemos definido ya la suma dentro de la lógica combinatoria y es esta denotada por $+$, la función dada por $\boldsymbol{C}+7$ Es aquella que recibe cualquier objeto y le suma 7:

$$\boldsymbol{C}+5(7) \to +5(7) \to 12$$

Definición 3.2.5. (Duplicador elemental). Dada una función f de dos argumentos, $\boldsymbol{W}f$ funciona como una función unaria que devuelve a la misma como una binaria aplicada al argumento que le fue pasado. Así, su regla de reducción es:

$$\boldsymbol{W}fx \to fxx$$

Ejemplo 3.2.3. Suponiendo que se cuenta con la definición de la multiplicación dentro de la lógica combinatoria, y que la hemos denotado por $*$, la función dada por $\boldsymbol{W}*$ es aquella que recibe a un número y lo eleva al cuadrado:

$$\boldsymbol{W}*(2) \to *(2)(2) \to 4$$

Definición 3.2.6. (Compositor elemental). El compositor elemental $\boldsymbol{B}$, representado en la notación de Göttingen por la *Zusammensetzungsfuktion*, reproduce la composición de dos funciones. Su regla de reducción es:

$$\boldsymbol{B}fgx \to f(gx)$$

Este combinador toma un significado distinto y muy útil al ser utilizado como un combinador unitario, en tanto que convierte a f, la única función que recibe como argumento, en una que es aplicable a funciones, de forma tal que si f fuera, por ejemplo, la negación lógica, Bf pasaría a ser un combinador que niega al predicado que recibe como argumento.

Ejemplo 3.2.4. Dado por hecho que contamos con una definición tanto para la función seno como la función coseno dentro de la lógica combinatoria, y que estas son denotadas por *sen* y *cos* respectivamente, tenemos que $\boldsymbol{B}sen(cos)$ es una función que compone a estas:

$$\boldsymbol{B}sen(cos)\frac{\pi}{2} \to sen(cos(\frac{\pi}{2})) \to sen(0) \to 0$$

Como comentamos ya, el combinador $\boldsymbol{B}$ tiene también relevancia al ser empleado como un combinador que recibe un único argumento, pues convierte la función con la que se encuentra en una que es aplicada al argumento que se le pase; así, $\boldsymbol{B}sen$ es un combinador que recibe una función de un argumento, misma cuyo resultado será utilizado como argumento de la función seno. Este proceso puede ser iterado, en tanto que

$$\boldsymbol{B}(\boldsymbol{B}f)gxy \to \boldsymbol{B}f(gx)y \to f(gxy)$$

Y así, $\boldsymbol{B}(\boldsymbol{B}f)$ es ahora un combinador que toma una función de dos argumentos a los que les aplica la función f, $\boldsymbol{B}(\boldsymbol{B}(\boldsymbol{B}f)g)$ es un combinador que toma una función de tres argumentos a los que les aplica la función f, etc.[1]

Definición 3.2.7. (Cancelador elemental). Este combinador, equivalente a la *Konstantzfunktion* de Schönfinkel, es útil para representar una constante c en términos de una función que recibe un argumento x, pues tiene como regla de reducción:

$$\boldsymbol{K}cx \to c$$

Ejemplo 3.2.5.

$$\boldsymbol{K}ab \to a$$
$$\boldsymbol{K}\boldsymbol{I}\boldsymbol{X} \to \boldsymbol{I}$$

Por último, definimos como parte de los combinadores básicos que aquí exponemos el combinador $\boldsymbol{S}$, equivalente a la *Verschmelzungsfunktion* de Schönfinkel.

Definición 3.2.8. (Substituidor). Supóngase que tanto $\mathcal{F}$ como $\mathcal{G}$ son dos objetos que dependen de un tercero, x. Si tenemos f y g tales que $fx \to \mathcal{F}$ y $gx \to \mathcal{G}$, podemos expresar a $\mathcal{F}\mathcal{G}$ en función de x. Si a tal función la nombramos $\boldsymbol{S}fg$, obtenemos la siguiente regla de reducción:

$$\boldsymbol{S}fgx \to fx(gx)$$

Ejemplo 3.2.6.

$$\boldsymbol{S}abc \to ac(bc)$$
$$\boldsymbol{S}\boldsymbol{K}\boldsymbol{K}x \to \boldsymbol{K}x(\boldsymbol{K}x)$$

[1] Esto se hace, en realidad, por medio de *currying*, pues la evaluación total se obtiene a partir de evaluar una secuencia de funciones, cada una de las cuales recibe un único argumento.

Es a través de los combinadores que podemos realizar el proyecto original de Schönfinkel: prescindir del uso de variables ligadas dentro de la lógica de predicados. A modo de ilustración, ejemplifiquemos la forma en la que el uso de los combinadores ayuda a lograr esto:

Ejemplo 3.2.7. Tomemos la propiedad conmutativa de la disyunción dentro de la lógica:

$$\forall x \forall y (x \vee y = y \vee x)$$

Como vimos ya, es posible reducir tanto a los conectivos lógicos como a los cuantificadores a un único conectivo ($\downarrow$) y a un único cuantificador ($\uparrow$):

$$(a \uparrow^y a) \uparrow^x (b \uparrow^y b) = (b \uparrow^y b) \uparrow^x (a \uparrow^y a)$$

Sin embargo, esto no nos libra de tener variables ligadas dentro de nuestras expresiones, como puede notarse en las variables que están siendo ligadas por cada cuantificador $\uparrow$. Lo que nos es lícito hacer es reescribir esta expresión para lograr una forma que se pueda, posteriormente, transformar en una que no haga uso de las variables x y y como variables ligadas:

$$\forall x \forall y (X(x, y) = x \vee y)$$

Ahora, introducimos un combinador $\boldsymbol{Co}$:

$$\forall f \forall x \forall y (\boldsymbol{Co}(f))(x, y) = f(x, y)$$

De forma que la propiedad inicial puede reescribirse ahora como

$$\boldsymbol{Co}X$$

Donde se prescinde del uso de cualquier variable ligada, y se tiene a un combinador ($\boldsymbol{Co}$) que representa la propiedad de conmutatividad para una operación X.

3.3. El cálculo $\boldsymbol{SK}$

Es posible utilizar los combinadores $\boldsymbol{K}$ y $\boldsymbol{S}$, definidos anteriormente, para establecer en sus términos los demás combinadores primitivos. Estos dos combinadores forman una *base*, es decir, a partir de ellos se pueden construir todos los λ-términos por medio de reducciones.

Teorema 3.3.1. $\{\boldsymbol{S}, \boldsymbol{K}\}$ *forma una base.*

Demostración: Dado un λ-término, elimínense todas las abstracciones a través de la aplicación repetida de las siguientes reglas:

$$\begin{aligned} \lambda x.x &\Rightarrow \boldsymbol{SKK} \\ \lambda x.A &\Rightarrow \boldsymbol{K}A \text{ (donde } x \notin FV(A)) \\ \lambda x.AB &\Rightarrow \boldsymbol{S}(\lambda x.A)(\lambda x.B) \end{aligned}$$

Estas reglas cubren todos los escenarios posibles, pues el cuerpo de las abstracciones es o una variable ligada (primer caso), u otro término (segundo caso); una aplicación (tercer caso) o una abstracción, la cual puede eliminarse de antemano por medio de estas mismas reglas. El proceso termina en tanto que sólo introduce abstracciones conformadas por un menor número de términos. □

Definición 3.3.1. (FV). Definimos dentro del contexto de la lógica combinatoria al conjunto de las variables libres de un combinador M, FV, como cualesquiera variables x en M.

Definición 3.3.2. (Reductos débiles). En el contexto de la lógica combinatoria, llamamos a los combinadores que conforman nuestra base *reductos débiles*. Un reducto débil es cualquier término $\boldsymbol{S}MNX$ (cuya reducción es $MX(NX)$) o $\boldsymbol{K}XY$ (cuya reducción es X). Denotamos como

$$X \rightarrow_w Y$$

a la transformación de un término X en uno Y por medio de una serie finita de aplicaciones de las reglas de reducción de estos combinadores.

El cancelador elemental, $\boldsymbol{K}$, elimina combinadores las veces necesarias para expresar a objetos $\mathcal{FG}$ en la forma $fx(gx)$, permitiéndonos usar la regla de reducción del subsitutidor $\boldsymbol{S}$, que añade un combinador extra y reacomoda a los existentes, de forma que nos permite llegar a la configuración necesaria para representar a los demás combinadores primitivos, como mostramos a continuación.

Para definir a $\boldsymbol{I}$ en términos del substituidor y el cancelador elemental, tenemos:

$$\begin{aligned} \boldsymbol{SK}gx &\rightarrow_w \boldsymbol{K}x(gx) \\ \boldsymbol{K}x(gx) &\rightarrow_w x \end{aligned}$$

De forma que $\boldsymbol{I}$ puede ser expresado como $\boldsymbol{SKX}$ donde $\boldsymbol{X}$ es un combinador arbitrario. Así:

$$\boldsymbol{I} \equiv \boldsymbol{SKS} \equiv \boldsymbol{SKK}$$

Es además trivial a partir de 3.3.1 y la representación en términos de $\boldsymbol{S}$ y $\boldsymbol{K}$ que el combinador $\boldsymbol{I}$ puede definirse como λ-término de la siguiente forma:

$$\boldsymbol{I} \equiv \lambda x.x$$

Es también una labor bastante sencilla la de seguir el sentido semántico de los combinadores primitivos para establecer sus representaciones en forma de λ-términos:

$$\begin{aligned}
\boldsymbol{S} &\equiv \lambda f.\lambda g.\lambda x.fx(gx) \\
\boldsymbol{K} &\equiv \lambda f.\lambda x.f \\
\boldsymbol{B} &\equiv \lambda f.\lambda g.\lambda x.f(gx) \\
\boldsymbol{C} &\equiv \lambda f.\lambda x.\lambda y.fyx \\
\boldsymbol{W} &\equiv \lambda f.\lambda x.fxx
\end{aligned}$$

Para definir al compositor elemental, $\boldsymbol{B}$, notemos que:

$$\begin{aligned}
\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K}f &\rightarrow_w \boldsymbol{KS}f(\boldsymbol{K}f) \\
\boldsymbol{KS}f(\boldsymbol{K}f) &\rightarrow_w \boldsymbol{S}(\boldsymbol{K}f) \\
\boldsymbol{S}(\boldsymbol{K}f) &\leftarrow \boldsymbol{B}f
\end{aligned}$$

De forma que[2]:

$$\boldsymbol{B} \equiv \boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K}$$

El duplicador elemental puede ser definido en términos de los combinadores $\boldsymbol{K}$ y $\boldsymbol{S}$ de la siguiente manera:

$$\boldsymbol{SS}(\boldsymbol{KI})fx \rightarrow_w \boldsymbol{S}f\boldsymbol{I}x$$

[2]Nótese que la última de las reducciones (i.e., la que muestra que $\boldsymbol{B}f$ se reduce en $\boldsymbol{S}(\boldsymbol{K}f)$) no es, como las demás, "$\rightarrow_w$". Esto se debe a un sencillo aspecto técnico, y es que, hasta entonces, no habíamos establecido la reescritura de $\boldsymbol{B}$ en términos de $\boldsymbol{S}$ y $\boldsymbol{K}$, únicos implicados en esta clase de reducción. El mismo razonamiento se emplea en las siguientes reducciones.

$$\begin{aligned} \boldsymbol{S}f\boldsymbol{I}x &\rightarrow_w fx(\boldsymbol{I}x) \\ fx(\boldsymbol{I}x) &\rightarrow_w fxx \end{aligned}$$

De donde podemos concluir lo siguiente:

$$\boldsymbol{W} \equiv \boldsymbol{SS}(\boldsymbol{KI})$$

Por último, para el permutador elemental, $\boldsymbol{C}$, notemos que:

$$\begin{aligned} fyx &\leftarrow_w fy(\boldsymbol{K}xy) \\ fy(\boldsymbol{K}xy) &\leftarrow_w \boldsymbol{S}f(\boldsymbol{K}x)y \end{aligned}$$

Con lo cual tenemos que $\boldsymbol{C}fx \rightarrow \boldsymbol{S}f(\boldsymbol{K}x)$ y:

$$\begin{aligned} \boldsymbol{C}f &\rightarrow \boldsymbol{B}(\boldsymbol{S}f)\boldsymbol{K}x \\ \boldsymbol{B}(\boldsymbol{S}f)\boldsymbol{K}x &\leftarrow_w \boldsymbol{BBS}f\boldsymbol{K} \\ \boldsymbol{BBS}f\boldsymbol{K} &\leftarrow_w \boldsymbol{BBS}f(\boldsymbol{KK}f) \\ \boldsymbol{BBS}f(\boldsymbol{KK}f) &\leftarrow_w \boldsymbol{S}(\boldsymbol{BBS})(\boldsymbol{KK})f \end{aligned}$$

De modo que:

$$\boldsymbol{C} \equiv \boldsymbol{S}(\boldsymbol{BBS})(\boldsymbol{KK})$$

Así, es posible representar a todos los combinadores primitivos que definimos en un principio con $\{\boldsymbol{K}, \boldsymbol{S}\}$ como base, de donde podemos concluir que estos dos combinadores son suficientes para formar cualquier expresión de la lógica combinatoria.

Esta base es una de las muchas que pueden tomarse, pues bien podríamos reducir nuestros combinadores a uno único, tal como muestra Fokker [35]:

Proposición 3.3.2. *Existe una base para la lógica combinatoria que consiste de un único elemento* $\boldsymbol{X}$.

Demostración: Definimos $\boldsymbol{X} \equiv \boldsymbol{KSK}$. Puede entonces verificarse que:

$$\begin{aligned} \boldsymbol{K} &= \boldsymbol{XX} \\ \boldsymbol{S} &= \boldsymbol{X}(\boldsymbol{XX}) \end{aligned}$$

Y teniendo estos dos, es posible construir cualquier expresión. □

Como se puede constatar, uno de los intereses típicos dentro de la lógica combinatoria es el de determinar las formas en las que es posible derivar a objetos a partir de otros con los que ya se cuenta. El sistema de combinadores que hemos descrito, hecho a partir de únicamente el cancelador elemental ($\boldsymbol{K}$) y el substituidor ($\boldsymbol{S}$), constituye un cálculo combinatorio que cumple con ser completo, es decir, es capaz de representar cualquier combinador que ponga en relación a un número n de variables.

Definición 3.3.3. (Cálculo combinatorio). Un cálculo combinatorio es dado por una colección $\mathcal{C}$ finita de operadores que son usados para definir a los $\mathcal{C}$-combinadores a través de la aplicación:

$$M, N ::= O|MN$$

Donde M, N son $\mathcal{C}$-combinadores arbitrarios y O una operación cualquiera.

Definición 3.3.4. (Completud combinatorial). Un conjunto de combinadores $\mathcal{C} = \mathcal{C}_1, \ldots, \mathcal{C}_n$ que son determinados por reglas de reducción se dicen funcional, o combinatorialmente completos, si para cualquier objeto X, que esté en combinación con distintas variables $c_1, \ldots, c_n$, existe un combinador U expresable en términos de $\mathcal{C}_1, \ldots, \mathcal{C}_n$ tal que

$$Ux_1 \ldots x_n = X$$

Las únicas reglas de reducción con las que contamos en el cálculo $\boldsymbol{SK}$ son las que hemos expuesto anteriormente para estos combinadores, a saber:

$$\boldsymbol{S}MNX \rightarrow_w MX(NX)$$
$$\boldsymbol{K}XY \rightarrow_w X$$

El substituidor duplica al último objeto, X como argumento tanto de M como de N, en tanto que el cancelador elemental elimina al segundo de los argumentos que recibe (Y) y devuelve únicamente el primero (X). Las definiciones que dimos de estos combinadores en términos del cálculo λ nos permiten mostrar que existe un homomorfismo del cálculo $\boldsymbol{SK}$ al cálculo λ, es decir, hay un mapeo de los combinadores de nuestro cálculo a λ-términos que preserva la aplicación y reducción. Las reglas de conversión que tenemos entre el cálculo $\boldsymbol{SK}$ y el cálculo λ son las siguientes:

$$\boldsymbol{S} = \lambda f.\lambda g.\lambda x.fx(gx)$$

$$\begin{aligned} \boldsymbol{K} &= \lambda x.\lambda y.x \\ MN &= (M)(N) \end{aligned}$$

Ejemplo 3.3.1. Como ejemplo, veamos que el combinador $\boldsymbol{SKK}$, el cual hemos afirmado es $\boldsymbol{I}$, resulta en la función identidad del cálculo λ ($\lambda x.x$):

$$\begin{aligned} \boldsymbol{SKK} &\equiv (\lambda g.\lambda f.\lambda x.gx(fx))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x) \\ &\to_\beta (\lambda f.\lambda x.(\lambda x.\lambda y.x)x(fx))(\lambda x.\lambda y.x) \\ &\to_\beta (\lambda x.(\lambda x.\lambda y.x)x([\lambda x.\lambda y.x]x) \\ &\to_\beta \lambda x.(\lambda y.x)([\lambda x.\lambda y.x]x) \\ &\to_\beta \lambda x.(\lambda y.x)(\lambda y.x) \\ &\to_\beta \lambda x.x \equiv \boldsymbol{I} \end{aligned}$$

La lógica combinatoria y, en este caso, el cálculo $\boldsymbol{SK}$, pueden, al igual que el cálculo λ, ser vistos como lenguajes de programación universales, en el sentido en que estos (al igual que el cálculo de Church) son capaces de expresar cualquier cómputo concebible. De esta forma, podríamos hacer un símil entre los términos de, digamos, las expresiones del cálculo $\boldsymbol{SK}$ con los módulos o funciones de un programa; en tanto que las expresiones completas serían vistas como programas: cada programa concebible en un lenguaje de programación (e.g., C, Python, Haskell, etc.) puede ser representado en términos del cálculo combinatorio.

Ejemplo 3.3.2. Como ejemplo de la afirmación que hemos hecho en el párrafo anterior, ilustremos el tratamiento de expresiones recursivas a través del combinador $\boldsymbol{Y}$, visto con anterioridad en el ámbito del cálculo λ, y expresado ahora en términos del cálculo $\boldsymbol{SK}$. Tomamos aquí la definición que de este combinador hace V. E. Wolfengangen [36]:

$$\begin{aligned} \boldsymbol{Y} &\equiv \boldsymbol{S}(\boldsymbol{BWB})(\boldsymbol{BWB}) \\ &\equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{KSS}(\boldsymbol{KI})\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K})(\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{KSS}(\boldsymbol{KI})\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K}) \\ &\equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{KSS}(\boldsymbol{KSKK})\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K})(\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{KSS}(\boldsymbol{KSKK})\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K}) \end{aligned}$$

Al igual que en el cálculo λ, dentro del cálculo $\boldsymbol{SK}$, el combinador $\boldsymbol{Y}$ es un combinador de punto fijo, lo que quiere decir que

$$\boldsymbol{Y}F = F(\boldsymbol{Y}F)$$

Y en efecto, aplicando este combinador $\boldsymbol{Y}$ a un objeto arbitrario X del cálculo, obtenemos:

$$\begin{aligned}\boldsymbol{Y}X &\equiv \boldsymbol{S}(\boldsymbol{BWB})(\boldsymbol{BWB})X \\ &\to_w \boldsymbol{W}(\boldsymbol{B}X)(\boldsymbol{BWB}X) \\ &\to_w \boldsymbol{B}X(\boldsymbol{BWB}X)(\boldsymbol{BWB}X) \\ &\to_w X(\boldsymbol{BWB}X(\boldsymbol{BWB}X)) \\ &\to_w X(\boldsymbol{S}(\boldsymbol{BWB})(\boldsymbol{BWB})X) \\ &\to_w X(\boldsymbol{Y}X)\end{aligned}$$

Ejemplo 3.3.3. De la misma forma en que se definieron los números naturales dentro del cálculo λ, es posible hacerlo en el cálculo $\boldsymbol{SK}$, estableciendo al $\bar{0}$ como $\boldsymbol{I}sz$, y definiendo a los demás en la siguiente forma:

Definición 3.3.5. (Números naturales).

$$\begin{aligned}\bar{1} &\equiv sz \\ \bar{2} &\equiv ssz \\ \bar{3} &\equiv sssz \\ &\vdots\end{aligned}$$

Nuevamente, todos los números naturales pueden ser vistos como expresiones donde un primer término s es aplicado un n número de veces a un segundo, z, resultando en la representación de n-ésimo número natural.

Definición 3.3.6. (Sucesor).

$$\mathcal{S} \equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K})$$

Como ejemplo, computando $\bar{1}$ y $\bar{2}$, resultan las siguientes reducciones:

$$\begin{aligned}\bar{1} \equiv \mathcal{S}\bar{0} &\equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K})\boldsymbol{I}sz \\ &\to_w (\boldsymbol{S}(\boldsymbol{KS})\boldsymbol{K}s)\boldsymbol{I}sz \\ &\to_w \boldsymbol{KS}s\boldsymbol{K}s\boldsymbol{I}sz \\ &\to_w \boldsymbol{SK}s\boldsymbol{I}sz \\ &\to_w \boldsymbol{KI}s\boldsymbol{I}sz \\ &\to_w \boldsymbol{II}sz\end{aligned}$$

$$\rightarrow_w sz$$

$$\begin{aligned}\bar{2} \equiv \mathcal{S}\bar{1} &\equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}\boldsymbol{S})\boldsymbol{K})sz \\ &\rightarrow_w (\boldsymbol{S}(\boldsymbol{K}\boldsymbol{S})\boldsymbol{K}z)sz \\ &\rightarrow_w \boldsymbol{K}\boldsymbol{S}z\boldsymbol{K}zsz \\ &\rightarrow_w \boldsymbol{S}\boldsymbol{K}zsz \\ &\rightarrow_w \boldsymbol{K}szsz \\ &\rightarrow_w ssz\end{aligned}$$

3.4. Relación con el cálculo λ

La noción de las funciones como objetos en sí, vacíos de argumentos, no fue estudiada ni distinguida de aquella que las veía únicamente en su relación con las variables o valores que pudieran fungir como sus argumentos sino hasta la década de 1930, con los trabajos de Alonzo Church. En estos, la λ liga a variables y hace un trabajo similar a aquel de los cuantificadores dentro de la lógica proposicional. La noción de las funciones expresables como λ-términos es equivalente al de aquellas que son computables, y en este sentido, muestra su más distinguida y evidente conexión con la lógica combinatoria, pues ambas son capaces de expresar en sus términos todo aquello que sea efectivamente computable [37].

La lógica combinatoria, desarrollada poco antes que el cálculo λ, lidia con combinadores que prescinden de variables y se ufana de lograr lo mismo que el cálculo λ con un lenguaje más sencillo que el establecido por Church, al no poseer noción alguna para la ligadura de variables y contar con un proceso de reducción de términos menos entramado, pues el proceso clásico de reducción en el cálculo λ, como vimos en el capítulo anterior, es mucho más complejo que el utilizado en la lógica combinatoria, que depende únicamente de completar los elementos necesarios para la aplicación de las reglas de reducción de sus combinadores.

De forma similar a lo ya hecho con la β-reducción del cálculo λ, formalizamos aquí la reducción débil de la lógica combinatoria, que mencionamos en la sección anterior y denotamos como $\rightarrow_w$.

Definición 3.4.1. (Reducción débil). Definimos de forma inductiva, dentro de la lógica combinatoria, a la relación $\rightarrow_w$:

1. $X \rightarrow_w X$
2. $\boldsymbol{S}MNX \rightarrow_w MX(NX)$
3. $\boldsymbol{K}XY \rightarrow_w X$
4. $X \rightarrow_w Y, Y \rightarrow_w Z \Rightarrow X \rightarrow_w Z$
5. $X \rightarrow_w Y \Rightarrow UX \rightarrow_w UY$
6. $X \rightarrow_w Y \Rightarrow XU \rightarrow_w YU$

A pesar de la disminución en las diversas dificultades que se suscitan por el renombramiento de variables dentro de las expresiones del cálculo λ a partir del uso de los índices de de Bruijn, estas se mantienen latentes, pues los índices dan cuenta únicamente del nivel en el que se encuentra la λ con la que un término es ligado, es decir, que le abstrae, mas no eliminan el uso de la abstracción, como sí logra la lógica combinatoria.

Es este concepto de la abstracción dentro del cálculo λ uno de los de mayor importancia en toda su teoría, dado que es necesario para la construcción de operaciones; en tanto que en la lógica combinatoria los términos abstraibles se construyen a partir de combinadores atómicos (como lo son $\boldsymbol{S}$, $\boldsymbol{K}$, $\boldsymbol{I}$, etc...). Esta diferencia es la que conduce a las distintas definiciones en torno a la reducción en ambos sistemas. Dentro del cálculo λ, la β-reducción ($\rightarrow_\beta$) se lleva a cabo como el reemplazo de términos de la forma $(\lambda x.M)N$, llamados reductos; en tanto que en la lógica combinatoria, la relación de reducción está dada por el reemplazo de ciertos reductos asociados a los combinadores, como se muestra en las reglas de reducción que para cada combinador se establecieron.

A pesar de las diferencias existentes entre los mecanismos de reducción de ambas teorías, es posible establecer algoritmos que permitan la conversión de los términos de la lógica combinatoria al cálculo λ, como se establece ya en las definiciones dadas de cada uno de los combinadores expuestos. De forma semejante, es posible hacerlo en la otra dirección.

Como ya se ha visto dentro de este capítulo, es posible escoger distintas bases para la lógica combinatoria y a partir de éstas establecer diversas formas en las que se han de transformar sus expresiones a λ-términos (como la conversión que presentamos para el cálculo $\boldsymbol{SK}$), así como pasar de los λ-términos a los combinadores utilizados en las distintas bases.

El algoritmo que presentamos a continuación, para la conversión de los λ-términos a expresiones de la lógica combinatoria está formulado en el entorno clásico de esta, y hace uso de los combinadores que en un principio fueron expuestos:

- $x \to x$.
- $MN \to MN$.
- $\lambda x.M \to \boldsymbol{K}M$, si x no ocurre libre en M.
- $\lambda x.x \to \boldsymbol{I}$.
- $\lambda x.Mx \to M$, si x no ocurre libre en M.
- $\lambda x.MN \to \boldsymbol{B}M(\lambda x.N)$, si x no ocurre libre en M.
- $\lambda x.MN \to \boldsymbol{C}(\lambda x.M)N$, si x no ocurre libre en N.
- $\lambda x.MN \to \boldsymbol{S}MN$, si x no ocurre libre ni en M ni en N.

Y resumimos en la siguiente definición dada por Çağman y Hindley [37]:

Definición 3.4.2. (H-mapeo). Para cada λ-término M, se asigna un término de la lógica combinatoria, M_H, como sigue:

- $x_H \equiv x$
- $(MN)_H \equiv \lambda^* x.(M_H)$

Donde $\lambda^* x.X$ está definido por inducción en X para todos los términos M de la lógica combinatoria a partir del siguiente algoritmo:

a) $\lambda^* x.X \equiv \boldsymbol{K}X$ si $x \notin FV(X)$.

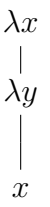
b) $\lambda^* x.x \equiv \boldsymbol{I}$.

c) $\lambda^* x.Ux \equiv U$ si $x \notin FV(U)$.

d) $\lambda^* x.UV \equiv \boldsymbol{S}(\lambda^* x.U)(\lambda^* x.V)$ si no aplica ninguna de las anteriores.

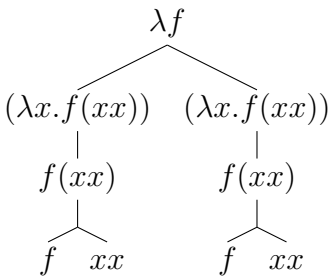
De expresar al λ-término como un árbol, la conversión se hace a partir de los nodos más bajos, estableciendo así el orden en el cual debe ser aplicado el anterior algoritmo.

Ejemplo 3.4.1.



$\boldsymbol{K}$ o $\lambda x.\lambda y.x$.

Ejemplo 3.4.2.



Definición del combinador de punto fijo, $\boldsymbol{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ del cálculo λ.

Mucho se ha discutido en la literatura respecto a las diferencias entre los diversos mecanismos definibles de reducción en los múltiples modelos de la computación que existen (entre los que contamos tanto a la lógica combinatoria como al cálculo λ). Como Hindley sugiere [38], la principal diferencia entre la reducción débil y la β-reducción estriba en la sutil diferencia que hay entre substitución y reemplazo: la *substitución* $[N/x]M$ es el resultado de substituir N por todas las ocurrencias libres de x en M, así como el renombramiento por medio de la α-conversión para no incurrir en ningún conflicto en el nombramiento de las variables; en tanto que el *reemplazo*, si bien cambia las ocurrencias libres de x en M por N, al igual que la substitución, no se encarga después del renombramiento de variables. Esto puede ser ilustrado por un ejemplo dado por el mismo Hindley, en conjunto con Çağman:

Ejemplo 3.4.3. Considérense las expresiones:

$$\begin{aligned} P &\equiv \lambda y.R \\ R &\equiv (\lambda x.xy)z \\ Q &\equiv \lambda y.zy \end{aligned}$$

Así, tenemos:

$$P \equiv \lambda y.((\lambda x.xy)z) \rightarrow_\beta \lambda y.zy \equiv Q$$

Si intentamos traducir las mismas expresiones a la lógica combinatoria por medio del H-mapeo para posteriormente reducir la primera como hicimos dentro del cálculo λ, obtendremos lo siguiente:

$$\begin{aligned} P_H &\equiv \lambda^* y.R_H \\ &\equiv \boldsymbol{S}(S(K(SI))K)(Kz) \\ R_H &\equiv (\lambda^* x.xy)z \end{aligned}$$

$$\begin{aligned} &\equiv \boldsymbol{S}I(Ky)z \\ Q_H &\equiv \lambda^* y.zy \\ &\equiv z \end{aligned}$$

Donde P_H no puede reducirse a Q_H, pues no contiene ningún reducto débil. El problema es, realmente, que P contiene una abstracción que posteriormente aparece como variable libre dentro de R (y). Esto se soluciona de manera simple, cambiando el nombre de esta, por ejemplo, teniendo $P \equiv \lambda w.R$, en cuyo caso el resultado sería:

$$P \equiv \lambda w.((\lambda x.xy)z) \to_\beta \lambda w.zy$$
$$P_H \equiv \boldsymbol{K}R_H \equiv \boldsymbol{K}(SI(Ky)z) \to_w \boldsymbol{K}(zy) \equiv \lambda^* w.zy$$

3.5. Programas en el cálculo $\boldsymbol{SK}$

Reiteradamente hemos mencionado la posibilidad de representar programas en forma de expresiones del cálculo $\boldsymbol{SK}$, tal como en su momento hicimos y ejemplificamos dentro del cálculo λ, ilustrando la forma en la que es posible en el cálculo de Church definir diversas instrucciones y operadores comunes en forma de λ-términos.

En los diversos ejemplos expuestos de funciones recursivas dentro del cálculo λ, utilizadas para ejemplificar la concepción del cálculo como un lenguaje de programación y sus expresiones como programas, se tuvo cuidado en el proceso de reducción de estas, aplicando siempre el orden normal, pues una mirada detenida en ellas deja ver que el orden aplicativo no es capaz de conducir a forma normal alguna; en tanto que ninguna de las expresiones construidas por medio de la fórmula $\boldsymbol{Y}F = F(\boldsymbol{Y}F)$ es fuertemente normalizable y, así, tales expresiones pueden incurrir en una recursión infinita dentro del cálculo λ.

En este sentido, el panorama de los programas vistos como expresiones cambia en gran manera dentro del cálculo $\boldsymbol{SK}$, mientras que la aplicación de los combinadores no se reduce hasta recibir a todos los argumentos necesarios, lo que impone una estrategia de evaluación interna al cálculo y no una mera convención externa al sistema.

Al agregar variables $x, y, z, \ldots$ al cálculo $\boldsymbol{SK}$, así como al definir términos de la forma $\lambda^* x.X$, expuestos ya en la definición del H-mapeo, es posible

introducir *términos combinatoriales*, mismos que permiten hacer uso del concepto de abstracción tal y como aparece en el cálculo λ, pues su definición misma representa a la λ-abstracción en tanto que mantiene a la β-reducción:

$$(\lambda^* x.M)N \to \{N/x\}M$$

Lo que a su vez permite replicar la estructura del combinador de punto fijo $\boldsymbol{Y}$ dentro del cálculo $\boldsymbol{SK}$ a través de una definición más familiar que la dada anteriormente por Wolfengagen.

Al igual que en el cálculo λ, podemos definir a los puntos fijos a través del término combinatorio

$$\omega = \lambda^* x.\lambda^* f.f(xxf)$$

Teniendo que $\boldsymbol{Y} = \omega\omega$ llegamos a la propiedad del punto fijo vista antes en el ámbito del cálculo λ, pues

$$\boldsymbol{Y}f = \lambda^* x.\lambda^* f.f(xxf))\omega \to f(\omega\omega f) = f(\boldsymbol{Y}f)$$

Podemos no sólo replicar el combinador de punto fijo con una construcción básicamente igual a la hecha en el cálculo λ, sino que, como muestra Barry Jay [39], es posible retrasar la reducción de los términos, de forma que se puedan definir funciones recursivas de n argumentos en forma normal.

Definición 3.5.1. (Combinadores de punto fijo para n argumentos). Es posible definir una función recursiva de forma normal que necesite de n argumentos a través de las definiciones de A_n, $\boldsymbol{Y}_n$ y ω_n:

$$\begin{aligned} A_n &= \lambda^* f.\lambda^* x_1, \ldots \lambda^* x_n.fx_1 \ldots x_n \\ \omega_n &= \lambda^* z.\lambda^* f.\lambda^* x_1, \ldots \lambda^* x_{n-1}.f(A_{n+1}zzf)x_1 \ldots x_{n-1} \\ \boldsymbol{Y}_n &= A_{n+1}\omega_n\omega_n \end{aligned}$$

De esta forma, cualquier programa recursivo que haga uso de n variables y represente el cómputo de un conjunto de instrucciones M puede representarse en forma normal dentro del cálculo $\boldsymbol{SK}$ como

$$\boldsymbol{Y}_{n+1}(\lambda^* f.\lambda^* x_1 \ldots \lambda^* x_n.M)$$

Capítulo 4

Cálculo $\boldsymbol{SF}$

"But the principium individuationis ... was to me, at all times, a consideration of intense interest."

Edgar Allan Poe, [40]

4.1. Motivación

La lógica combinatoria, tal y como se establece en su exposición clásica, es capaz de representar todas las funciones descritas por el cálculo λ (y, así, todas las funciones que son Turing-computables sobre los números naturales), como puede fácilmente advertirse a partir de las equivalencias establecidas en las definiciones de los combinadores elementales que hemos dado en el anterior capítulo, y como puede demostrarse a partir del H-mapeo definido anteriormente (3.4.2).

A pesar del enorme poder computacional que engloban estos dos sistemas de reescritura (constatable en los ejemplos hasta ahora expuestos), existen funciones efectivamente computables capaces de distinguir a los combinadores $\boldsymbol{SKK}$ y $\boldsymbol{SKS}$ (que, como ya vimos, son equivalentes al identificador elemental $\boldsymbol{I}$) que no pueden ser representadas por medio de ningún $\boldsymbol{SK}$-combinador (y, por consiguiente, tampoco por ningún λ-término) [9]; de existir tal combinador, este implicaría la capacidad de efectuar, dentro de la lógica combinatoria, un procedimiento de factorización de combinadores para dar cuenta de los objetos que los constituyen, proceso que no es realizable dentro del cálculo $\boldsymbol{SK}$ ni de ningún otro cálculo que tome como base los combinadores primitivos clásicos de la lógica combinatoria.

Que existan, tal y como afirman Barry Jay y Thomas Given-Wilson, funciones efectivamente computables capaces de distinguir a estos combinadores y que no sean representables por medio de ningún ***SK***-combinador [9] implican la existencia de funciones computables que son a su vez funciones intensionales (es decir, estas pueden dar cuenta de la forma en la que el cómputo se realiza) no representables ni en la lógica combinatoria ni en el cálculo λ. La dificultad que impone la no representabilidad de estas funciones dentro de los sistemas de reescritura de Schönfinkel y de Church estriba en la imposibilidad de efectuar el proceso metamatemático que implica la factorización de objetos que en la teoría son vistos como "totales" en sus partes constituyentes.

Definición 4.1.1. (Igualdad extensional). Dos funciones f y g se dicen extensionalmente iguales si tienen el mismo rango de argumentos y, para cualquier elemento α perteneciente a su rango, $f\alpha$ es igual que $g\alpha$.

Definición 4.1.2. (Igualdad intensional). Dos funciones f y g se dicen intensionalmente iguales si son extensionalmente iguales e idénticas en la forma en la que producen sus valores.

Tanto el cálculo λ como la lógica combinatoria proveen de herramientas suficientes para dar cuenta del caracter extensional de las funciones en el entorno más general posible; en este sentido, podemos decir que son teorías extensionales; su carencia de herramientas para un análisis intensional impide establecer cualquier solución a problemas de índole similar al recién expuesto, así como utilizarlos como base en la implementación de lenguajes de programación que den cuenta de la computación más allá del énfasis que ya tienen en los aspectos funcionales de esta.

A lo largo de este capítulo ilustraremos el tratamiento y distinción de la igualdad intensional y extensional a partir de las funciones ***SKK*** y ***SKS***, motivándonos a usarlas no otra cosa que su fácil manipulación, así como nuestra familiaridad con ellas y su significado, al ser ambas representaciones de la función identidad.

Ejemplo 4.1.1. Apartándonos de un ejemplo tan trivial como puede ser la función identidad, sirva para mostrar la importancia de la distinción entre estas formas de igualdad dentro de la computación el siguiente ejemplo, rescatando un problema de la final mundial de la ACM ICPC del año 2015:

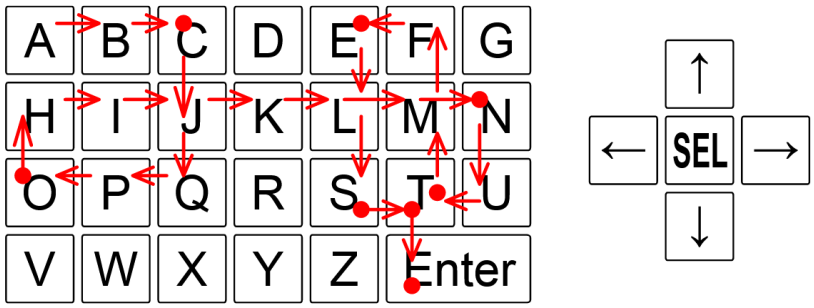
> ¿Cuántos tecleos son necesarios para escribir un mensaje de texto? A simple vista, podrá pensarse que los mismos que los caracteres en el texto, sin embargo esto sólo es correcto cuando las

> teclas generan únicamente un caracter. Distintos dispositivos proveen de sólo unos cuantos botones que, normalmente, tienen un número singificantemente más bajo de teclas que de caracteres tiene el alfabeto. Para tales dispositivos, se requiere de un número mucho mayor de tecleos para producir un mismo mensaje que, digamos, en una máquina de escribir. Considérese que para todo teclado virtual, se desea obtener el número mínimo de tecleos necesarios para escribir una cadena de texto dada si se inicia en la tecla superior izqueirda y las únicas formas de desplazarse a través de las teclas es por medio de cuatro teclas mecánicas que sirven para ello, y la única forma de seleccionar una tecla es a partir de una tecla mecánica "`SEL`".

La entrada que se da para el problema inicia por dos números enteros r y c, que representan el número de filas y columnas de la matriz que representa al teclado virtual, seguido de r líneas con c caracteres cada una, donde el asterisco representa "enter". Después de las r líneas, sigue una última que contiene una palabra para la cual deberá decirse el mínimo número de tecleos necesarios para escribirla. La entrada ejemplo para el problema es:

```
4 7
ABCDEFG
HIJKLMN
OPQRSTU
VWXYZ**
CONTEST
```

Figura 4.1: Movimientos que ejemplifican la solución a la entrada ejemplo, la cual es 30. Reproducción del problemario de la final mundial del ACM ICPC 2015.



Este problema se puede resolver de varias formas, una de ellas es la siguiente:

Considérese la gráfica dirigida $G = (V, E)$, donde los nodos V codifican la posición en el teclado de un caracter $(i, j), 1 \leq i \leq r, 1 \leq j \leq c$ y su posición k en el texto a escribir, $1 \leq k \leq l + 1$, donde l es la longitud de la cadena. Para cada nodo $v \in V$, este tiene aristas $e \in E$ de la forma (i', j', k), donde (i', j') es la posición en el teclado virtual a la que se llega en alguna de las direcciones a las que le es posible moverse. Además, para cada v, se agrega una arista a $(i, j, k + 1)$ (i.e., toda tecla que represente a la siguiente que debe ser teclada). Modelando el problema de esta forma, es posible minimizar el número necesario de tecleos para escribir la cadena deseada, empezando en $(1, 1, 1)$ y realizando una búsqueda a lo ancho sobre los caminos que terminen en $(i, j, l + 1)$. De esta manera, la complejidad de nuestro algoritmo es $\mathcal{O}(|V| + |E|)$, donde el número de nodos $|V|$ es $r * c * l$ y $|E|$ está acotada por $5|V|$.

Otra posible solución es a partir del uso repetido del algoritmo de Dijkstra para el camino más corto en una gráfica, haciendo nuevamente la búsqueda sobre los caminos que van de $(1, 1, 1)$ hasta $(i, j, l + 1)$, resultando en una complejidad de $\mathcal{O}(lr \log r)$.

Muchas estrategias podrían idearse para solucionar el problema, y, si bien la naturaleza extensional de cada solución es la misma en el sentido en que para el espacio de problemas estos algoritmos mapean a los mismos números que son la solución de los diversos escenarios, y el análisis de complejidad nos lleva a notar o al menos intuir alguna diferencia intensional en los mismos a partir de su eficiencia, no existe forma en la que, utilizando los mismos modelos computacionales que usemos para formalizar los distintos algoritmos, podamos dar cuenta de su diferencia intensional, y así analizar las partes relevantes en su construcción.

4.2. Extensionalidad en la lógica combinatoria

Antes de mostrar la manera en la que nos es posible extender a la lógica combinatoria para dar cuenta de la estructura interna de sus objetos, definamos en forma breve y concisa el alfabeto, términos y axiomas de esta:

Definición 4.2.1. ($\mathcal{CL}$). La teoría ecuacional de la lógica combinatoria es formulada a partir del alfabeto dado por los siguientes elementos:

1. Combinadores o constantes: $\boldsymbol{K}, \boldsymbol{S}$.

2. Variables: $x_0, x_1, \ldots$.

3. Relación de igualdad: $=$.

4. Símbolos impropios: $(,), [,]$.

A través de los cuales es posible expresar el conjunto de $\mathcal{CL}$-términos, mismo que es dado de forma inductiva por:

1. $x \in \mathcal{CL}$ para cualquier variable x.

2. $\boldsymbol{K} \in \mathcal{CL}, \boldsymbol{S} \in \mathcal{CL}$.

3. $P, Q \in \mathcal{CL} \Rightarrow (PQ) \in \mathcal{CL}$.

Definición 4.2.2. (Axiomas y reglas). Los axiomas de $\mathcal{CL}$ son los siguientes:

1. Esquema de axioma $\boldsymbol{K}$: $\boldsymbol{K}MN = M$.

2. Esquema de axioma $\boldsymbol{S}$: $\boldsymbol{S}MNX = MX(NX)$.

3. Axioma de reflexividad: $P = P$.

4. Regla de simetría: $P = Q \Rightarrow Q = P$.

5. Regla de transitividad $P = Q, Q = R \Rightarrow P = R$.

6. Primera regla de Leibniz: $P = P' \Rightarrow PR = P'R$.

7. Segunda regla de Leibniz: $P = P' \Rightarrow RP = RP'$.

Tal y como se esboza el sistema de deducción de $\mathcal{CL}$, no es posible dar cuenta siquiera de la igualdad entre los combinadores $\boldsymbol{SKK}$ y $\boldsymbol{SKS}$ expuestos en 4.1, mucho menos de su diferencia. El problema estriba en que mostrar la equivalencia de estos dos combinadores en un sentido extensional implica probar que ambos actúan de la misma forma cuando les es aplicado un argumento M cualquiera, lo que conduce rápidamente a la labor de demostrar que, para todo combinador M que pueda hacer las veces de X en el esquema de axioma $\boldsymbol{S}$, se cumple la igualdad, labor imposible en tanto que hay un número infinito de estos.

Así, al intentar demostrar $\mathcal{CL} \vdash \boldsymbol{SKK} = \boldsymbol{SKS}$ a partir de los axiomas y reglas establecidos, es necesario introducir un combinador para efectuar las reducciones implicadas en los esquema de axioma $\boldsymbol{K}$ y $\boldsymbol{S}$, pues tanto $\boldsymbol{SKK}$ como $\boldsymbol{SKS}$ son combinadores parcialmente aplicados, lo que nos conduce a una prueba de la deducibilidad de

$$\boldsymbol{SKK}X = \boldsymbol{SKS}X$$

para cada combinador X, mas no de la igualdad de la que partimos y que buscamos probar.

De esta forma, para mostrar la equivalencia entre estos combinadores, es necesario expandir $\mathcal{CL}$ de manera que pueda dar cuenta de la equivalencia extensional entre los objetos de la teoría.

Definición 4.2.3. (Combinador parcialmente aplicado). La aridad de cada combinador M está dada por el mínimo número de argumentos necesarios para instanciar su regla de reducción. Así, $\boldsymbol{K}$ tiene aridad dos, en tanto que $\boldsymbol{S}$ tiene aridad tres. Un combinador parcialmente aplicado se define como aquél de la forma

$$MX_1 \dots X_k,$$

donde k es menor que la aridad de M. A los combinadores que cumplen con ser parcialmente aplicados llamámosles también "compuestos", en caso contrario, "átomos".

Definición 4.2.4. (Regla de inferencia extensional). Si una variable x no forma parte de los $\mathcal{CL}$-términos M ni N y se cumple que $\mathcal{CL} \vdash Mx = Nx$, entonces $\mathcal{CL} \vdash M = N$.

Con lo que es posible demostrar la equivalencia extensional de los combinadores mencionados anteriormente:

Proposición 4.2.1. $\boldsymbol{SKK} = \boldsymbol{SKS}$.

Demostración: Tenemos que

$$\begin{aligned} \boldsymbol{SKK}x &= \boldsymbol{K}x(\boldsymbol{K}x) \\ \boldsymbol{K}x(\boldsymbol{K}x) &= x, \end{aligned}$$

similarmente,

$$\begin{aligned} \boldsymbol{SKS}x &= \boldsymbol{K}x(\boldsymbol{K}x) \\ \boldsymbol{K}x(\boldsymbol{K}x) &= x, \end{aligned}$$

por regla de transitividad, se cumple tanto $\boldsymbol{SKK}x = x$ como $\boldsymbol{SKS}x = x$. Por regla de simetría, así como nuevamente la regla de transitividad, tenemos que $\boldsymbol{SKK}x = x$ como que $x = \boldsymbol{SKS}x$, de donde, por regla de inferencia extensional, $\boldsymbol{SKK} = \boldsymbol{SKS}$. $\square$

Para los fines que buscamos, es menester introducir ahora algunas definiciones que nos facilitarán el tratamiento y análisis de los combinadores dentro del cálculo combinatorio como programas.

Definición 4.2.5. (Forma normal). En el contexto de la lógica combinatoria, un $\mathcal{CL}$-término M se dice en forma normal si y sólo si existe un λ-término N tal que este sea $\mathcal{R}$-fn y $M_H \equiv N$.

Es fácil notar que los $\mathcal{CL}$-términos en forma normal son irreductibles bajo la reducción débil de $\mathcal{CL}$ en tanto que son equivalentes a λ-términos, que son irreductibles por medio de β-reducción.

4.3. Factorización

Tanto el cálculo λ como la lógica combinatoria puros son capaces de representar las diversas estructuras de datos, ya sea a través de λ-términos o de combinadores. Si bien la representabilidad de estas es asequible, lo es en una forma ambigua y *ad hoc*, como puede notarse en los ejemplos dados hasta ahora, donde el mismo λ-término puede ser usado tanto para representar el número cero (2.4.1) como para el valor booleano de falsedad (2.3.2): $\lambda s.\lambda z.z$. Es fácil escapar de la ambigüedad en la construcción de estructuras introduciendo tipos de datos algebraicos, de manera que la representación de números, listas, cadenas, etc., pueda darse en forma de árboles que son construidos a partir de constructores básicos.

Ejemplo 4.3.1.

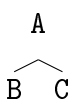
```
class A { };
class B: public A { };
class C: public A { };
```

La forma canónica para aproximarse a los tipos algebraicos dentro de los lenguajes orientados a objetos es a través del concepto de herencia de clases.

Ejemplo 4.3.2.

```
data A = B | C | None
```

En lenguajes puramente funcionales, como Haskell, los tipos de datos algebraicos se muestran de forma más natural, poniendo dentro de la misma definición de un tipo de dato las formas en las que puede darse.



En ambos ejemplos, los tipos `B` y `C` derivan de uno `A`.

Sin embargo, ha de notarse que, aunque la ambigüedad en la representabilidad de las estructuras de datos sea resuelta a través de los tipos de datos algebraicos, el acercamiento sigue siendo *ad hoc* y no alcanza la generalidad, en tanto que es necesario dar cuenta de la cadena de tipos o clases de los cuales se compone la estructura para poder acceder a sus datos, problema básicamente inexistente en paradigmas distintos al funcional u orientado a objetos, como los lenguajes de consultas, donde no es necesario un conocimiento completo de los tipos o esquemas para obtener los valores de las consultas (*queries*) que se hacen a los distintos campos particulares dentro de las bases de datos.

La uniformidad en el acceso a los datos internos de las estructuras puede alcanzarse viendo las propiedades que son comunes a todo árbol y, así, conceptualizar a cada elemento como un nodo. Utilizando la terminología que hemos introducido anteriormente, cada estructura de datos expresable es o bien un átomo, o un compuesto. Al representar las estructuras de datos como árboles, la naturaleza de los datos pasa a ser irrelevante en tanto que las consultas son hechas utilizando estrategias generales para recorrer árboles, donde la única diferencia relevante que es posible hallar en cada nodo es si este es un átomo o un compuesto.

La capacidad de definir diversos programas como formas normales dentro de la lógica combinatoria lleva implícito que estos adopten una naturaleza ambivalente, en tanto que pueden ser vistos como expresiones capaces de ser aplicadas a otras o efectuar las reducciones marcadas por las reglas de los combinadores que forman parte de sí, como, al ser todo programa una estructura de datos, pueden también verse como árboles sintácticos conformados por nodos que contienen a los combinadores primivitos.

Todos los combinadores irreductibles que puedan construirse a partir de la combinación de los combinadores primitivos y que resulten en forma normal, tales como $\boldsymbol{SKK}$ y $\boldsymbol{SKS}$, califican como distintos programas que, como ya vimos, no es posible distinguir a partir de la inferencia extensional, en tanto que esta da cuenta únicamente de los resultados de sus cómputos y, extensionalmente, representan a la misma función identidad, pues la aplicación de un combinador arbitrario M a cualquiera de ellos, se reduce siempre a M. Así, surge de manera natural la pregunta, ¿de qué forma podemos separarlos y hacer visible esta distinción? Damos respuesta a la pregunta a través del cálculo $\boldsymbol{SF}$, que logra, a partir de la introducción de un combinador $\boldsymbol{F}$, dar cuenta de la configuración de sus términos a partir de la factorización de los mismos.

En el cálculo $\boldsymbol{SF}$, los programas se construyen a partir de los dos com-

binadores atómicos $\boldsymbol{S}$ y $\boldsymbol{F}$, que permiten la factorización de la estructura interna de todo combinador del cálculo. El combinador $\boldsymbol{S}$ es tomado de la lógica combinatoria tal cual, mientras que el factorizador [1], $\boldsymbol{F}$, rompe a los compuestos en sus componentes y a los átomos los mantiene en su misma forma irreductible. Las reglas de reducción del combinador $\boldsymbol{F}$ son las siguientes:

Definición 4.3.1. (Reglas de reducción para $\boldsymbol{F}$).

$$\begin{aligned} \boldsymbol{F}PMN &\to M \text{ si } P \text{ es } \boldsymbol{S} \text{ o } \boldsymbol{F}, \\ \boldsymbol{F}(PQ)MN &\to NPQ \text{ si } PQ \text{ es una forma factorizable.} \end{aligned}$$

Donde las formas factorizables del cálculo son todos los combinadores de las formas $\boldsymbol{S}, \boldsymbol{S}M, \boldsymbol{S}MN, \boldsymbol{F}, \boldsymbol{F}M$ y $\boldsymbol{F}MN$ para cualesquiera combinadores M y N.

4.4. El cálculo $\boldsymbol{SF}$

Además de permitir la identificación de expresiones dentro suyo con programas, cosa realizable ya dentro de otras formas de la lógica combinatoria, permitiendo que los programas, representados por $\mathcal{CL}$-términos, sean aplicados, tal y como los λ-términos en el cálculo λ o los combinadores de cualquier lógica combinatoria, sobre otros programas o sobre sí, el cálculo $\boldsymbol{SF}$ logra examinar las partes que constituyen a estos, y ya no únicamente verlos a través de la lente de los resultados en los que deriva la maquinaria de reescritura.

Antes de continuar con la introducción que hacemos al cálculo $\boldsymbol{SF}$, introducimos los axiomas ecuacionales de este, mismos que complementan a los expuestos para $\mathcal{CL}$ (prescindiendo, claro está, del esquema de axioma para $\boldsymbol{K}$):

Definición 4.4.1. (Axiomas ecuacionales del cálculo $\boldsymbol{SF}$).

1. $\boldsymbol{S}MNP = MP(NP)$

[1] $\boldsymbol{F}$ no puede ser tomado como un combinador si nos apegamos al sentido que tiene el término dentro de Curry y Feys (cap. 5, [22]), en tanto que es imposible expresarlo a través de λ-términos de la forma $\lambda.x_1 \ldots x_n.X$; como es lógico, en tanto que el cálculo λ es una teoría extensional. Nos limitamos a esta única nota y lo tildamos como combinador a lo largo del texto, justificando la decisión a partir del caracter primitivo que tiene dentro del cálculo.

2. $\boldsymbol{FS}MN = M$
3. $\boldsymbol{FF}MN = M$
4. $\boldsymbol{F}(\boldsymbol{S}P)MN = N\boldsymbol{S}P$
5. $\boldsymbol{F}(\boldsymbol{F}P)MN = N\boldsymbol{F}P$
6. $\boldsymbol{F}(\boldsymbol{S}PQ)MN = N(\boldsymbol{S}P)Q$
7. $\boldsymbol{F}(\boldsymbol{F}PQ)MN = N(\boldsymbol{F}P)Q$

En tanto que el cálculo $\boldsymbol{SK}$ es un subsistema propio del cálculo $\boldsymbol{SF}$, toda función representable en el primero puede también ser representada en el último, así como se puede dar parte de la igualdad extensional de todas las expresiones a partir de la inferencia extensional. La demostración es elemental, pues basta mostrar que $\boldsymbol{K}$ es representable por medio de los combinadores $\boldsymbol{S}$ y $\boldsymbol{F}$ del cálculo $\boldsymbol{SF}$.

Proposición 4.4.1. *Todos los $\mathcal{CL}$-términos expresables del cálculo $\boldsymbol{SK}$ lo son también en el cálculo $\boldsymbol{SF}$.*

Demostración: Dado que $\boldsymbol{K}$ es representable dentro del cálculo $\boldsymbol{SF}$ como $\boldsymbol{FF}$, podemos dar cuenta de todos los combinadores existentes en el cálculo $\boldsymbol{SK}$, dentro del cálculo $\boldsymbol{SF}$:

$$\boldsymbol{FF}xy = x \tag{4.1}$$
$$\boldsymbol{K}xy = x \tag{4.2}$$

Por regla de simetría en (4,2) y transitividad de (4,1) y (4,2), tenemos $\boldsymbol{FF}xy = \boldsymbol{K}xy$ y utilizando dos veces la regla de inferencia extensional, podemos concluir que $\boldsymbol{FF} = \boldsymbol{K}$. □

De forma conversa $\boldsymbol{F}$ no puede definirse en términos de $\boldsymbol{S}$ y $\boldsymbol{K}$, dado que ambos son combinadores primitivos de caracter extensional: en tanto que $\boldsymbol{S}$ realiza una copia de uno de los combinadores que obtiene y los reordena, $\boldsymbol{K}$ los elimina, lo que imposibilita toda forma en la que pudiera hacerse cualquier clase de consulta a la estructura interna de los combinadores.

Aunque a primera vista resulte extraño que un cálculo combinatorialmente completo, como el cálculo $\boldsymbol{SK}$, no pueda expresar en sus términos un objeto que es expansión de $\mathcal{CL}$, esto es entendible pues el combinador $\boldsymbol{F}$ no es un combinador en el sentido clásico [22], ya que no es posible representarlo a partir de ningún λ-término.

El poder expresivo del cálculo $\boldsymbol{SF}$ no sólo es suficiente para expresar al cálculo $\boldsymbol{SK}$ como hemos visto, más importante, nos permite examinar la forma en la que fueron construidos los distintos combinadores. Podemos ilustrar esta superioridad frente a otros sistemas combinatorios como el cálculo $\boldsymbol{SK}$ y equivalentes por medio de un combinador capaz de mostrar la igualdad estructural entre las formas normales del cálculo $\boldsymbol{SF}$, resolviendo así el problema que usamos en un principio para motivar el capítulo. El algoritmo para esto es, en términos llanos, el siguiente: si las formas normales que se comparan son ambas compuestos, entonces se han de comparar sus componentes; si estos son ambos átomos, entonces compárense directamente por medio de un combinador `eqatom`; de otra forma, las formas normales no son iguales.

Definimos algunos combinadores que nos ayudan a formalizar nuestro algoritmo. Iniciamos por establecer los conceptos más elementales de la lógica proposicional:

Definición 4.4.2. (Verdad booleana).

$$\mathcal{V} \equiv \boldsymbol{K}$$

Definición 4.4.3. (Falsedad booleana).

$$\mathcal{F} \equiv \boldsymbol{SK}$$

Definición 4.4.4. (Negación).

$$\texttt{not}\ x \equiv \lambda^* x.(\boldsymbol{SK})(\boldsymbol{K})$$

De modo que:

$$\begin{aligned}\texttt{not}\ \mathcal{F} &\equiv \boldsymbol{SK}(\boldsymbol{SK})\boldsymbol{K} \\ &\to \boldsymbol{KK}(\boldsymbol{SKK}) \\ &\to \boldsymbol{K} \equiv \mathcal{V} \\ \texttt{not}\ \mathcal{V} &\equiv \boldsymbol{K}(\boldsymbol{SK})\boldsymbol{K} \\ &\to \boldsymbol{SK} \equiv \mathcal{F}\end{aligned}$$

Definición 4.4.5. (Disyunción).

$$\texttt{or}\ x\ y \equiv \lambda^* x.\lambda^* y.x(\boldsymbol{K})(y)$$

Así:

$$\texttt{or}\ \mathcal{F}\ \mathcal{F} \equiv \boldsymbol{SK}(\boldsymbol{K})(\boldsymbol{SK})$$

$$\begin{aligned}
&\to \boldsymbol{K}(\boldsymbol{SK})[\boldsymbol{K}(\boldsymbol{SK})]\\
&\to \boldsymbol{SK} \equiv \mathcal{F}\\
\texttt{or}\ \mathcal{F}\ \mathcal{V} &\equiv \boldsymbol{SK}(\boldsymbol{K})(\boldsymbol{K})\\
&\to \boldsymbol{K}(\boldsymbol{K})[\boldsymbol{K}(\boldsymbol{K})]\\
&\to \boldsymbol{K} \equiv \mathcal{V}\\
\texttt{or}\ \mathcal{V}\ \mathcal{F} &\equiv \boldsymbol{K}(\boldsymbol{K})(\boldsymbol{SK})\\
&\to \boldsymbol{K} \equiv \mathcal{V}\\
\texttt{or}\ \mathcal{V}\ \mathcal{V} &\equiv \boldsymbol{K}(\boldsymbol{K})(\boldsymbol{K})\\
&\to \boldsymbol{K} \equiv \mathcal{V}
\end{aligned}$$

Como se mostró en 3.1.1, al tener definidas la disyunción y negación, es posible expresar cualquier conectivo lógico. En los resultados y enunciaciones que siguen, hacemos uso de los símbolos clásicos dentro de la lógica proposicional para representar a los conectivos restantes (i.e., usamos $\Rightarrow$ para la implicación, $\Leftrightarrow$ para la doble implicación y $\wedge$ para la conjunción), obviando sus definiciones dentro del cálculo $\boldsymbol{SF}$ y tratándolos como operadores infijos.

Continuamos por definir un combinador que sirva para distinguir si un objeto del cálculo $\boldsymbol{SF}$ es un compuesto:

Definición 4.4.6. (`isComp`). El combinador `isComp` recibe un objeto x y reduce a $\boldsymbol{K}$ si este es un compuesto, mientras lo hace a $\boldsymbol{SK}$ en caso contrario.

$$\texttt{isComp} = \lambda x.\boldsymbol{F}x(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK}))$$

Sigue las siguientes reducciones:

$$\begin{aligned}
\texttt{isComp}O &\equiv \boldsymbol{F}O(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK}))\\
&\to \boldsymbol{SK}\\
\texttt{isComp}(PQ) &\equiv \boldsymbol{F}(PQ)(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK}))\\
&\to \boldsymbol{K}(\boldsymbol{KK})PQ\\
&\to \boldsymbol{KK}Q\\
&\to \boldsymbol{K}, \text{ si } PQ \text{ es un compuesto.}
\end{aligned}$$

Ejemplo 4.4.1.

$$\begin{aligned}
\texttt{isComp}\boldsymbol{S} &\equiv \boldsymbol{FS}(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK}))\\
\boldsymbol{FS}(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK})) &\to \boldsymbol{SK} \text{ (por axioma 2 de 4.4.1).}
\end{aligned}$$

Nótese que, por el axioma 3, el caso es idéntico si utilizamos al único átomo restante del cálculo, $\boldsymbol{F}$.

Ejemplo 4.4.2. Tómese $\boldsymbol{SKK}$, donde $P \equiv \boldsymbol{S}$ y $Q \equiv \boldsymbol{KK}$, en tanto que $PQ \equiv \boldsymbol{SKK}$.

$$\begin{aligned} \texttt{isComp}(\boldsymbol{SKK}) &\equiv \boldsymbol{F}(\boldsymbol{SKK})(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK})) \\ \boldsymbol{F}(\boldsymbol{SKK})(\boldsymbol{SK})(\boldsymbol{K}(\boldsymbol{KK})) &\to \boldsymbol{K}(\boldsymbol{KK})(\boldsymbol{SK})\boldsymbol{K} \\ &\to \boldsymbol{KKK} \\ &\to \boldsymbol{K} \end{aligned}$$

Definimos ahora combinadores homónimos a las famosas funciones `car` y `cdr` del lenguaje Lisp con el mismo comportamiento que en el lenguaje de McCarthy:

Definición 4.4.7. (`car`). La función `car` recupera la primera componente de un objeto que puede ser factorizado.

$$\texttt{car} = \lambda x.\boldsymbol{F}x\boldsymbol{IK}$$

Ejemplo 4.4.3.

$$\begin{aligned} \texttt{car}(\boldsymbol{S}MN) &\equiv (\lambda x.\boldsymbol{F}x\boldsymbol{IK})(\boldsymbol{S}MN) \\ (\lambda x.\boldsymbol{F}x\boldsymbol{IK})(\boldsymbol{S}MN) &\to \boldsymbol{K}(\boldsymbol{S}M)N \text{ (por 6 de 4.4.1)} \\ &\to \boldsymbol{S}M \end{aligned}$$

Definición 4.4.8. (`cdr`). De forma similar a `car`, definimos el combinador `cdr`, que recupera la segunda componente de un objeto factorizable.

$$\texttt{cdr} = \lambda x.\boldsymbol{F}x\boldsymbol{I}(\boldsymbol{KI})$$

Ejemplo 4.4.4.

$$\begin{aligned} \texttt{cdr}(\boldsymbol{S}MN) &\equiv \boldsymbol{F}(\boldsymbol{S}MN)\boldsymbol{I}(\boldsymbol{KI}) \\ \boldsymbol{F}(\boldsymbol{S}MN)\boldsymbol{I}(\boldsymbol{KI}) &\to \boldsymbol{KI}(\boldsymbol{SM})\boldsymbol{N} \text{ (por 6 de 4.4.1)} \\ &\to \boldsymbol{I}N \\ &\to N \end{aligned}$$

Los únicos dos átomos que existen en nuestro cálculo son $\boldsymbol{S}$ y $\boldsymbol{F}$. Es necesario el poder separarlos para la implementación de nuestro algoritmo. Definimos al siguiente combinador con este propósito:

Definición 4.4.9. (isF).

$$\texttt{isF} = \lambda x.x(\boldsymbol{KS})(\boldsymbol{KK})\boldsymbol{K}$$

De forma que $\boldsymbol{F}$ aplicado a isF mapea a $\boldsymbol{K}$ en tanto que $\boldsymbol{S}$ lo hace a $\boldsymbol{SK}$:

$$\begin{aligned}
\texttt{isF}\boldsymbol{F} &\equiv \boldsymbol{F}(\boldsymbol{KS})(\boldsymbol{KK})\boldsymbol{K} \\
&\to \boldsymbol{KKS} \\
&\to \boldsymbol{K} \\
\texttt{isF}\boldsymbol{S} &\equiv \boldsymbol{S}(\boldsymbol{KS})(\boldsymbol{KK})\boldsymbol{K} \\
&\to \boldsymbol{KSK}(\boldsymbol{KKK}) \\
&\to \boldsymbol{S}(\boldsymbol{KKK}) \\
&\to \boldsymbol{SK}
\end{aligned}$$

Podemos ahora utilizar al combinador isF para definir la igualdad entre átomos del cálculo $\boldsymbol{SF}$:

Definición 4.4.10. (eqatom).

$$\texttt{eqatom} = \lambda x.\lambda y.\texttt{isF}x \Rightarrow \texttt{isF}y$$

Con los combinadores hasta aquí definidos, podemos ahora pasar de la descripción del algoritmo a su implementación en forma de un combinador que es capaz de verificar la igualdad estructural de formas normales dentro del cálculo $\boldsymbol{SF}$:

```
equal = fix(λe.λx.λy.
   if isComp x
   then if isComp y
      then (e (car x)(car y)) and (e (cdr x)(cdr y))
      else KS
   else if isComp y
      then KS
   else eqatom x y).
```

A través del combinador equal es que podemos dar cuenta de la diferencia que guardan los dos combinadores que motivaron la discusión con la que se inició el análisis en torno a un cálculo combinatorio capaz de factorizar a sus

elementos, mismo que hemos expuesto en forma de cálculo $\boldsymbol{SF}$. Una corrida del algoritmo pasando como argumentos a $\boldsymbol{SKS}$ y $\boldsymbol{SKK}$ pintaría de la siguiente forma:

1. Nombrando a todo lo que se encuentra dentro de `fix` como α, el programa (que es un combinador de punto fijo y, así, tiene un comportamiento recursivo) computa $\alpha(\boldsymbol{F}\alpha)\boldsymbol{SKS}\ \boldsymbol{SKK}$, donde $e = (\boldsymbol{F}\alpha)$, $x = \boldsymbol{SKS}$ y $y = \boldsymbol{SKK}$.

2. `isComp` $\boldsymbol{SKS}$ evalúa a $\boldsymbol{K}$ (verdadero), en tanto que es un compuesto:

$$\begin{aligned}\texttt{isComp}(\boldsymbol{SKS}) &\equiv \boldsymbol{F}(\boldsymbol{SKS})(\boldsymbol{KI})(\boldsymbol{K}(\boldsymbol{KK}))\\ &\to \boldsymbol{K}(\boldsymbol{KK})(\boldsymbol{SK})\boldsymbol{K}\\ &\to \boldsymbol{KKK}\\ &\to \boldsymbol{K}\end{aligned}$$

3. `isComp` $\boldsymbol{SKK}$ evalúa a verdadero, pues es un combinador:

$$\begin{aligned}\texttt{isComp}(\boldsymbol{SKK}) &\equiv \boldsymbol{F}(\boldsymbol{SKK})(\boldsymbol{KI})(\boldsymbol{K}(\boldsymbol{KK}))\\ &\to \boldsymbol{K}(\boldsymbol{KK})(\boldsymbol{SK})\boldsymbol{K}\\ &\to \boldsymbol{KKK}\\ &\to \boldsymbol{K}\end{aligned}$$

4. Se evalúa la conjunción de $(\boldsymbol{Y}\alpha)\boldsymbol{SK}\ \boldsymbol{SK}$ con $(\boldsymbol{Y}\alpha)\boldsymbol{S}\ \boldsymbol{K}$.

5. Siendo `eqatom` el caso base de la recursión, la parte izquierda de la conjunción en (4) evalúa a verdadero después de bajar dos niveles en la recursión.

6. El cómputo de `eqatom` $\boldsymbol{S}\ \boldsymbol{K}$ resulta en falso, en tanto que al ser $\boldsymbol{S}$ un átomo, se procede a evaluar si $\boldsymbol{K}$ es un compuesto, y lo es, ya que:

$$\begin{aligned}\texttt{isComp}\ \boldsymbol{K} &\equiv \boldsymbol{FK}(\boldsymbol{KI})(\boldsymbol{K}(\boldsymbol{KK}))\\ &\to \boldsymbol{F}(\boldsymbol{FF})(\boldsymbol{KI})(\boldsymbol{K}(\boldsymbol{KK}))\\ &\to \boldsymbol{K}(\boldsymbol{KK})\boldsymbol{FF}\\ &\to \boldsymbol{KKF}\\ &\to \boldsymbol{K}\end{aligned}$$

7. La conjunción en (4) resulta en falso, en tanto que el lado derecho evaluó a $\boldsymbol{K}$ (6).

8. $\therefore \boldsymbol{SKS} \neq \boldsymbol{SKK}$.

Teorema 4.4.2. *Sean M y N $\boldsymbol{SF}$-combinadores en forma normal,* `equal` $M\ N \to \boldsymbol{K}$ *si y sólo si* $M = N$.

Demostración: La demostración es por inducción en la estructura de M. La formalización de ella se encuentra en [10].

Siendo que `equal` es una función recursiva en forma normal, expresemos a la igualdad estructural por medio del combinador de punto fijo $\boldsymbol{Y}$ tal como lo presentamos en 3.5.1 [39]:

```
equal = Y3(λ*e.λ*x.λ*y.
          Fx(Fy(eqatom x y)(SK))
            [F F [F F (λ*x1.λ*x2.Fy(SK)
              [F F [F F (λ*y1.λ*y2. e x1 y1 (e x2 y2)(SK))]])]]
```

Ejemplo 4.4.5. Con la nueva definición que hemos dado para `equal`, podemos volver a computar la igualdad intensional entre los combinadores $\boldsymbol{SKK}$ y $\boldsymbol{SKS}$, esta vez de una forma completamente transparente, no haciendo uso de nociones no formalizadas aún dentro del lenguaje, como son las expresiones condicionales. Para facilitar la escritura, usamos alias para partes todavía no computadas, y llamamos a la función que se aplica al combinador de punto fijo $\boldsymbol{Y}_3$, f.

$$
\begin{aligned}
\texttt{equal}\ (\boldsymbol{SKK})\ \boldsymbol{SKS} &= f(\boldsymbol{Y}_3 f)(\boldsymbol{SKK})(\boldsymbol{SKS}) \\
&= \boldsymbol{F}(\boldsymbol{SKK})\underbrace{(\boldsymbol{F}(\boldsymbol{SKS})\dots)}_{M}\underbrace{[\boldsymbol{FF}[\dots]\dots]}_{N} \\
&\to N(\boldsymbol{SK})\boldsymbol{K} \\
&= \boldsymbol{FF}\underbrace{[\boldsymbol{FF}(\boldsymbol{F}(\boldsymbol{SKK})(\boldsymbol{SK})\dots)\dots]}_{M}\underbrace{(\boldsymbol{SK})}_{N}\boldsymbol{K} \\
&\to \boldsymbol{M} \\
&= \boldsymbol{F}(\boldsymbol{SKS})\underbrace{(\boldsymbol{SK})}_{M}\underbrace{[\boldsymbol{FF}[\dots]\dots]}_{N} \\
&\to N(\boldsymbol{SK})\boldsymbol{S} \\
&= \boldsymbol{FF}\underbrace{[\boldsymbol{FF}(\dots)]}_{M}\underbrace{(\boldsymbol{SK})}_{N}\boldsymbol{S} \\
&\to M \\
&= \boldsymbol{FF}\underbrace{(\dots)}_{M}\underbrace{\boldsymbol{S}}_{N}
\end{aligned}
$$

$$
\begin{aligned}
&\to M \\
&= (\boldsymbol{Y}_3 f)(\boldsymbol{SK})(\boldsymbol{SK})((\boldsymbol{Y}_3 f)\boldsymbol{K}\ \boldsymbol{S})(\boldsymbol{SK}) \\
&= \texttt{equal}\ \ (\boldsymbol{SK})(\boldsymbol{SK})(\texttt{equal}\ \ \boldsymbol{K}\ \boldsymbol{S})(\boldsymbol{SK})
\end{aligned}
$$

Es claro que la reducción izquierda devuelve $\boldsymbol{K}$, en tanto que la segunda $\boldsymbol{SK}$ hace a la reducción total falsa, i.e. $\boldsymbol{SK}$. Computamos la segunda reducción para mostrar esto:

$$
\begin{aligned}
\texttt{equal}\ \ \boldsymbol{K}\ \boldsymbol{S} &= f(\boldsymbol{Y}_3 f)\ \boldsymbol{K}\ \boldsymbol{S} \\
&= \boldsymbol{F}\ \boldsymbol{K}(\boldsymbol{F}\ \boldsymbol{S}(\texttt{eqatom}\ \ \boldsymbol{K}\ \boldsymbol{K})(\boldsymbol{S}\ \boldsymbol{K}))\ldots \\
&= \boldsymbol{F}(\boldsymbol{FF})\underbrace{(\boldsymbol{F}\ \boldsymbol{S}\ (\texttt{eqatom}\ \ (\boldsymbol{FF})\ \boldsymbol{S})(\boldsymbol{SK}))}_{M}\underbrace{[\ldots]}_{N} \\
&\to \boldsymbol{NFF} \\
&= \boldsymbol{FF}\underbrace{[\boldsymbol{FF}(\boldsymbol{F}\ \boldsymbol{S}\ (\boldsymbol{SK})\ldots)\ldots]}_{M}\underbrace{\boldsymbol{F}}_{N}\boldsymbol{F} \\
&\to \boldsymbol{M} \\
&= \boldsymbol{FF}\underbrace{(\boldsymbol{F}\ \boldsymbol{S}(\boldsymbol{SK})\ldots)}_{M}\boldsymbol{F}_N \\
&\to \boldsymbol{M} \\
&= \boldsymbol{F}\ \boldsymbol{S}\underbrace{(\boldsymbol{SK})}_{M}\underbrace{[\boldsymbol{FF}[\ldots]]}_{N} \\
&\to \boldsymbol{M} \\
&= \boldsymbol{SK}
\end{aligned}
$$

De donde la reducción final de $\texttt{equal}\ \ (\boldsymbol{SKK})\ (\boldsymbol{SKS})$ resulta en:

$$
\begin{aligned}
&= \texttt{equal}\ \ (\boldsymbol{SK})(\boldsymbol{SK})(\texttt{equal}\ \ \boldsymbol{K}\ \boldsymbol{S})(\boldsymbol{SK}) \\
&= \texttt{equal}\ \ (\boldsymbol{SK})(\boldsymbol{SK})(\boldsymbol{SK})(\boldsymbol{SK}) \\
&= \boldsymbol{K}(\boldsymbol{SK})(\boldsymbol{SK}) \\
&= \boldsymbol{SK} \equiv \mathcal{F}
\end{aligned}
$$

La capacidad de factorización de los términos del lenguaje (capacidad aportada por el combinador $\boldsymbol{F}$), así como la caracterización sintáctica de las formas factorizables del mismo, dota de expresividad suficiente al cálculo para poder consultar la estructura interna de las expresiones dentro suyo y así, poder dar cuenta de la igualdad de los términos en un sentido intensional, contrastándolo con el extensional, que se hallaba ya en la lógica combinatoria, pero se interesaba únicamente por los resultados de los cómputos.

El cálculo aquí presentado cuenta con expresividad suficiente para hacer posible un análisis sintáctico de sus expresiones, logrando evaluar la forma en la que son construidos sus combinadores. Todo esto sin tener que sacrificar el caracter extensional que se halla dentro de la lógica combinatoria, mismo que permite ver que, en algún sentido, los dos $\mathcal{CL}$-términos que han servido para ejemplificar a los diversos combinadores definidos a lo largo de este capítulo, así como la igualdad intensional, son el mismo, en tanto que sus propiedades externas son idénticas al ambos representar a una misma función.

4.5. Emparejamiento de patrones

El emparejamiento de patrones explora las formas en las que es posible representar la factorización de la estructura interna de los objetos. Las operaciones necesarias para la implementación de esta maquinaria son generalmente integradas en los lenguajes de programación modernos como operaciones primitivas o funciones estándar, como es el caso de las famosas funciones `car` y `cdr` introducidas en Lisp por John McCarthy [15]. Estas operaciones, como fue ejemplificado previamente, permiten dar cuenta de la representación de compuestos arbitrarios como patrones de la forma $x\ y$ y, así, dotan a los lenguajes de la capacidad para obtener a x y a y y con ello, poder nombrar, reutilizar, etc., las partes de los objetos definidos en el lenguaje.

Ejemplo 4.5.1.

```
Inductive lterm : Type :=
| LVar : string -> lterm
| LAbs : string -> lterm -> lterm
| LApp : lterm -> lterm -> lterm.

Fixpoint search_st (s: string) (t: lterm) : bool :=
match t with
| LVar x => if string_dec x s then true else false
| LAbs a b => if string_dec s a then true else (search_st s b)
| LApp a b => orb (search_st s a) (search_st s b)
end.
```

Ejemplo de una función de emparejamiento de patrones dentro de Coq.

El emparejamiento de un patrón p con un argumento u, tal y como se define en [39], se lleva a cabo de la siguiente forma: Si p es una variable x, entonces se liga a x con u por medio de λ^*. Si el patrón es un combinador, se determina si el argumento lo es también, en cuyo caso se aplica `eqatom`. Si el

patrón es una aplicación de la forma $p_1\ p_2$, entonces se busca si el argumento u es un compuesto $u_1\ u_2$, en cuyo caso se emparejan p_1 con u_1 y p_2 con u_2, respectivamente.

Este emparejamiento puede entenderse viendo a los patrones p, que son términos en forma normal que contienen variables libres, como objetos que dentro de una función de emparejamiento tratan de amoldarse a los argumentos u que le son aplicados, de manera que se busca a p tal que las variables libres dentro suyo puedan recibir valor por medio de u; al encontrarse tal patrón, la función de emparejamiento devuelve el valor m que resulta de la substitución de las variables libres de p por uno o varios de los términos constitutivos de u.

La dificultad en formalizar esta clase de emparejamiento, conocida en la literatura como funciones de camino polimórfico [6], que consulta la estructura interna de sus argumentos y logra diferenciar su constitución a partir del uso de un patrón $x\ y$ utilizado para representar a compuestos arbitrarios, estriba en la unión de los casos utilizados en la función de emparejamiento. Al aplicarse un argumento u a una función de emparejamiento que contiene una secuencia de casos $p_0 \to m_0, p_1 \to m_1, \ldots p_n \to m_n$, la función se reduce al primer patrón $p_i \to m_i$ que logre ser emparejado. Comúnmente se representa a esta secuencia de patrones como una lista de casos

$$\begin{aligned} & p_0 \to m_0 \\ \mid\ & p_1 \to m_1 \\ & \ldots \\ \mid\ & p_n \to m_n \end{aligned}$$

donde '|' liga de forma más débil que la aplicación y asocia a la derecha, de manera que esta lista puede expresarse en forma lineal como:

$$p_0 \to m_0 \mid (p_1 \to m_1 \mid (\ldots (p_n \to m_n)))$$

Sobre la cual podemos utilizar *extensiones* [41, 42], generalizando esta representación como $p \to m \mid n$, donde n representa a los casos restantes. De esta manera, los casos dentro de una función de emparejamiento de patrones toman la siguiente forma:

$$p \to m \mid n = \boldsymbol{S}(p \to \boldsymbol{K}m)n$$

La cual es posible reducir a partir de la regla de reducción que establece la extensión, para lo cual introducimos las siguientes definiciones [6, 41]:

Definición 4.5.1. (Substitución). Una substitución σ es una función parcial que va de variables a términos. Su dominio $dom(\sigma)$ está dado por las variables $x_0, x_1, \ldots, x_n$ de un λ-término y mapea a cada x_i a una u_i: $\{u_0/x_0, u_1/x_1, \ldots, u_n/x_n\}$. La aplicación de una substitución σ a cualquier término t del cálculo λ no tipado, es dada por las siguientes reglas:

$$\begin{aligned}
\sigma x &= \sigma x && \text{si } x \in dom(\sigma) \\
\sigma x &= x && \text{si } x \notin dom(\sigma) \\
\sigma(ru) &= (\sigma r)(\sigma u) \\
\sigma(\lambda x.s) &= \lambda x.\sigma s && \text{si } x \notin dom(\sigma) \cup \bigcup_{x \in dom(\sigma)} FV(\sigma x)
\end{aligned}$$

Nótese que la condición que imponemos para la aplicación de la última regla sólo es relevante en representaciones del cálculo λ que necesiten de la α-equivalencia para evitar la colisión de variables.

Definición 4.5.2. (Emparejamientos). Los emparejamientos m pueden ser emparejamientos exitosos, dados por una substitución σ, o fracasos, denotados por Υ. El emparejamiento de un patrón p contra un argumento u se denota por $\{u/p\}$ y produce ya sea una substitución σ o Υ a partir de la regla de reducción

$$(p \to s)u \to \{u/p\}s$$

donde los emparejamientos básicos del patrón p contra el argumento u son dados por las siguientes ecuaciones ordenadas:

$$\begin{aligned}
\{u/x\} &= \{u/x\} \\
\{c/c\} &= \emptyset \\
\{uv/pq\} &= \{u/p\} \,\tilde{\cup}\, \{v/q\} \\
\{u/p\} &= \Upsilon
\end{aligned}$$

donde x y c son metavariables que denotan a variables de ligadura y a constructores, respectivamente, y la unión disjunta $\tilde{\cup}$ de dos formas emparejables m y n está dada por:

$$m \,\tilde{\cup}\, n = \begin{cases} \sigma_m x & \text{si } x \in dom(\sigma_m) \\ \sigma_n x & \text{si } x \in dom(\sigma_n) \\ \Upsilon & \text{otro caso} \end{cases} \tag{4.3}$$

Definición 4.5.3. (Extensiones). De manera intuitiva, el comportamiento que busca capturar la extensión es el siguiente:

$$\begin{aligned}(p \to m \mid n)u &\to \sigma n && \text{si } \{u/p\} = \text{ algún } \sigma \\ (p \to m \mid n)u &\to \boldsymbol{I}(nu) && \text{si } \{u/p\} = \Upsilon\end{aligned}$$

Lo que se consigue definiéndola de la siguiente manera:

$$p \to m \mid n = x \to (\Upsilon y \to y)((p \to z \to \Upsilon m)x(nx))$$

Donde x, y y z están ligadas en p, m y n.

A partir de estas definiciones, podemos ahora mostrar cómo aplicando un término u a una función de emparejamiento en la que se tenga un patrón p tal que para alguna σ tenga correspondencia (i.e. $\{u/p\}$), obtenemos la siguiente reducción:

$$\begin{aligned}(p \to m \mid n)u &= \boldsymbol{S}(p \to \boldsymbol{K}m)nu \\ &\to (p \to \boldsymbol{K}m)u(nu) \\ &\to \{u/p\}(\boldsymbol{K}m)(nu) \\ &= \sigma(\boldsymbol{K}m)(nu) \\ &\to (\sigma\boldsymbol{K})(\sigma m)(nu) \\ &\to \boldsymbol{K}(\sigma m)(nu) \\ &= \sigma m\end{aligned}$$

[2]

De forma similar, al no existir p tal que u pueda emparejarse dentro de la función de emparejamiento, resulta la siguiente reducción:

$$\begin{aligned}(p \to m \mid n)u &= \boldsymbol{S}(p \to \boldsymbol{K}m)nu \\ &\to (p \to \boldsymbol{K}m)u(nu) \\ &= nu\end{aligned}$$

Lo que muestra que es posible formalizar todas las nociones alrededor de las funciones de camino polimórfico para representar el emparejamiento

[2]Nótese que el paso de $\{u/p\}$ a σ se da en tanto que hemos afirmado que el patrón p logra emparejarse con u.

de patrones dentro del cálculo ***SF*** que, como ya se ha visto, contiene al cálculo ***SK*** y así, puede definir cualesquiera construcciones que provengan del cálculo λ, como las aquí expuestas.

Como el mismo Barry Jay muestra, es posible definir por medio de esta maquinaria las distintas funciones que hemos utilizado hasta ahora:

Definición 4.5.4. (`isCompound`).

```
isCompound M =
| x y ⇒ K
| x ⇒ KI
```

Definición 4.5.5. (`car`).

```
car M =
| u v ⇒ u
```

Definición 4.5.6. (`cdr`).

```
cdr M =
| u v ⇒ v
```

Definición 4.5.7. (`isF`).

```
isF M =
| F ⇒ K
| S ⇒ SK
```

Definición 4.5.8. (`equal`).

```
equal = Y3 f M N
| x1 x2 ⇒
   | y1 y2 ⇒ f x1 y1 (f x2 y2) (SK)
   | y ⇒ SK
| x ⇒
   | y1 y2 ⇒ SK
   | y ⇒ eqop x y.
```

Puesto que el cálculo ***SF*** permite la descomposición de toda estructura en sus partes constitutivas, el emparejamiento de patrones que nos abre es más expresivo que el que puede hallarse en lenguajes de programación

comunes, siendo los puramente funcionales los que retratan el uso más extendido y popular de estos métodos y que, al depender de tipos de datos algebraicos para realizar el emparejamiento, obligan a definir las funciones de emparejamiento de forma específica para la estructura de datos a la que habrán de ser aplicadas, por lo que tienen que establecer distintas funciones con mismo significado semántico para poder ser utilizadas sobre estructuras diversas; este problema no se encuentra en el cálculo que hemos expuesto, en tanto que cualquier patrón puede emparejarse con un compuesto, de forma que basta una única representación como función de emparejamiento para poder computarse sobre las estructuras más diversas, sin siquiera hacer uso de tipos.

Ilustramos el anterior párrafo definiendo funciones, tanto en Haskell como dentro del cálculo $\boldsymbol{SF}$, que recuperen el tamaño l de una estructura de datos; en este caso, computamos estas funciones sobre árboles binarios etiquetados y listas.

Ejemplo 4.5.2.

```
data L a = Nil | Cons a (L a) deriving (Show, Read, Eq, Ord)

lengthL :: L a -> Int
lengthL Nil = 0
lengthL (Cons h (t)) = 1 + lengthL t
```

Definición del tipo de datos `L` para representar listas dentro de Haskell, así como su respectiva función `lengthL` para calcular longitud.

Ejemplo 4.5.3.

```
data T a = Empty | Branch a (T a) (T a) deriving (Show, Read, Eq)

lengthT :: T a -> Int
lengthT Empty = 0
lengthT (Branch x (l) (r)) = 1 + lengthT l + lengthT r
```

Definición del tipo de datos `T` para representar árboles binarios etiquetados dentro de Haskell, así como su respectiva función `lengthT` para calcular longitud.

Ejemplo 4.5.4. Podemos definir a las listas dentro del cálculo λ de la siguiente forma:

$$\texttt{CONS} \equiv \lambda x.\lambda y.\lambda z.zxy$$
$$\texttt{NIL} \equiv \lambda x.\lambda y.y$$

En tanto que tenemos una forma de transformar todo λ-término en un combinador del cálculo $\boldsymbol{SK}$, y este último está a su vez contenido en nuestro cálculo, existe una formulación únicamente en términos del substituidor y el factorizador para estas definiciones.

De manera similar, definimos a los árboles binarios etiquetados como:

$$\texttt{EMPTY} \equiv \lambda e.\lambda b.e$$
$$\texttt{BRANCH} \equiv \lambda e.\lambda b.bx(leb)(reb)$$

Haciendo uso de la función sucesor, definida por:

$$\texttt{Succ} \equiv \lambda n.\lambda s.\lambda z.s(nsz)$$

Podemos ahora establecer una función genérica que para una estructura arbitraria X, devuelva su longitud:

```
length = Y₂ f X
| x ⇒ 0̄
| x y ⇒ Succ (length y)
```

Equivalente a:

```
length = Y₂(λ*f.λ*X.
      (S(x → K0̄)[x y → Succ(f y)]))
```

Nótese que tanto `EMPTY` como `NIL`, al no corresponder con alguna forma compuesta (como por ejemplo, `BRANCH` 0 `EMPTY` (`BRANCH` 5 `EMPTY` `EMPTY`)), se emparejan con el primer patrón, x, resultando en ambos casos en $\bar{0}$.

Capítulo 5

Cálculo $\lambda\boldsymbol{SF}$

En este capítulo se condensan varias de las ideas desarrolladas en los últimos quince años por Barry Jay [6, 43, 44, 41, 45, 9, 10, 39], que alcanzan el punto final de este trabajo en la exposición de las propiedades del cálculo $\lambda\boldsymbol{SF}$, así como la enumeración de diversos resultados en torno suyo.

Los intereses centrales en el trabajo de Barry Jay pueden verse enfocados en solucionar los conflictos que se dan al tener tantos y tan diversos paradigmas en los lenguajes de programación, preguntándose por la necesidad que hay de tenerlos tan distintos cuando todos tienen como base la misma noción intuitiva del cómputo. El autor se ha encargado de trazar la ruta hacia un cálculo más expresivo que el cálculo λ de Church, comúnmente visto como el "lenguaje de programación universal más simple", en el sentido en que este último no da cabida a la enorme riqueza que aportan los diversos paradigmas distintos al funcional, en particular los orientados a objetos y la programación por consultas (lenguajes de *querying*), así como existen limitaciones dentro de los mismos lenguajes basados en la formalización de Church, los lenguajes funcionales, al ser esta un sistema meramente extensional.

Esta accidentada ruta pasó del estudio del cálculo de patrones [6, 44, 45], que logra unificar a los paradigmas funcional, de consultas y orientado a objetos bajo la premisa de que todo cómputo efectuado en estos puede ser representado por medio de la coincidencia de patrones, al cálculo $\boldsymbol{SF}$ [9], extensión de la lógica combinatoria que, por medio de un operador capaz de factorizar términos del cálculo, puede hacer consultas sobre los programas vistos como formas normales cerradas del lenguaje y así permitir el análisis de ellos; finalmente, desemboca en el cálculo $\lambda\boldsymbol{SF}$ [10, 39], el cual es capaz tanto de hacer consultas a programas vistos ya como λ-términos, ya como combinadores, como de acceder por medio de consultas a los componentes que conforman a estos, permitiendo tanto el análisis de programas como la optimización de ellos, todo dentro del mismo cálculo, prescindiendo así de

cualquier metaproceso en la producción de las estructuras de datos optimizadas.

La forma común de ver a los lenguajes de programación de las ciencias de la computación y a los lenguajes formales de la lógica impone un cierto relieve artificial entre estos con el afán de establecer una distinción fundamental entre ellos. Se dice que los primeros no son otra cosa que herramientas hechas para "generar" una serie de instrucciones ejecutables dentro de una computadora digital, y los segundos, una formalización que sirve para definir y demostrar diversas propiedades sobre las bases teóricas que fundamentan a los primeros.

Uno de los propósitos de este trabajo (quizá el más importante) es establecer una postura contraria a esta, afirmando que esta distinción es completamente artificial, y sostenerla por medio de un formalismo capaz de fungir en los dos sentidos: poder utilizar lenguajes de programación para trabajar sobre las aseveraciones hechas por los lenguajes formales, así como poder mencionar los distintos programas (hechos desde los distintos lenguajes de programación), sus argumentos, los valores en los que resultan, etc., dentro de las aseveraciones hechas y verificadas por los lenguajes formales.

5.1. Programas como formas normales cerradas

El afán de enfocarnos en la identificación de los programas con las formas normales cerradas del cálculo responde a la necesidad de aislar términos que han llegado a su máxima reducción posible. Si bien aislar las funciones en aquellas que alguna vez terminan es imposible (pues esto sería equivalente a resolver el problema del paro [5]), se puede, tal y como se muestra en [10], utilizar técnicas de la lógica combinatoria para bloquear las reducciones que puedan llegar a ser problemáticas dentro de un programa, hasta que los argumentos necesarios sean recibidos. La principal dificultad de esto estriba en la representación de los distintos programas recursivos, en tanto que son estos los que podrían no llegar nunca a forma normal alguna. El ángulo que toma Barry Jay en la solución a este problema es el siguiente:

El tratamiento común de los programas recursivos se hace a partir de su representación por medio de un término en la forma $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, llegando a la definición ya familiar de los combinadores de punto fijo: $\boldsymbol{Y}F = F(WW)$, donde $W = \lambda x.F(xx)$. De este modo nos es posible definir funciones recursivas en entornos donde no es posible hacerlo por métodos iterativos, tal y como en el cálculo λ; sin embargo, surge el problema de representar a

esta clase de combinadores, que representan a programas recursivos, en una forma normal, i.e., llegar a una forma en la que no sea posible reducción alguna. La solución es retrasar la aplicación del primer término, W, al segundo (también W) a partir del reemplazo de la forma común, $\lambda f.WW$, por una extensionalmente equivalente introduciendo un combinador que preserva el comportamiento externo de los objetos a los que es aplicado:

Definición 5.1.1. (`wait`.)

$$\texttt{wait}\ M\ N \equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}\ M)(\boldsymbol{K}\ N))\boldsymbol{I}$$

Donde la reducción desemboca en una forma normal si M y N son a su vez formas normales. Nótese que la aplicación de algún combinador P reduce a la expresión a $M\ N\ P$:

$$\begin{aligned}\boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}M)(\boldsymbol{K}N))\boldsymbol{I}P &\to \boldsymbol{S}(\boldsymbol{K}M)(\boldsymbol{K}N)P(\boldsymbol{I}P)\\ &\to \boldsymbol{K}MP(\boldsymbol{K}NP)P\\ &\to M(N)P\\ &\to MNP\end{aligned}$$

De forma que `wait` espera a recibir tal término antes de aplicar M a N.

Lema 5.1.1. *Para cualesquiera términos M y N, `wait` $M\ N \xrightarrow{*} M\ N$.*

Demostración: Basta con mostrar que la forma normal a la que reduce `wait` MN no es otra cosa que MN:

$$\begin{aligned}\texttt{wait}\ MN &\equiv \boldsymbol{S}(\boldsymbol{S}(\boldsymbol{K}M)(\boldsymbol{K}N))\boldsymbol{I}\\ &\to \boldsymbol{S}(\boldsymbol{K}M)\boldsymbol{I}[(\boldsymbol{K}N)\boldsymbol{I}]\\ &\to \boldsymbol{K}M((\boldsymbol{K}N)\boldsymbol{I})[\boldsymbol{I}((\boldsymbol{K}N)\boldsymbol{I})]\\ &\to M[\boldsymbol{I}((\boldsymbol{K}N)\boldsymbol{I})]\\ &\to M[\boldsymbol{K}N\boldsymbol{I}]\\ &\to MN. \quad \square\end{aligned}$$

De esta forma, podemos definir un combinador de punto fijo $\boldsymbol{Y}^*$ que retrase la aplicación de los términos $\lambda x.f(xx)$, obteniendo así una forma normal como resultado de cualquier aplicación de otra forma normal f a su vez:

$$\boldsymbol{Y}^* = \lambda f.\texttt{wait}[\texttt{wait}(\lambda x.f(xx))(\lambda x.f(xx))]\ f$$

Interpretemos esto como una función de aridad binaria $\boldsymbol{Y}^2$:

$$\boldsymbol{Y}^2 = (\lambda f.\lambda x.M)$$

De esta manera, podemos retrasar la ejecución de la función f hasta recibir la cantidad necesaria de argumentos, utilizando combinadores de punto fijo con retraso de la forma $\boldsymbol{Y}^n$, donde:

$$\boldsymbol{Y}^n = (\lambda f.\lambda x_2.\lambda x_3 \ldots \lambda x_n.M)$$

$$\begin{aligned}\boldsymbol{Y}^n &= (\lambda f.\lambda x_2.\lambda x_3 \ldots \lambda x_n.M)\\ &= (\lambda f.\lambda x_2.\lambda x_3 \ldots \lambda x_n.\texttt{wait}[\texttt{wait}(f(x_2x_3 \ldots x_n))(f(x_2x_3 \ldots x_n))]\ f)\end{aligned}$$

Lema 5.1.2. $\boldsymbol{Y}^*$ *es extensionalmente equivalente a* $\boldsymbol{Y}$.

Demostración: Tenemos las siguientes definiciones:

$$\begin{aligned}\boldsymbol{Y}^* &\equiv \lambda f.\texttt{wait}[\texttt{wait}(\lambda x.f(xx))(\lambda x.f(xx))]f\\ \boldsymbol{Y} &\equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))\end{aligned}$$

Mostramos que $\boldsymbol{Y}^*$ se reduce a $\boldsymbol{Y}$ por medio de la regla de inferencia extensional de la siguiente manera:

$$\begin{aligned}\boldsymbol{Y}^*u &\equiv (\lambda f.\texttt{wait}[\texttt{wait}(\lambda x.f(xx))(\lambda x.f(xx))]f)u\\ &\to \texttt{wait}[\texttt{wait}(\lambda x.u(xx))(\lambda x.u(xx))]u\\ &\equiv \texttt{wait}(\lambda x.u(xx))(\lambda x.u(xx))u && \text{(por 5.1.1)}\\ &\equiv (\lambda x.u(xx))(\lambda x.u(xx))u && \text{(nuevamente)}\\ &\equiv \boldsymbol{Y}u.\end{aligned}$$

Así, $\boldsymbol{Y}^*u \equiv \boldsymbol{Y}u$, y por inferencia extensional, $\boldsymbol{Y}^* \equiv \boldsymbol{Y}$. □

Si bien esto logra establecer un combinador equivalente extensionalmente al combinador de punto fijo $\boldsymbol{Y}$ que nos garantiza obtener una forma normal cuando la función en recursión, f, recibe argumentos que están en forma

normal, topamos con un problema en cuanto funciones recursivas son compuestas, pues reintroducen al combinador de punto fijo con retardo y aumenta el número original de argumentos que se esperan recibir:

$$\begin{aligned}(\boldsymbol{Y}^2)\boldsymbol{Y}^2 &= (\lambda f.\texttt{wait}[\texttt{wait}(\lambda x.f(xx))(\lambda x.f(xx))]\ f)\boldsymbol{Y}^2\\ &= (\texttt{wait}[\texttt{wait}(\lambda x.\boldsymbol{Y}^2(xx))(\lambda x.\boldsymbol{Y}^2(xx))])\\ &= (\texttt{wait}[\texttt{wait}(\lambda x.\lambda f.\texttt{wait}[\texttt{wait}(\lambda x_1.f(x_1x_1))(\lambda x.f(x_1x_1))]\ f(xx))\\ &\quad (\lambda x.\lambda f.\texttt{wait}[\texttt{wait}(\lambda x_1.f(x_1x_1))(\lambda x.f(x_1x_1))]\ f(xx))])\end{aligned}$$

Para esto introducimos una estrategia de evaluación estricta (*eager*) a partir del uso de un combinador que permita la reducción únicamente cuando la forma del último término que se recibe es factorizable:

Definición 5.1.2. (`eager`).

$$\texttt{eager}\ M\ N \equiv \boldsymbol{F}N(\boldsymbol{SI}(\boldsymbol{K}N))(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{SI}))))(\boldsymbol{S}(\boldsymbol{KK})))$$

El combinador `eager` obliga la evaluación de N, de forma que resulte reducida a una forma factorizable, antes de permitir la aplicación de M a este.

Lema 5.1.3. *Para cualesquiera términos M y N, `eager` $M\ N \xrightarrow{*} M\ N$.*

Demostración: Procedemos desdoblando la definición de `eager`:

$$\texttt{eager}\ MN \equiv \boldsymbol{F}N(\boldsymbol{SI}(\boldsymbol{K}N))(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{SI}))))(\boldsymbol{S}(\boldsymbol{KK})))M$$

Sin pérdida de generalidad, asumamos que M y N son formas factorizables, es decir, son de la formas $\boldsymbol{S}, \boldsymbol{S}P, \boldsymbol{S}PQ, \boldsymbol{F}, \boldsymbol{F}P$ o $\boldsymbol{F}PQ$. En el caso en que N es un átomo (es decir, $\boldsymbol{S}$ o $\boldsymbol{F}$), la reducción resulta en M:

$$\begin{aligned}\boldsymbol{F}N(\boldsymbol{SI}(\boldsymbol{K}N))(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{SI}))))(\boldsymbol{S}(\boldsymbol{KK})))M &\to \boldsymbol{SI}(\boldsymbol{K}N)M\\ &\to \boldsymbol{I}M(\boldsymbol{K}NM)\\ &\to MN\end{aligned}$$

En tanto que la primera reducción está determinada por N, y M no determina ninguno de los pasos antes de llegar a la forma MN, como veremos a continuación, bástenos tomar $N = \boldsymbol{X}P$, donde X es un átomo. Tenemos ahora que:

$$\begin{aligned}
\texttt{eager}\ M\ \boldsymbol{X}P &\to \boldsymbol{S}(\boldsymbol{K}(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{SI}))))(\boldsymbol{S}(\boldsymbol{KK}))\boldsymbol{X}PM \\
&\to \boldsymbol{K}(\boldsymbol{S}(\boldsymbol{K}(\boldsymbol{SI})))\boldsymbol{X}[\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{X}]PM \\
&\to \boldsymbol{S}(\boldsymbol{K}(\boldsymbol{SI}))[\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{X}]PM \\
&\to \boldsymbol{K}(\boldsymbol{SI})P[\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{X}P]M \\
&\to \boldsymbol{SI}P[\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{X}P]M \\
&\to \boldsymbol{I}M[\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{X}PM] \\
&\to M[\boldsymbol{S}(\boldsymbol{KK})\boldsymbol{X}PM] \\
&\to M[\boldsymbol{KK}P(\boldsymbol{X}P)M] \\
&\to M[\boldsymbol{K}(\boldsymbol{X}P)M] \\
&\to M(\boldsymbol{X}P) \equiv MN
\end{aligned}$$

Para formas factorizables $\boldsymbol{X}PQ$ el procedimiento es análogo al anterior. Basta substituir $\boldsymbol{X}$ por $(\boldsymbol{X}P)$ y P por Q en la reducción que acabamos de hacer:

$$\begin{aligned}
\texttt{eager}\ M\ (\boldsymbol{X}P)Q &\xrightarrow{*} \boldsymbol{I}M[\boldsymbol{S}(\boldsymbol{KK})(\boldsymbol{X}P)QM] \\
&\to M[\boldsymbol{S}(\boldsymbol{KK})(\boldsymbol{X}P)QM] \\
&\to M[\boldsymbol{KK}Q((\boldsymbol{X}P)Q)M] \\
&\to M[\boldsymbol{K}((\boldsymbol{X}P)Q)M] \\
&\to M((\boldsymbol{X}P)N) \equiv MN. \quad \square
\end{aligned}$$

Ahora definimos una vez más un combinador de punto fijo, esta vez cuidándonos no sólo de que obligue a resultar en una forma normal, como hicimos a través del retardo en la aplicación de los términos, sino también que, en el caso de recibir cualquier forma recursiva, bloquee la recursión exterior hasta recibir a toda variable ligada necesaria:

$$\boldsymbol{Y}^{*'} = \lambda f.\lambda x.\texttt{eager}(\texttt{wait}[\texttt{wait}(\lambda x.f(xx))(\lambda x.f(xx))]\ f)\ x$$

5.2. Propiedades

Podemos desprender una serie de resultados acerca del cálculo $\lambda\boldsymbol{SF}$ ahora que hemos establecido la interpretación que tienen las formas normales cerradas de este (que no son otra cosa que los programas). Antes de la enumeración de distintas propiedades del cálculo, pongamos de manera concisa

su alfabeto, términos, reglas y axiomas tal como hicimos antes con el cálculo $\boldsymbol{SK}$ y el cálculo $\boldsymbol{SF}$:

Definición 5.2.1. ($\mathcal{CL}_{\lambda\boldsymbol{SF}}$). El alfabeto a partir del cual se formulan todos los términos del cálculo es el siguiente:

1. Combinadores o constantes: $\boldsymbol{S}, \boldsymbol{F}$.
2. Variables: $x_0, x_1, \ldots$.
3. Relación de igualdad: $=$.
4. Símbolos impropios: $(,), [,]$.
5. Símbolo de abstracción o ligadura de variable: λ.

A partir del cual se puede expresar el conjunto de los $\mathcal{CL}_{\lambda\boldsymbol{SF}}$-términos, mismos que establecemos de forma inductiva como:

1. $x \in \mathcal{CL}_{\lambda\boldsymbol{SF}}$ para cualquier variable de $\mathcal{CL}_{\lambda\boldsymbol{SF}}$.
2. $\boldsymbol{S} \in \mathcal{CL}_{\lambda\boldsymbol{SF}}, \boldsymbol{F} \in \mathcal{CL}_{\lambda\boldsymbol{SF}}$.
3. $P, Q \in \mathcal{CL}_{\lambda\boldsymbol{SF}} \Rightarrow (PQ) \in \mathcal{CL}_{\lambda\boldsymbol{SF}}$.
4. $P, x \in \mathcal{CL}_{\lambda\boldsymbol{SF}} \Rightarrow \lambda x.P \in \mathcal{CL}_{\lambda\boldsymbol{SF}}$.

Finalmente, los axiomas de nuestro cálculo son los siguientes:

1. $(\lambda x.M)N \to M[x := N]$.
2. $\boldsymbol{S}MNP \to MP(NP)$.
3. $\boldsymbol{F}PMN \to M$, si P es $\boldsymbol{S}$ o $\boldsymbol{F}$.
4. $\boldsymbol{F}PMN \to NP\rceil\lceil P$, donde P es el compuesto formado por $P\rceil$ y $\lceil P$.

Definición 5.2.2. (Componente izquierdo). Definimos al componente izquierdo $M\rceil$ de un $\mathcal{CL}_{\lambda\boldsymbol{SF}}$-término como [10]:

$$\begin{aligned}(MN)\rceil &= M\\ M\rceil &= \boldsymbol{SKF}\end{aligned}$$

Definición 5.2.3. (Componente derecho). El componente derecho $\lceil M$ de un $\mathcal{CL}_{\lambda\boldsymbol{SF}}$-término M se define como [10]:

$$\begin{aligned}\lceil(MN) &= N\\ \lceil(\lambda x.M) &= \lambda^* x.M\\ \lceil M &= M\end{aligned}$$

Los siguientes teoremas fueron anteriormente formalizados en Coq [10]. Ofrecemos aquí un tratamiento "común" de ellos, dando las primeras demostraciones de estos resultados en la literatura en el sentido más clásico de la palabra.

Teorema 5.2.1. *La reducción del cálculo λ**SF** es confluente.*

Antes de demostrar este teorema, introducimos una serie de definiciones y lemas que facilitarán derivar nuestro resultado.

Definición 5.2.4. (Sistema abstracto de reescritura). Llamamos a una tupla (S, R), donde S es un conjunto de elementos y R una relación, que llamamos de reducción, sobre los elementos de S, un sistema abstracto de reescritura.

Afirmación: $(\mathcal{CL}_{\lambda \boldsymbol{SF}}, \rightarrow)$ es un sistema abstracto de reescritura.

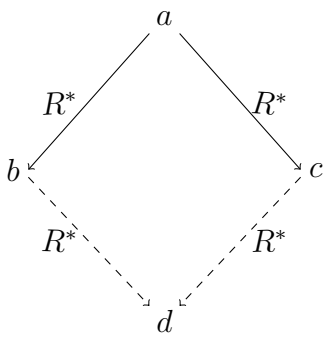
Definición 5.2.5. (Cerradura reflexiva transitiva). Dada una relación R sobre un conjunto S, decimos que esta es la cerradura reflexiva transitiva, y la denotamos por R^*, si es la menor relación dentro de S que cumple con:

- $\forall a \in S : aR^*a$.
- $\forall a, b, c \in S : aR^*b \wedge bR^*c \Rightarrow aR^*c$.

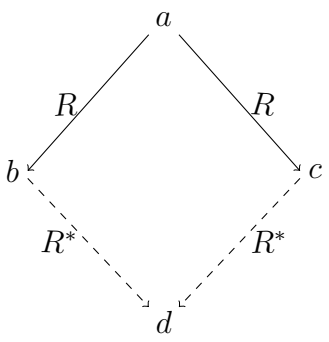
Definición 5.2.6. (Relación de equivalencia). Dada una relación R sobre un conjunto S, decimos que esta es de equivalencia, y la denotamos por R^+, si cumple con ser reflexiva, transitiva y simétrica:

- $\forall a \in S : aR^+a$.
- $\forall a, b, c \in S : aR^+b \wedge bR^+c \Rightarrow aR^+c$.
- $\forall a, b \in S : aR^+b \Rightarrow bR^+a$.

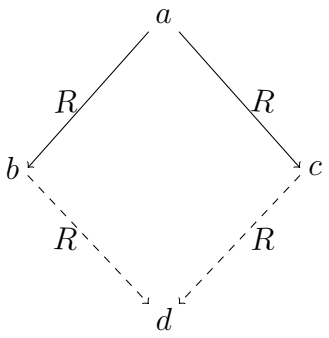
Definición 5.2.7. (Confluencia). Decimos que una relación R^* sobre un conjunto S es confluente si $\forall a, b, c \in S : aR^*b \wedge aR^*c \Rightarrow \exists d., bR^*d \wedge cR^*d$.



Definición 5.2.8. (Confluencia local). Decimos que las relaciones R y R^* sobre un conjunto S son localmente confluentes si $\forall a, b, c \in S : aRb \wedge aRc \Rightarrow \exists d., bR^*d \wedge cR^*d$.

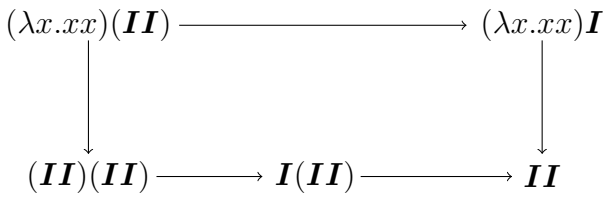


Definición 5.2.9. (Propiedad del diamante). Decimos que una relación R sobre un conjunto S cumple con la propiedad del diamante si $\forall a, b, c \in S : aRb \wedge aRc \Rightarrow \exists d., bRd \wedge cRd$.



Definición 5.2.10. (Propiedad de Church-Rosser). Decimos que una relación R sobre un conjunto S tiene la propiedad de Church-Rosser si $\forall a, b \in S : aR^+b \Rightarrow \exists c., aR^*c \wedge bR^*c$.

Es en general fácil mostrar que un sistema abstracto de reescritura (S, R) es confluente mostrando únicamente que R cumple con la propiedad del diamante. Sin embargo, para la mayoría de sistemas abstractos de reescritura, no es el caso, lo que vuelve la prueba un tanto complicada. Tomemos por ejemplo al cálculo λ, $(\Lambda, \rightarrow_\beta)$, para el cual no se cumple la propiedad (estando contenido el cálculo λ dentro del cálculo $\lambda\boldsymbol{SF}$, el resultado aplica también para este último), como es fácil mostrar:



Para demostrar que nuestro cálculo es confluente, iniciamos por seguir la técnica empleada por Tait y Martin-Löf [46, 47]: definimos primero una relación de reducción paralela $\gg$ que cumple con ser reflexiva y reducir términos de forma simultánea. Posteriormente, definimos una reducción a "desarrollos completos", $\succ$, introducidos originalmente por Takahashi [48], para cada elemento de $\mathcal{CL}_{\lambda\boldsymbol{SF}}$ y finalmente, llegamos a la propiedad del diamante a partir de mostrar que nuestras relaciones cumplen con ser triángulares dentro del cálculo $\lambda\boldsymbol{SF}$.

Definición 5.2.11. (Reducción paralela). Definimos a la reducción paralela dentro del cálculo $\lambda\boldsymbol{SF}$, denotada por $\gg$, de forma inductiva como:

$$\frac{M \gg M' \quad N \gg N'}{(\lambda x.M)N \gg M'[x := N']}\ \text{rp-1} \qquad \frac{}{x \gg x}\ \text{rp-2}$$

$$\frac{M \gg M' \quad N \gg N'}{MN \gg M'N'}\ \text{rp-3}$$

$$\frac{M \gg M'}{\lambda x.M \gg \lambda x.M'}\ \text{rp-4} \qquad \frac{M \gg M' \quad N \gg N' \quad P \gg P'}{\boldsymbol{S}MNP \gg M'P'(N'P')}\ \text{rp-5}$$

$$\frac{P \gg P' \quad N \gg N' \quad \texttt{isComp}(P)}{\boldsymbol{F}PMN \gg N'P'\rceil\lceil P'}\ \text{rp-6} \qquad \frac{M \gg M' \quad \neg\texttt{isComp}(P)}{\boldsymbol{F}PMN \gg M'}\ \text{rp-7}$$

$$\frac{M \gg M' \quad N \gg N'}{M[x := N] \gg M'[x := N']}\ \text{rp-8}$$

Nótese que $\gg$ no es es una relación determinística en tanto que puede haber diversas elecciones para la reducción de un mismo $\mathcal{CL}_{\lambda\boldsymbol{SF}}$−término.

Definición 5.2.12. (Desarrollos completos). Definimos a la reducción a desarrollos completos dentro del cálculo $\lambda\boldsymbol{SF}$, denotada por $\succ$ de la siguiente forma:

(dc-1) Si $A \equiv (\lambda x.M)N$, entonces $A \succ \tilde{A} \equiv \tilde{M}[x := \tilde{N}]$.

(dc-2) Si $A \equiv x$, entonces $A \succ \tilde{A} \equiv x$.

(dc-3) Si $A \equiv MN$, entonces $A \succ \tilde{A} \equiv \tilde{M}\tilde{N}$ si M no es una abstracción.

(dc-4) Si $A \equiv \lambda x.M$, entonces $A \succ \tilde{A} \equiv \lambda x.\tilde{M}$.

(dc-5) Si $A \equiv \boldsymbol{S}MNP$, entonces $A \succ \tilde{A} \equiv \tilde{M}\tilde{P}(\tilde{N}\tilde{P})$.

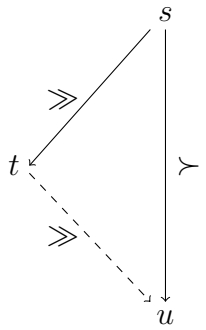
(dc-6) Si $A \equiv \boldsymbol{F}PMN$, entonces $A \succ \tilde{A} \equiv \tilde{N}\tilde{P}\rceil\lceil\tilde{P}$ si $\texttt{isComp}(P)$.

(dc-7) Si $A \equiv \boldsymbol{F}PMN$, entonces $A \succ \tilde{A} \equiv \tilde{M}$ si $\neg\texttt{isComp}(P)$.

Lema 5.2.2. *$\forall s \in \mathcal{CL}_{\lambda\boldsymbol{SF}}\ \exists t., s \succ t$, i.e., para cualquier elemento s del cálculo $\lambda\boldsymbol{SF}$, existe un desarrollo completo.*

Demostración: La prueba es por inducción estructural en s. En tanto que las reducciones hechas por $\succ$ son determinísticas, exactamente una regla aplica en la reducción de cada forma en la que puede aparecer un término s, empleando cada uno de los casos de la definición anterior y la hipótesis de inducción, es suficiente para establecer la existencia de un desarrollo completo para las diversas formas que pueda tener un $\mathcal{CL}_{\lambda\boldsymbol{SF}}$-término.

Lema 5.2.3. *$\forall s,t,u \in \mathcal{CL}_{\lambda\boldsymbol{SF}} : s \gg t \wedge s \succ u \Rightarrow t \gg u$, i.e., $\succ$ cierra cualquier triángulo dado por $\gg$.*



Demostración: Procedemos por inducción estructural en $s \succ u$.

(i) Si $s = (\lambda x.M)N \succ \tilde{M}[x := \tilde{N}] = u$ por dc-1, hay dos subcasos posibles para la reducción paralela $\gg$ de s:

  (a) $s = (\lambda x.M)N \gg M'[x := N']$ por rp-1, y por hipótesis de inducción tenemos que $M' \gg \tilde{M}$ y $N' \gg \tilde{N}$, por rp-8 $t = M'[x := N'] \gg \tilde{M}[x := \tilde{N}] = u$.

  (b) $s = (\lambda x.M)N \gg (\lambda x.M')N'$ por rp-3 a partir de rp-4, por lo que $\lambda x.M \gg \lambda x.M'$, por hipótesis de inducción, $M' \gg \tilde{M}$ y $N' \gg \tilde{N}$, por rp-1 $t = (\lambda x.M')N' \succ \tilde{M}[x := \tilde{N}] = u$.

(ii) Si $s = x \succ x = u$, $s \gg x = t$, por rp-2, $t \gg u$, pues $s = t = u$.

(iii) Si $s = MN \succ \tilde{M}\tilde{N} = u$ por dc-3, $s \gg M'N' = t$, por hipótesis de inducción tenemos que $M' \gg \tilde{M}$ y $N' \gg \tilde{N}$, de donde $t = M'N' \gg \tilde{M}\tilde{N} = u$ a partir de rp-3.

(iv) Si $s = \lambda x.M \succ \lambda x.\tilde{M} = u$ por dc-3, tenemos por rp-4 $s \gg \lambda x.M' = t$ y por hipótesis de inducción $M' \gg \tilde{M}$. Nuevamente por rp-4, $t = \lambda x.M' \gg \lambda x.\tilde{M} = u$.

(v) Si $s = \boldsymbol{S}MNP \succ \tilde{M}\tilde{P}(\tilde{N}\tilde{P}) = u$ por dc-5. Nuevamente hay dos casos posibles para la reducción paralela $\gg$ de s.

(a) $s \gg M'P'(N'P') = t$ por rp-5, y por hipótesis de inducción $M' \gg \tilde{M}$, $N' \gg \tilde{N}$ y $P' \gg \tilde{P}$, por aplicación repetida de rp-3, $t = M'P'(N'P') \gg \tilde{M}\tilde{P}(\tilde{N}\tilde{P})$.

(b) $s \gg \boldsymbol{S}'M'N'P'$ por aplicación de rp-3 para cualquier combinación de aplicaciones en tanto que todos los elementos existen en $\mathcal{CL}_{\lambda\boldsymbol{SF}}$. Por hipótesis de inducción, $M' \gg \tilde{M}$, $N' \gg \tilde{N}$ y $P' \gg \tilde{P}$, de donde, por rp-5, $t = \boldsymbol{S}'M'N'P' \gg \tilde{M}\tilde{P}(\tilde{N}\tilde{P}) = u$.

(vi) Si $s = \boldsymbol{F}PMN$ y P es un componente, $s \succ \tilde{N}\tilde{P}\rceil\lceil\tilde{P} = u$ por dc-6. Hay dos casos posibles para la reducción paralela $\gg$ de s.

(a) $s \gg N'P'\rceil\lceil P' = t$ por rp-6. Por hipótesis de inducción $P' \gg \tilde{P}$ y $N' \gg \tilde{N}$, por la definición de los componentes izqueirdo y derecho $t = N'\boldsymbol{SKF}P'$, y aplicación repetida de rp-3 sobre las distintas combinaciones posibles: $t = N'\boldsymbol{SKF}P' \gg \tilde{N}\boldsymbol{S}'\boldsymbol{K}'\boldsymbol{F}'\tilde{P} = \tilde{N}P'\rceil\lceil\tilde{P} = \tilde{N}\tilde{P}\rceil\lceil\tilde{P} = u$.

(b) $s \gg \boldsymbol{F}'P'M'N'$ por aplicación repetida de la reducción paralela para la aplicación, rp-3. Por hipótesis de inducción $P' \gg \tilde{P}$, $M' \gg \tilde{M}$ y $N' \gg \tilde{N}$, de donde, ya que P' es un componente, y usando rp-6: $t = \boldsymbol{F}'P'M'N' \gg \tilde{N}\tilde{P}\rceil\lceil\tilde{P} = u$.

(vii) Finalmente, si $s = \boldsymbol{F}PMN$ con P un átomo, $s \succ \tilde{M}$. Nuevamente existen dos casos para la reducción paralela $\gg$ de s.

(a) El caso en el que $s \gg \boldsymbol{F}'P'M'N'$ por la aplicación repetida de rp-3 es análogo al caso (b) del inciso anterior, cambiando únicamente la última reducción por rp-7.

(b) $s \gg M' = t$ por rp-7, y por hipótesis de inducción, $M' \gg \tilde{M}$, pero $M' = t$, de donde $t \gg u$.

□

Lema 5.2.4. $\gg$ *cumple con la propiedad del diamante.*

Demostración: Como muestra Takahashi [49], para probar el teorema de Church-Rosser para una relación R, basta con probar la propiedad del diamante para esta relación, pero es también posible hacerlo mostrando que cumple con la propiedad del triángulo: $aRn \Rightarrow bRa^*$, donde a^* es determinado por a e independiente de b. Como van Oostrom afirma [50], la propiedad del triángulo implica la propiedad del diamante, así, por 5.2.3, $\gg$ cumple con ella. $\square$

Lema 5.2.5. *Si una relación R cumple con la propiedad del diamante entonces R es confluente.*

Demostración: Basta con notar que la cerradura reflexiva transitiva de una relación R es idempotente, i.e., $\forall a, b \in S : aRb = aR^*b$. $\square$

Lema 5.2.6. $\gg$ *es confluente.*

Demostración: Por 5.2.4 y 5.2.5. $\square$

Corolario 5.2.7. *La reducción paralela del cálculo $\lambda\boldsymbol{SF}$ está contenida en la $\mathcal{CL}_{\lambda\boldsymbol{SF}}$-reducción: $\rightarrow \subseteq \gg$.*

Demostración: La demostración es inmediata en tanto que cada reducción $\rightarrow$ del cálculo $\lambda\boldsymbol{SF}$ es el primer paso de una reducción paralela $\gg$. $\square$

Corolario 5.2.8. *La cerradura reflexiva transitiva de la $\mathcal{CL}_{\lambda\boldsymbol{SF}}$-reducción está contenida en la reducción paralela del cálculo $\lambda\boldsymbol{SF}$: $\gg \subseteq \rightarrow^*$.*

Demostración: Por inducción en cada una de las derivaciones de $\gg$ para cualesquiera $a, b \in \mathcal{CL}_{\lambda\boldsymbol{SF}}$. $\square$

Lema 5.2.9. *Sean R y S relaciones binarias sobre el mismo conjunto tales que $R \subseteq S \subseteq R^*$. Entonces R es confluente si S tiene la propiedad del diamante.*

Demostración: La cerradura reflexiva transitiva de una relación cumple con ser idempotente y monótona, de donde $R \subseteq S \subseteq R^* \Rightarrow R^* \subseteq S^* \subseteq (R^*)^* = R^*$, y en tanto $R^* \subseteq S^*$ y $S^* \subseteq R^*$, $R^* = S^*$. Por hipótesis, S cumple con la propiedad del diamante, de donde, por 5.2.5, es confluente, y por idempotencia de la cerradura reflexiva transitiva, S^* cumple también con la propiedad del diamante y ser confluente, pero se vio ya que $R^* = S^*$, de donde R^* cumple con la propiedad del diamante, y al ser $R \subseteq R^*$ por nuestra hipótesis, concluimos que R es confluente. $\square$

Finalmente, concluimos con la prueba de 5.2.1:

Demostración: Por los lemas 5.2.4, 5.2.7, 5.2.8 y 5.2.9. □

Nótese que al incluir este cálculo al cálculo $\boldsymbol{SF}$, el resultado de este teorema se extiende al último.

Teorema 5.2.10. *No existe ningún homomorfismo del cálculo $\lambda\boldsymbol{SF}$ al cálculo λ.*

Tal y como se define dentro de [10], un homomorfismo es una función que va de un sistema de reescritura con variables (tal y como el cálculo $\lambda\boldsymbol{SF}$) a otro y mantiene las relaciones de equivalencia derivadas de las reglas de reducción, las aplicaciones, las variables y no introduce variables libres. La siguiente es la demostración hecha por Jay en el artículo citado:

Demostración: Procedemos por reducción al absurdo. Asúmase que existe dicho homomorfismo. Puede entonces tenerse uno que vaya del cálculo λ junto con la β y η reducciones, para obtener un homomorfismo $[\![-]\!]$. De aquí, se sigue que:

$$[\![\boldsymbol{S}]\!] \equiv \lambda x.\lambda y.\lambda z.[\![\boldsymbol{S}]\!]xyz \equiv \lambda x.\lambda y.\lambda z.[\![\boldsymbol{S}xyz]\!] \equiv \lambda x.\lambda y.\lambda z.xz(yz)$$

De manera similar, podemos mostrar que $[\![\boldsymbol{K}]\!] \equiv \lambda x.\lambda y.x$ y $[\![\boldsymbol{I}]\!] \equiv \lambda x.x$. Ahora, dentro del cálculo $\boldsymbol{SF}$ tenemos:

$$\begin{aligned}\boldsymbol{F}(\boldsymbol{SK}M)\boldsymbol{I}(\boldsymbol{KI}) &\to \boldsymbol{KI}(\boldsymbol{SK})M \\ &\to \boldsymbol{I}M \\ &\to M\end{aligned}$$

Por lo que:

$$[\![\boldsymbol{F}(\boldsymbol{SK}M)\boldsymbol{I}(\boldsymbol{KI})]\!] \equiv [\![\boldsymbol{F}]\!]([\![\boldsymbol{S}]\!][\![\boldsymbol{K}]\!][\![M]\!])[\![\boldsymbol{I}]\!]([\![\boldsymbol{K}]\!][\![\boldsymbol{I}]\!])$$

En tanto que $\boldsymbol{SK}X \equiv \boldsymbol{I}$ para todo combinador X:

$$\begin{aligned}&\equiv [\![\boldsymbol{F}]\!][\![\boldsymbol{I}]\!][\![\boldsymbol{I}]\!]([\![\boldsymbol{K}]\!][\![\boldsymbol{I}]\!]) \\ &\equiv [\![\boldsymbol{F}]\!](\lambda x.x)(\lambda x.x)(\lambda x.\lambda y.x(\lambda x.x)) \\ &\equiv [\![\boldsymbol{F}]\!](\lambda x.x)(\lambda x.x)(\lambda y.\lambda x.x)\end{aligned}$$

$$\equiv_\alpha [\![\boldsymbol{F}]\!](\lambda x.x)(\lambda x.x)(\lambda x.\lambda y.y)$$

Y, pues afirmamos que tenemos un homomorfismo, se tiene que:

$$[\![M]\!] \equiv [\![\boldsymbol{F}]\!](\lambda x.x)(\lambda x.x)(\lambda x.\lambda y.y)$$

Así, el lado derecho de la equivalencia es independiente de M por lo que, para cualquier N, es posible derivar $[\![M]\!] \equiv [\![N]\!]$, lo que es una contradicción; de donde concluimos que no hay un homomorfismo del cálculo $\boldsymbol{SF}$ al cálculo λ, y en tanto que $\mathcal{CL}_{\boldsymbol{SF}} \subseteq \mathcal{CL}_{\lambda\boldsymbol{SF}}$, tampoco del cálculo $\lambda\boldsymbol{SF}$ al cálculo λ. □

5.3. Análisis y optimización de programas

El análisis de programas compone un área de investigación extensa dentro de los compiladores y lenguajes de programación encargada de, entre otras cosas, ofrecer técnicas estáticas en tiempo de compilación que logren predecir de manera aproximada las acciones computables seguras por las que es posible llegar a un conjunto de valores o comportamientos dados de forma dinámica durante el tiempo de ejecución de los programas [51]. Para esta clase de análisis, es necesario formalizar la manera en que hemos de razonar respecto a las transformaciones por las que pasa un programa: se requiere considerar a detalle la semántica tanto del programa que será transformado, como del programa que realizará dicha transformación.

No existen a la fecha lenguajes de programación donde los programas sean miembros de primera clase, como sí pueden llegar a serlo las funciones en los lenguajes de programación funcionales o los tipos de datos más clásicos como aquellos que se usan para los números enteros, de punto flotante o booleanos, en los lenguajes imperativos. Típicamente, la manipulación del código que compone a un programa y la generación de su ejecutable se llevan a cabo a partir de la representación de este en una manera *ad hoc*, utilizando los tipos de datos estándar en algún lenguaje: esta es una de las tareas más clásicas (acaso la principal) que realizan los compiladores, los cuales pueden ser vistos de la manera más simple posible como programas capaces de leer a otros, escritos en un lenguaje l, y traducirlos a programas extensionalmente equivalentes a los originales en un lenguaje t, aumentando las más de las veces el nivel de abstracción, y realizando un gran número de optimizaciones sobre el código del programador.

Las técnicas comúnmente utilizadas en el análisis y optimización de programas son logradas sólo a partir de la formalización de los conceptos necesarios para diseccionar a los programas, labor realizable frecuentemente en

la práctica a partir de técnicas de metaprogramación, las cuales permiten la inspección y manipulación de la estructura interna del código. Desafortunadamente, no es común dentro de los lenguajes de programación populares dar soporte a técnicas de metaprogramación suficientemente desarrolladas como para dar cuenta de los programas en su totalidad. Aunado a esto, es mucha la dificultad que representa el uso de tales herramientas, razón por la cual su popularidad es muy baja y su uso muy limitado.

Ejemplo 5.3.1. La metaprogramación en C++ se realiza a partir de su sistema de *templates*, usados por su compilador para generar código fuente temporal, mismo que se incorpora al resto del código durante el tiempo de compilación. Históricamente, estas técnicas fueron descubiertas de forma accidental; mostradas por vez primera por Erwin Unruh en 1995, quien en tiempo de compilación logró computar diversos números primos dentro de los mensajes de error del compilador [52, 53]. A continuación se encuentra el código de Unruh, actualizado a los estándares modernos del lenguaje:

```
template<int i> struct D { D(void*); operator int(); };

template<int p, int i> struct is_prime {
   enum { prim = (p == 2) || (p%i) && is_prime<(i>2?p:0),
       i-1>::prim };
};

template<int i> struct Prime_print {
   Prime_print<i-1> a;
   enum { prim = is_prime<i, i-1>::prim };
   void f() { D<i> d = prim ? 1 : 0; a.f(); }
};

template<> struct is_prime<0,0> { enum { prim = 1 }; };
template<> struct is_prime<0,1> { enum { prim = 1 }; };

template<> struct Prime_print<1> {
   enum { prim = 0 };
   void f() { D<1> d = prim ? 1 : 0; };
};

#ifndef LAST
#define LAST 18
#endif
```

```
int main() {
   Prime_print<LAST> a;
   a.f();
}
```

Al compilarse este código con los parámetros `g++ -std=c++03 -c -fpermissive a.cpp` el compilador arroja, entre varios mensajes de error, las siguientes líneas:

```
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 17]': (...)
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 13]': (...)
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 11]': (...)
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 7]': (...)
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 5]': (...)
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 3]': (...)
a.cpp: In instantiation of 'void Prime_print<i>::f() [with int i
    = 2]': (...)
```

No debe ignorarse el hecho de que en la mayoría de los modelos computacionales no es siquiera posible introducir los conceptos necesarios para estas labores, como es el caso del cálculo λ, el cual, como ya se ha visto, es comúnmente concebido como la fundación teórica de los lenguajes de programación funcionales y es una teoría únicamente extensional, incapaz de dar cuenta de sus mismos objetos.

La autointerpretación en el entorno de los lenguajes de programación dota de la capacidad necesaria para que estos puedan reproducirse a sí (e.g., bootstrapping). Tradicionalmente, esto se realiza en un metanivel desde el cual se hace un tratamiento del código como texto plano, realizándose el etiquetado de sus partes para poder producir una estructura de datos capaz de ser analizada.

Ejemplo 5.3.2. Dentro de la plataforma .NET existen diversos compiladores y herramientas open-source para el análisis y optimización de código englobadas bajo el proyecto "Roslyn" [54, 55]. A través de ellas es posible obtener programas como árboles sintácticos, examinar la lista de métodos que contienen, las paqueterías necesarias para su compilación, etc. Todo esto se

logra a partir del tratamiento de los programas como cadenas de texto y su análisis por medio de diversas implementaciones de lexers para los lenguajes en cuestión (C#, VisualBasic).

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(<file>.cs);
var root = (CompilationUnitSyntax)tree.GetRoot();
var members = root.Members.ToList();
var firstMethod = (MethodDeclarationSyntax)root.Members[0];
var argsParameter = firstMethod.ParameterList.Parameters[0];
```

Una de las implicaciones que participan en la formalización de un cálculo como el expuesto en este capítulo es aquella que, al facilitar la identificación de los programas con las formas normales cerradas del lenguaje, vuelve asequible el análisis de programas y su optimización dentro del propio lenguaje, sin la necesidad del hirsuto proceso de etiquetamiento necesario en lenguajes no intensionales. Pudiendo identificar a los programas con estas formas, se muestra el enorme potencial que otorga la capacidad de factorización de los elementos del lenguaje, y gana relevancia dentro del análisis y optimización de programas en tanto que los programas son, a fin de cuentas, la codificación de una función que será manipulada por un compilador antes de producir un ejecutable y, al tener la capacidad de examinar su estructura desde dentro del lenguaje, no existe necesidad alguna de incursionar en la implementación de ningún metalenguaje que permita el análisis correspondiente.

Poco se ha desarrollado en torno a la capacidad de análisis que tiene el cálculo λ**SF** a pesar de existir resultados que muestran el potencial del cálculo **SF** [56] en el análisis estático de programas, herramienta común para la optimización de código por medio de técnicas como el desdoblamiento de constantes, eliminación de subexpresiones o propagación de constantes. Aunque el desarrollo y demostraciones de correctud y solidez respecto a la formulación de cualquier especie de análisis estático para el cálculo λ**SF** están fuera del alcance de este trabajo que pretende ser una breve introducción a modelos computacionales capaces de expresar funciones intensionales, presentamos en forma breve las formulaciones existentes para el análisis de control de flujo tanto para el cálculo λ como para el cálculo **SF**, ilustrando el actual panorama en torno a una de las técnicas más populares de análisis estático en las formalizaciones que fungen como pilares del cálculo λ**SF**, con el fin de sentar las bases sobre las cuales es posible extender este análisis.

5.3.1. Análisis de control de flujo

Es fácil ilustrar a qué nos referimos cuando hablamos de un análisis estático si pensamos en lo que popularmente es visto como la mayor utilidad que otorgan los sistemas de tipos dentro de lenguajes de programación. Por lo general, la gente en el gremio de la programación, ve a los tipos como abstracciones que representan una aproximación al comportamiento de un programa, de manera que permiten a los programadores (o quienquiera que esté frente a la codificación del programa en un lenguaje tipado) saber la forma de lo que, después de una ejecución satisfactoria, está almacenado o ha resultado como parte de una rutina (un número de punto flotante, un número entero, un apuntador a memoria, etc.).

Ejemplo 5.3.3. En tanto que Haskell es un lenguaje tipado de manera estática, su compilador provee de procesos mediante los cuales puede garantizar seguridad a partir de los tipos, es decir, las funciones, durante tiempo de ejecución, almacenarán y computarán el tipo de dato especificado y ningún otro.

```
exp :: Integer -> Integer -> Integer
exp den x =
   let den2 = div den 2
      (int,frac) = divMod (x + den2) den
      expFrac = expSmall den (frac-den2)
   in  case compare int 0 of
      EQ -> expFrac
      GT -> powerAssociative (mul den) expFrac (eConst den)  int
      LT -> powerAssociative (mul den) expFrac (recipEConst den)
          (-int)
```

Ejemplo 5.3.4. En contraste con Haskell, Javascript es un lenguaje con tipado débil, lo que no obliga a las implementaciones de su compilador a establecer ningún tipo de tipado "correcto" en sus programas, razón por la que es posible obtener errores en el uso de tipos de datos distintos durante el tiempo de ejecución, e.g., es posible correr un programa que multiplique una cadena con un número, lo que resulta en un error durante la ejecución, mas no en tiempo de compilación, como ocurriría en cualquier lenguaje fuertemente tipado.

```
var a = 5;
var b = "abc";
var c = a + b;
```

Uno de los análisis estáticos más populares que prescinde de la utilización de tipos dentro de un lenguaje es el análisis de control de flujo (CFA por sus siglas en inglés), el cual busca aproximar el orden en el que las expresiones de un programa son evaluadas con el afán de computar todas las funciones a las que es posible llegar a partir de una expresión. Para ello, es necesario etiquetar a cada uno de los fragmentos del programa analizado, como ilustramos con las reglas de esta clase de análisis para el cálculo λ puro.

Existen múltiples variaciones del análisis de control de flujo, la que presentamos aquí es la más simple de ellas y acaso uno de los métodos de análisis de programas más sencillos que se pueden hallar en la literatura. Forma parte de una familia de análisis que toman en consideración diversos aspectos de los programas analizados; nosotros hacemos uso de la variación que no toma en cuenta ninguna clase de información sobre el contexto del programa (de ahí el numeral cero que precede a las siglas en inglés, CFA). Este tipo de análisis es trivial para lenguajes de primer orden, i.e., aquellos donde las funciones no pueden aparecer como valores.

0-CFA en el cálculo λ

Al analizar una expresión del cálculo λ, buscamos aproximar los valores a los que es posible que esta evalúe; para ello iniciamos por asignar a cada subexpresión e una etiqueta $l \in \mathbb{N}$, de forma que cada una de ellas obtenga un identificador único. Las subexpresiones a etiquetar son las variables, abstracciones y aplicaciones:

$$e ::= x^l \mid \lambda^l x.e_1 \mid (e_1 e_2)^l$$

El análisis busca encontrar una función $\tilde{\mathcal{C}} : \mathcal{L} \to \tilde{p}$ que para cada etiqueta $l \in \mathcal{L} \subseteq \mathbb{N}$ aproxima el conjunto de valores $\tilde{p}$ a los que puede evaluar. Asimismo, se encuentra una función $r : \mathcal{V} \to \tilde{p}$ que va de cada variable $x \in \mathcal{V}$ al conjunto de valores a los que esta puede estar ligada.

Se genera un conjunto de restricciones sobre estas dos funciones, $\tilde{\mathcal{C}}$ y r, por recursión en la estructura del programa p, aplicando la función $\Gamma[\![x]\!]_e : \mathcal{E} \to \gamma$ que para cada expresión $e \in \mathcal{E}$ regresa una restricción γ:

$$\begin{aligned}
\Gamma[\![x]\!]_e &= \{r(x) \subseteq \tilde{\mathcal{C}}(l)\} && \text{si } x \equiv x^l \\
\Gamma[\![x]\!]_e &= \{l \in \tilde{\mathcal{C}}(l)\} \cup \Gamma[\![e_1]\!]_e && \text{si } x \equiv \lambda^l x.e_1 \\
\Gamma[\![x]\!]_e &= \Gamma[\![e_1^{l_1}]\!]_e \cup \Gamma[\![e_2^{l_2}]\!]_e \\
&\quad \cup \{l' \in \tilde{\mathcal{C}}(l_1) \Rightarrow \tilde{\mathcal{C}}(l_2) \subseteq r(x) | \delta(e, l') = \lambda^{l'} x.e_0^{l_0}\} && \text{si } x \equiv (e_1^{l_1} e_2^{l_2})^l \\
&\quad \cup \{l' \in \tilde{\mathcal{C}}(l_1) \Rightarrow \tilde{\mathcal{C}}(l_0) \subseteq \tilde{\mathcal{C}}(l) | \delta(e, l') = \lambda^{l'} x.e_0^{l_0}\}
\end{aligned}$$

Donde δ es una función que devuelve la expresión dentro de e asociada a la etiqueta l.

Estas restricciones pueden resolverse de forma iterativa empezando con conjuntos vacíos de aproximaciones a las soluciones para cada una de las etiquetas l y variables x. Estos conjuntos serán actualizados en cada paso iterativo agregando las etiquetas correspondientes para satisfacer las restricciones hasta llegar a un punto fijo en el que nuevas iteraciones no agreguen ninguna etiqueta extra. Al sólo haber un número finito de etiquetas y consistir el algoritmo en agregar estas a los conjuntos de aproximación en cada paso, está garantizado que eventualmente se llega a dicho punto fijo. Así, iniciamos con:

$$\forall l \in \mathcal{L}(\tilde{\mathcal{C}_0}(l) = \emptyset)$$
$$\forall x \in \mathcal{V}(r_0(x) = \emptyset)$$

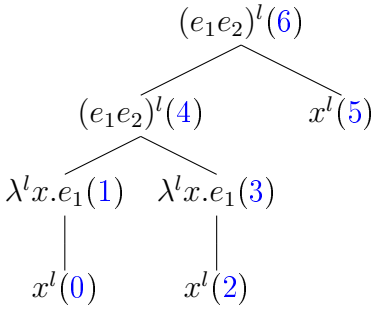
Y en cada paso iterativo i hacemos lo siguiente:

$$\begin{aligned}
\tilde{\mathcal{C}}_{i+1}(l) &= \tilde{\mathcal{C}}_i(l) \\
&\quad \cup \{l | (l \in \tilde{\mathcal{C}}(l)) \in \Gamma[\![e]\!]_e\} \\
&\quad \cup \bigcup \{r_i(x) | (r(x) \subseteq \tilde{\mathcal{C}}(l)) \in \Gamma[\![e]\!]_e\} \\
&\quad \cup \bigcup \{\tilde{\mathcal{C}}_i(l_0) | (l' \in \tilde{\mathcal{C}}(l_1) \Rightarrow \tilde{\mathcal{C}}(l_0) \subseteq \tilde{\mathcal{C}}(l)) \in \Gamma[\![e]\!]_e \wedge l' \in \tilde{\mathcal{C}}_i(l_1)\} \\
r_{i+1}(x) &= r_i(x) \\
&\quad \cup \bigcup \{\tilde{\mathcal{C}}_i(l_2) | (l' \in \tilde{\mathcal{C}}(l_1) \Rightarrow \tilde{\mathcal{C}}(l_2) \subseteq r(x)) \in \Gamma[\![e]\!]_e \wedge \tilde{\mathcal{C}}_i(l_1)\}
\end{aligned}$$

Ejemplo 5.3.5. Tomemos como ejemplo el análisis 0-CFA de la siguiente expresión, que aplica la función identidad a sí misma, y posteriormente una variable c a esta:

$$(\lambda a.a)(\lambda b.b)c \equiv [((\lambda a.a)(\lambda b.b))c]$$

El etiquetado se realiza conforme al orden que establece el regreso en la recursión del árbol generado por la búsqueda a profundidad en el programa p:



De forma que obtenemos la siguiente expresión etiquetada:

$$[((\lambda^1 a.a^0)(\lambda^3 b.b^2))^4 c^5]^6$$

Generamos ahora el conjunto de restricciones sobre $\tilde{\mathcal{C}}$ y r:

$$\begin{aligned}
\Gamma[\![p]\!]_p &= \Gamma[\![((\lambda^1 a.a^0)(\lambda^3 b.b^2))^4]\!]_p \\
&\cup \Gamma[\![c^5]\!]_p \\
&\cup \{l' \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(5) | \delta(e, l') = \lambda^{l'} x.e_0^{l_0}\} \\
&\cup \{l' \in \tilde{\mathcal{C}}(l_0) \Rightarrow \tilde{\mathcal{C}}(6) | \delta(e, l') = \lambda^{l'} x.e_0^{l_0}\} \\
&= \Gamma[\![((\lambda^1 a.a^0)(\lambda^3 b.b^2))^4]\!]_p \\
&\cup \Gamma[\![c^5]\!]_p \\
&\cup \{1 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(5) \subseteq r(a), 3 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(5) \subseteq r(b)\} \\
&\cup \{1 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(2) \subseteq \tilde{\mathcal{C}}(6), 3 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(2) \subseteq \tilde{\mathcal{C}}(6)\} \\
\Gamma[\![((\lambda^1 a.a^0)(\lambda^3 b.b^2))^4]\!]_p &= \Gamma[\![\lambda^1 a.a^0]\!]_p \\
&\cup \Gamma[\![\lambda^3 b.b^2]\!] \\
&\cup \{1 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(3) \subseteq r(a), 3 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(3) \subseteq r(b)\} \\
&\cup \{1 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(0) \subseteq \tilde{\mathcal{C}}(4), 3 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(2) \subseteq \tilde{\mathcal{C}}(4)\} \\
\Gamma[\![\lambda^1 a.a^0]\!]_p &= \{1 \in \tilde{\mathcal{C}}(1)\} \cup \Gamma[\![a^0]\!]_p \\
\Gamma[\![a^0]\!]_p &= \{r(a) \subseteq \tilde{\mathcal{C}}(0)\} \\
\Gamma[\![\lambda^3 b.b^2]\!]_p &= \{3 \in \tilde{\mathcal{C}}(3)\} \cup \Gamma[\![b^2]\!]_p \\
\Gamma[\![b^2]\!]_p &= \{r(b) \subseteq \tilde{\mathcal{C}}(2)\} \\
\Gamma[\![c^5]\!]_p &= \{r(c) \subseteq \tilde{\mathcal{C}}(5)\}
\end{aligned}$$

De manera que el conjunto de restricciones queda de la siguiente forma:

$$\begin{aligned}
\Gamma[\![p]\!]_p = \{&1 \in \tilde{\mathcal{C}}(1), 3 \in \tilde{\mathcal{C}}(3), \\
&r(a) \subseteq \tilde{\mathcal{C}}(0), r(b) \subseteq \tilde{\mathcal{C}}(2), r(c) \subseteq \tilde{\mathcal{C}}(5),
\end{aligned}$$

$$
\begin{aligned}
&1 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(5) \subseteq r(a),\\
&1 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(3) \subseteq r(a),\\
&3 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(5) \subseteq r(b),\\
&3 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(3) \subseteq r(b),\\
&1 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(2) \subseteq \tilde{\mathcal{C}}(6),\\
&1 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(0) \subseteq \tilde{\mathcal{C}}(4),\\
&3 \in \tilde{\mathcal{C}}(4) \Rightarrow \tilde{\mathcal{C}}(2) \subseteq \tilde{\mathcal{C}}(6),\\
&3 \in \tilde{\mathcal{C}}(1) \Rightarrow \tilde{\mathcal{C}}(2) \subseteq \tilde{\mathcal{C}}(4)\}
\end{aligned}
$$

Utilizando el proceso iterativo establecido en 5.3.1 llegamos, después de ocho iteraciones, al etiquetado de cada una de las restricciones $\mathcal{C}_i(l)$ donde tenemos, entre ellas, tanto a $\mathcal{C}(5) = \{4\}$ como $\mathcal{C}(6) = \{5\}$, mismas que pueden interpretarse como que la expresión etiquetada por 6 evalúa a lo que se tenga en la expresión dada por la etiqueta 5 y a que la expresión dada por la etiqueta 4 evalúa a la expresión etiquetada por 3, respectivamente. Es decir, la expresión total evalúa a c en tanto que $((\lambda a.a)(\lambda b.b))$ queda determinada por $\lambda b.b$.

0-CFA en el cálculo $\boldsymbol{SF}$

En tanto que existe un homomorfismo entre el cálculo $\boldsymbol{SK}$ y el cálculo λ (podemos enunciar gran parte del análisis de control de flujo del cálculo $\boldsymbol{SF}$), que extiende a la lógica combinatoria incorporando un combinador que factoriza a los términos, explorando la manera en la que se da la traducción de los términos al cálculo λ a partir del cálculo $\boldsymbol{SK}$.

Si bien este método funciona, pues $\Lambda \vdash x \Leftrightarrow \mathcal{CL} \vdash x$, la interpretación de las restricciones y su solución se vuelven confusas al ver los términos del cálculo $\boldsymbol{SK}$ a la luz de su equivalencia en otro sistema, ya que ello incorpora necesariamente un mayor número de variables, inexistentes en la representación dada en la lógica combinatoria al esta prescindir de variables ligadas, lo que de forma intrínseca conlleva un mayor uso de las reglas de reducción, pues por cada reducción débil $\rightarrow_w$ se incorporan dos o tres β-reducciones (por cada combinador $\boldsymbol{K}$ o $\boldsymbol{S}$, respectivamente).

Analizando la forma en la que se reducen los términos al accionar las reglas de reducción del combinador $\boldsymbol{S}$, es fácil notar la necesidad de incorporar etiquetas inexistentes al análisis de una expresión dada, pues la motivación principal del análisis de control de flujo, así como la principal función del etiquetado en este, es poder rastrear de dónde provienen los términos a los que se han reducido las expresiones para poder determinar a partir de ello los

resultados finales en la evaluación de un programa de manera estática, razón por la cual es necesario indicar de qué instancia de $\boldsymbol{S}$ es que se ha reducido una expresión, por ello la necesidad de nuevas etiquetas:

$$(((\boldsymbol{S}^{l_0} M^{l_1})^{l_2} N^{l_3})^{l_4} X^{l_5})^{l_6} \to ((M^{l_1} X^{l_5})^{l_{0\boldsymbol{S}L}} (N^{l_3} X^{l_5})^{l_{0\boldsymbol{S}R}})^{l_{0\boldsymbol{S}}}$$

Lo mismo ocurre con el factorizador $\boldsymbol{F}$ cuando este actúa sobre compuestos:

$$\begin{aligned} (((\boldsymbol{F}^{l_0} M^{l_1})^{l_2} N^{l_3})^{l_4} X^{l_5})^{l_6} &\to X^{l_5} && \text{si } M = \boldsymbol{S} \text{ o } M = \boldsymbol{F} \\ (((\boldsymbol{F}^{l_3} (P^{l_0} Q^{l_1})^{l_2})^{l_4} N^{l_5})^{l_6} X^{l_7})^{l_8} &\to ((X^{l_7} P^{l_0})^{l_{3\boldsymbol{F}M}} Q^{l_1})^{l_{3\boldsymbol{F}}} \end{aligned}$$

Como se puede notar, la idea de las etiquetas únicamente como elementos del conjunto de los números naturales tiene que ser extendida para poder indicar la procedencia de los elementos que resultan de la reducción de los combinadores elementales del cálculo. Aunque es posible mantener una notación que dependa únicamente en etiquetas de $\mathbb{N}$ estableciendo una convención en torno al etiquetado de las expresiones que nos ayude a sentar una codificación conveniente, siguiendo la exposición original de Martin Lester [57] alrededor del análisis de control de flujo sin contexto para el cálculo $\boldsymbol{SF}$, nos limitamos a expandir las posibilidades de etiquetado para abarcar etiquetas que indiquen si se proviene de un combinador dentro de la reducción de un factorizador ($l_{n\boldsymbol{F}M}$), la aplicación de un factorizador ($l_{n\boldsymbol{F}}$), la aplicación izquierda resultante de la reducción de un substituidor ($l_{n\boldsymbol{S}L}$), la aplicación derecha resultante de la reducción de un substitutidor ($l_{n\boldsymbol{S}R}$) o de la aplicación de un substituidor ($l_{n\boldsymbol{S}}$), $n \in \mathbb{S}$.

Un punto clave en el análisis de control de flujo de las expresiones de un cálculo que proviene de la lógica combinatoria, tal y como es el cálculo $\boldsymbol{SF}$, es que las restricciones que se desprenden de las expresiones capaces de ser reducidas únicamente sean agregadas cuando los combinadores capaces de activar la reducción cumplan con poder obtener los argumentos necesarios, i.e., no deben agregarse restricciones para los combinadores parcialmente aplicados.

Para dar cuenta de la activación en la reducción de un combinador, se introduce una función $\varphi : \mathcal{L} \to \{\mathcal{F}, \mathcal{V}\}$ que para una etiqueta $n \in \mathbb{N}$ evalúe a $\mathcal{V}$ cuando las restricciones del combinador correspondiente, $\boldsymbol{S}^n$ o $\boldsymbol{F}^n$ en nuestro caso, estén activas.

Incorporando las nuevas formas de etiquetado mencionadas, así como agregando etiquetado para los argumentos primero (0), segundo (1), tercero (2), y el resultado de la reducción de un combinador $\boldsymbol{S}$ o $\boldsymbol{F}$ (3) que

ha recibido los tres argumentos necesarios para accionar la reducción, extendemos el conjunto de etiquetas posibles para las expresiones del cálculo, $\mathcal{L} \subseteq \mathbb{N} \cup \{\boldsymbol{S}0, \boldsymbol{S}1, \boldsymbol{S}2, \boldsymbol{S}3, \boldsymbol{S}L, \boldsymbol{S}R, \boldsymbol{F}0, \boldsymbol{F}1, \boldsymbol{F}2, \boldsymbol{F}3, \boldsymbol{F}M\}$.

Tomando a las expresiones etiquetables del cálculo $\boldsymbol{SF}$ como:

$$e ::= \boldsymbol{S}^l \mid \boldsymbol{F}^l \mid (e_1 e_2)^l \mid x^l$$

podemos ahora introducir las reglas de obtención de restricciones establecidas por Lester [57]. Dichas reglas se distancian del tratamiento que hicimos del análisis de control de flujo para el cálculo λ, al verse aquí como condiciones de derivabilidad y no como pasos intermedios para un algoritmo que de forma iterativa dé un etiquetado final de las expresiones que conforman al programa inicial.

Lester une bajo Γ todas las restricciones resultantes de $\tilde{\mathcal{C}}$ y r. En su notación, $\Gamma \vDash e$ denota que Γ es una solución para las restricciones generadas por e, mientras que $\Gamma(l)$ denota a la expresión que está etiquetada por l.

Sin mayor preámbulo reproducimos aquí las reglas para la generación de restricciones establecidas en [57] para el análisis de control de flujo sin contexto de los programas dentro del cálculo $\boldsymbol{SF}$.

$$
\begin{aligned}
\Gamma, \varphi \vDash \boldsymbol{S}^l &\Leftrightarrow \boldsymbol{S}_0^l \in \Gamma(l) \wedge (\varphi(l) \Rightarrow \Gamma, \varphi \vDash e_{\boldsymbol{S}^l}) \\
\Gamma, \varphi \vDash \boldsymbol{F}^l &\Leftrightarrow \boldsymbol{F}_0^l \in \Gamma(l) \\
&\quad \wedge \varphi(l) \Rightarrow (\exists l_0 (\boldsymbol{S}_0^{l_0} \in \Gamma(l\boldsymbol{F}0) \vee \boldsymbol{F}_0^{l_0} \in \Gamma(l\boldsymbol{F}0))) \Rightarrow \Gamma(l\boldsymbol{F}1) \subseteq \Gamma(l\boldsymbol{F}3) \\
&\quad \wedge \varphi(l) \Rightarrow (\exists l_0 (\boldsymbol{S}_0^{l_0} \in \Gamma(l\boldsymbol{F}0) \vee \\
&\qquad \boldsymbol{F}_1^{l_0} \in \Gamma(l\boldsymbol{F}0) \vee \boldsymbol{F}_2^{l_0} \in \Gamma(l\boldsymbol{F}0))) \Rightarrow \Gamma, \varphi \vDash e_{\boldsymbol{F}^l} \wedge \\
&\qquad \forall (a^{l_1} b^{l_2}) \in \Gamma(l\boldsymbol{F}0)(\Gamma(l_1) \subseteq \Gamma(l\boldsymbol{F}L) \wedge \Gamma(l_2) \subseteq \Gamma(l\boldsymbol{F}R)) \\
\Gamma, \varphi \vDash x^l &\Leftrightarrow \mathcal{V} \\
\Gamma, \varphi \vDash (e_1^{l_1} e_2^{l_2})^{l_3} &\Leftrightarrow \Gamma, \varphi \vDash e_1 \wedge \Gamma, \varphi \vDash e_2 \\
&\quad \wedge \exists (a^{l_4} b^{l_5}) \in \Gamma(l_3)(\Gamma(l_1) \subseteq \Gamma(l_4) \wedge \Gamma(l_2) \subseteq \Gamma(l_5)) \\
&\quad \wedge \forall \boldsymbol{S}_0^l \in \Gamma(l_1)(\Gamma(l_2) \subseteq \Gamma(l\boldsymbol{S}0) \wedge \boldsymbol{S}_1^l \in \Gamma(l_3)) \\
&\quad \wedge \forall \boldsymbol{S}_1^l \in \Gamma(l_1)(\Gamma(l_2) \subseteq \Gamma(l\boldsymbol{S}1) \wedge \boldsymbol{S}_2^l \in \Gamma(l_3)) \\
&\quad \wedge \forall \boldsymbol{S}_2^l \in \Gamma(l_1)(\Gamma(l_2) \subseteq \Gamma(l\boldsymbol{S}2) \wedge \Gamma(l\boldsymbol{S}3) \subseteq \Gamma(l_3) \wedge \varphi(l)) \\
&\quad \wedge \forall \boldsymbol{F}_0^l \in \Gamma(l_1)(\Gamma(l_2) \subseteq \Gamma(l\boldsymbol{F}0) \wedge \boldsymbol{F}_1^l \in \Gamma(l_3)) \\
&\quad \wedge \forall \boldsymbol{F}_1^l \in \Gamma(l_1)(\Gamma(l_2) \subseteq \Gamma(l\boldsymbol{F}1) \wedge \boldsymbol{F}_2^l \in \Gamma(l_3)) \\
&\quad \wedge \forall \boldsymbol{F}_2^l \in \Gamma(l_1)(\Gamma(l_2) \subseteq \Gamma(l\boldsymbol{F}2) \wedge \Gamma(l\boldsymbol{F}3) \subseteq \Gamma(l_3) \wedge \varphi(l))
\end{aligned}
$$

Remitimos al lector al artículo original para un desarrollo detallado de cada una de las expresiones que aparecen en las reglas de generación de restricciones. Sin embargo, cabe destacar un par de aclaraciones en torno a dos puntos que se presentan repetidamente en la notación que usa Lester y que la esclarecen en gran medida. La primera anotación es concerniente a la noción intuitiva de $\boldsymbol{S}_0^l \in \Gamma(L)$, que debe interpretarse como que el combinador etiquetado $\boldsymbol{S}^l$ puede ocurrir dentro de una expresión etiquetada por L (similar al uso de δ que hicimos para el cálculo λ). Esta interpretación se extiende a expresiones similares donde $\boldsymbol{S}_x^l$ aparece con un subíndice distinto de cero. Estos subíndices indican los argumentos implicados en la construcción de una expresión reducible, e.g., $\boldsymbol{S}_1^l \in \Gamma(L)$ hace referencia al término que al aplicar un argumento al combinador $\boldsymbol{S}^l$ puede aparecer en la expresión etiquetada por L. Esto es análogo para distintos subíndices de los combinadores y para el combinador $\boldsymbol{F}$. La segunda anotación atañe a los valores dados por $\Gamma(l\boldsymbol{S}0)$, los cuales aproximan las expresiones que pueden ocurrir como primer argumento para un combinador $\boldsymbol{S}^l$; esto quizá no resulta tan confuso, en tanto que, como ya mencionamos, en la notación de la que se hace uso, $\Gamma(l)$ denota a la expresión etiquetada por l y, como se apuntó antes, el etiquetado fue extendido para abarcar formas en las que podamos referirnos a los argumentos que recibe un combinador. De forma similar, $\Gamma(l\boldsymbol{S}1)$ aproxima los valores que pueden aparecer como segundo argumento del combinador. Esto es análogo para el factorizador $\boldsymbol{F}$.

Capítulo 6

Conclusiones

En las páginas de este trabajo mostramos las bases de uno de los modelos computacionales más reconocidos a la fecha, el cálculo λ, que se ha establecido como el modelo canónico para el diseño de numerosos lenguajes de programación (basta contar a todos los lenguajes que siguen la venia funcional).

La expresividad de este modelo fue ejemplificada, junto a la de la lógica combinatoria (formulaciones equivalentes), ilustrando después de la introducción de sus conceptos básicos la manera en la que ambas formalizaciones son capaces de definir funciones de orden mayor y computar toda función recursiva sobre los números naturales.

Las formulaciones que establecieron la naturaleza de lo computable durante la primera mitad del siglo XX sentaron también, en forma consensual [4, 58], que todas las preguntas posibles en torno a los modelos de la computación podían reducirse a las preguntas respecto a las funciones numéricas calculables, las cuales fueron examinadas por diversos modelos propuestos, entre los más famosos, el cálculo λ de Church y las máquinas de Turing, capaces de determinar a la misma clase de funciones numéricas.

Sin tomar una postura crítica en torno a esta aseveración, los desarrollos mostrados aquí, a saber, el cálculo $\boldsymbol{SF}$ y el cálculo $\lambda\boldsymbol{SF}$, no abonan nada nuevo a los fundamentos de la computación ni otorgan una razón sólida por la cual debiésemos considerarlos como base para nuevos lenguajes de programación en lugar de los que actualmente son utilizadas en las decisiones en torno al diseño de los lenguajes, en tanto que, tal y como muestra Landin en su famoso artículo [59], se ve al cálculo λ como una formulación que ya materializa la mayor expresividad posible en torno a lo que es computable. Además, si bien los dos cálculos desarrollados en este trabajo logran resolver ciertas tensiones existentes entre lenguajes concebidos bajo distintos paradigmas al poseer formas de emparejamiento de patrones más expresivas que

las que se hallan hoy día en los lenguajes funcionales más populares, logrando dar un tratamiento natural de los objetos vistos como elementos de una clase, así como dar formas de acceso sencillos a sus miembros constitutivos, diversas soluciones alternativas han sido propuestas en las comunidades que utilizan estos lenguajes puramente funcionales, la mayoría de ellas abusando de sus características o haciendo uso de estructuras sofisticadas como pueden ser las máquinas de Mealey [60, 61]. Casos similares se encuentran en las comunidades enfocadas en lenguajes orientados a objetos para hacer uso de características propias de los lenguajes funcionales.

Separándonos de esta postura que ve en el desarrollo de Church la posibilidad de alcanzar la máxima expresividad en torno a lo que es computable, adoptamos una novedosa, uniéndonos a Barry Jay [39]: establezcamos a $\varepsilon(\mathbb{N})$ como la clase de las funciones recursivas que van a los números naturales. Por la tesis de Church, tenemos que Λ es completo para $\varepsilon(N)$. Esto es, si tomamos al cálculo λ, este es capaz de definir a cualquier función sobre los números naturales representados a través de los numerales de Church 2.4 $f : \Lambda \to \mathbb{N}$. Cambia esta situación cuando analizamos la completud de este respecto a las funciones representables dentro del mismo cálculo λ; es decir, $\varepsilon(\Lambda)$ no es completo pues, como vimos anteriormente, no es posible definir dentro de este función alguna que pueda determinar la igualdad de sus formas normales. En este aspecto, el enfoque adoptado por Turing sí logra ser completo, en tanto es posible definir en sus términos toda función que vaya a una máquina de Turing $f : T_A \to T_A$, mas no logra, como el cálculo λ, representar funciones de orden mayor.

Los dos nuevos modelos aquí expuestos, el cálculo $\boldsymbol{SF}$ y el cálculo $\lambda\boldsymbol{SF}$, logran, tal y como el cálculo λ, dar cuenta de funciones de orden mayor, a la vez que, análogamente a las máquinas de Turing, ser completos en sí, como puede verse fácilmente a partir de la posibilidad de definir a una función que determine la igualdad estructural de sus formas normales 4.4.

Si bien el cálculo λ ha sido visto como un sistema formal que encierra la mayor expresividad posible en torno a las funciones computables, lo ha sido en tanto que el análisis de su expresividad se ha limitado a su capacidad de representar a las funciones recursivas numéricas, dando una falsa impresión de los límites de lo expresable dentro de lenguajes diseñados en el espíritu de lo "puramente funcional".

La expresividad de las formalizaciones aquí mostradas puede compararse con la de otros modelos de varias formas. Nosotros nos hemos centrado en su capacidad para definir funciones intensionales, lo que dota a estos sistemas de reescritura de la capacidad de actuar sobre cualquier forma normal representada en ellos, abriéndonos a diversas posibilidades en las áreas del diseño de lenguajes, análsis y optimización de programas, removiendo las capas de

abstracción usualmente utilizadas en los análisis que hacen los compiladores de los diversos lenguajes.

La clase de funciones capaces de ser definidas dentro de los dos cálculos aquí formulados es mayor que aquellas correspondientes a otras formulaciones como la de Church o Turing y, en este sentido, logran ser sistemas formales más expresivos.

Hay varios ejes sobre los cuales podríamos dirigir el trabajo futuro a partir del análisis hecho en este trabajo:

- La formulación de un lenguaje funcional capaz de autointerpretación, así como de definir funciones de emparejamiento de patrones de manera general sobre cualquier estructura de datos, evitando la verbosidad que se encuentra actualmente en varios lenguajes.

- Un análisis en torno a la aritmetización de los programas hechos en estos cálculos y el desarrollo de técnicas por las cuales aritmetizar a los programas identificados con las formas normales cerradas de estas formalizaciones.

- Añadir un sistema de tipos que pueda facilitar el análisis estático de los programas escritos ya sea de forma directa en estos cálculos, o en lenguajes basados en ellos.

- Implementación de diversos algoritmos de análisis y optimización de programas dentro de las mismas formalizaciones, explotando la naturaleza intensional de ellas.

- Desarrollar el análisis de control de flujo para el cálculo $\lambda\boldsymbol{SF}$.

- Traducir la notación utilizada por Lester para el 0-CFA a una que permita un tratamiento mecánico más intuitivo, así como desarrollar el análisis de control de flujo haciendo uso de contextos (k-CFA) para mejorar la precisión del análisis.

- Elaborar el desarrollo de diversas técnicas de análisis que puedan utilizarse en paralelo con el análisis de control de flujo.

- Implementar tanto el cálculo $\boldsymbol{SF}$ como el cálculo $\lambda\boldsymbol{SF}$ en un lenguaje de alto nivel que permita el tratamiento de funciones intencionales a partir de técnicas de metaprogramación y en el que se puedan utilizar las técnicas de análisis y optimización de programas anteriormente mencionadas.

Apéndice A

El asistente de pruebas Coq

Que uno de los trabajos más notables en tiempos recientes encaminado a establecer los fundamentos para una ciencia, como lo es el enorme trabajo hecho por Donald Knuth [62], lleve por título "The Art of Computer Programming" y no "The Science of Computer Programming" nos dice mucho respecto al estatus en el que se encuentra actualmente la computación vista como disciplina.

Fluctuante entre la ciencia, la ingeniería y la matemática, la computación siempre ha tenido, dentro de las áreas de investigación en torno suyo, un círculo esperanzado de poder establecer métodos mediante los cuales se pueda diseñar software en el que se tenga la misma confianza que en, digamos, los resultados en torno a experimentos físicos publicados en las distintas revistas especializadas, buscando aliar, como en esta última (o al menos disipando ciertos problemas en la conformación de la computación como una ciencia y la programación como una práctica ingenieril formal) los procesos propios de su práctica con los fundamentos teóricos de ella [63].

El interés que existe en poder contar con herramientas capaces de asistirnos en la verificación de la correctud de los programas que hacemos se ve ilustrada por los grupos de investigación que emergieron y maduraron durante la segunda mitad del siglo XX, cuyo trabajo se enfocó principalmente en la mecanización de demostraciones de diversos teoremas en la matemática.

Aunque se ha puesto en entredicho el pragmatismo que pueda hallarse en el uso de un asistente de pruebas, ya sea en la demostración de teoremas matemáticos o en la verificación de programas, el estudio que hay sobre las bases teóricas y las capacidades prácticas de estas nuevas herramientas que pretenden fungir como réferi o guía en el quehacer de las personas dedicadas a distintas áreas relacionadas con la matemática es relativamente nuevo en comparación con, por poner ejemplo, las prácticas que hay alrededor del diseño, desarrollo e implementación de lenguajes de programación, área en

la que puede verse de forma sumamente tangible el progreso si comparamos a los primeros programas hechos en BASIC con programas actuales hechos en lenguajes modernos como Python o Scala.

No encontrando razón por la cual se pudiese dudar del enorme impacto que pueda tener un asistente de pruebas y creyendo que el rol que tomarán esta clase de herramientas dentro de las disciplinas afines a la matemática (y más específicamente, dentro de la computación) en el futuro cercano será cada vez mayor, varias definiciones y resultados establecidos a lo largo de los capítulos segundo y tercero de este trabajo fueron formalizados en el asistente de pruebas Coq y se encuentran disponibles en la siguiente liga: `http://github.com/mateoatr/sf`, o bien como documentos adjuntos a este trabajo.

Para los capítulos cuarto y quinto, cuyas definiciones y resultados se encuentran ya formalizados por los autores que inicialmente propusieron los cálculos $\boldsymbol{SF}$ y $\lambda\boldsymbol{SF}$, cuyo código puede encontrarse adjunto en el repositorio ya mencionado, el trabajo fue realizado en forma inversa, como puede notarse en resultados como 5.2.1, pasando de la formalización a la exposición "común" de la demostración.

Cabría añadir al trabajo futuro la necesidad de formalizar los mecanismos por los que nos es posible pasar de una prueba o definición dentro de un asistente de pruebas como lo es Coq a una prueba o definición en el sentido más común, así como pensar y establecer estos procesos en la dirección contraria.

Bibliografía

[1] Immanuel Kant. *Crítica de la razón pura*. Editorial Tecnos, 2002.

[2] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 2004.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[5] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[6] Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[7] John Sargeant. Uniting functional and object-oriented programming. *International Symposium on Object Technologies for Advanced Software*, 742:1–26, 1993.

[8] Martin Odersky and Tiark Rompf. Unifying functional and object-oriented programming with scala. *Communications Of The Acm*, 57(4):11. 76–86, 2014.

[9] Barry Jay and Thomas Given-Wilson. A combinatory account of internal structure. *The Journal of Symbolic Logic*, 76(3):807–826, 2011.

[10] Barry Jay. Programs as data structures in lambda sf-calculus. *Electronic Notes in Theoretical Computer Science*, 325:221 – 236, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).

[11] Raúl Rojas and Ulf Hashagen, editors. *The First Computers: History and Architectures*. MIT Press, Cambridge, MA, USA, 2000.

[12] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, 10th edition, 2012.

[13] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989.

[14] John Backus. The history of fortran i, ii, and iii. *SIGPLAN Not.*, 13(8):165–180, August 1978.

[15] Herbert Stoyan. Early lisp history (1956 - 1959). In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 299–310, New York, NY, USA, 1984. ACM.

[16] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[17] S.C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36:630–636, 1935.

[18] J. A. Amor Montaño. *Teoría de conjuntos para estudiantes de ciencias*. Las prensas de ciencias, 2da edition, 2011.

[19] A. M. Turing. Computability and lambda-definability. *The Journal of Symbolic Logic*, 2:153–163, 1937.

[20] Herbert B. Enderton. *A Mathematical Introduction to Logic*. New York: Academic Press, 1972.

[21] H. S. M. Coxeter. *Non-Euclidean Geometry*. Mathematical Association of America, 1998.

[22] Craig W. Curry H. B., Feys R. *Combinatory logic*. North-Holland Publishing Company, 1st edition, 1958.

[23] ALONZO CHURCH. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941.

[24] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[25] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 75:381–392, 1972.

[26] Fairouz Kamareddine. Reviewing the Classical and the de Bruijn Notation for Lambda Calculus and Pure Type Systems. *Journal of Logic and Computation*, 11(3):363–394, 06 2001.

[27] H. P. Barendregt. *The Lambda Calculus Its Syntax and Semantics*. Elsevier (Studies in logic and the foundations of mathematics : v. 103), 1st edition, 1981.

[28] Raymond Smullyan. *To Mock a Mockingbird*. Alfred A. Knpof, 1985.

[29] Moses Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:307–316, 1924.

[30] Benjamin Peirce. *Linear Associative Algebra*. Godfrey Lowell Cabot Science Library, 1st edition, 1870.

[31] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Verlag von louis nebert edition, 1879.

[32] Misak C. *The Cambridge Companion to Peirce*. Cambridge Companions to Philosophy. Cambridge University Press, 2004.

[33] H. Sheffer. A set of five independent postulates for Boolean algebras, with application to logical constants. *Trans. Amer. Math. Soc.*, 14(4):481–488, 1913.

[34] Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[35] Fokker J. The systematic construction of a one-combinator basis for lambda-terms. *Formal Aspects of Computing*, 4:776–780, 1992.

[36] Wolfengagen V. E. *Combinatory Logic in Programming*. Library of "JurInfoR", 2nd edition, 2003.

[37] Hindley J. R. Çağman N. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198:239–247, 1998.

[38] Hindley J. R. Combinatory reductions and lambda reductions compared. *Zeit. Math. Logik*, 23:169–180, 1977.

[39] Barry Jay. Intensional computation with higher-order functions. *Theoretical Computer Science*, 02 2019.

[40] Allen Harvey Poe, Edgar A. *The complete tales and poems of Edgar Allan Poe*. The Modern Library, 1938.

[41] Kesner Delia Jay Barry. Pure pattern calculus. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 100–114, Berlin, Heidelberg, 2006. Springer-Verlag.

[42] John T. Kearns. Combinatory logic with discriminators. *Journal of Symbolic Logic*, 34(4):561–575, 1969.

[43] B. Jay and J. Vergara. Growing a language in pattern calculus. In *2013 International Symposium on Theoretical Aspects of Software Engineering*, pages 233–240, July 2013.

[44] Jose Vergara Barry Jay. Conflicting accounts of λ-definability. *J. Log. Algebr. Meth. Program.*, 87:1–3, 2017.

[45] BARRY JAY and DELIA KESNER. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009.

[46] Tobias Nipkow. More church–rosser proofs. *Journal of Automated Reasoning*, 26(1):51–66, Jan 2001.

[47] Robert Pollack. Polishing up the tait-martin-löf proof of the church-rosser theorem, 1995.

[48] Masako Takahashi. Parallel reductions in lambda-calculus. *Journal of Symbolic Computation*, 7(2):113 – 123, 1989.

[49] Masako Takahashi. Parallel reductions in lambda-calculus. *Information and Computation*, 118(1):120 – 127, 1995.

[50] Vincent van Oostrom. Developing developments. *Theor. Comput. Sci.*, 175(1):159–181, March 1997.

[51] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.

[52] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[53] Peter Gottschling. *Discovering Modern C++: An Intensive Course for Scientists, Engineers, and Programmers*. Addison-Wesley Professional, 1st edition, 2015.

[54] Manish Vasani. *Roslyn Cookbook*. Packt Publishing, 2017.

[55] Jason Bock. *.NET Development Using the Compiler API*. Apress, 2016.

[56] Martin Lester. Control Flow Analysis for SF Combinator Calculus. *arXiv e-prints*, page arXiv:1512.03861, Dec 2015.

[57] Martin Lester. Control flow analysis for sf combinator calculus. 2015.

[58] S. C. Kleene. Lambda-definability and recursiveness. *Duke Math. J.*, (2):340–353, 06.

[59] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, March 1966.

[60] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739 – 751, 2000.

[61] Victor Cacciari Miraldo. Object oriented programming with monadic mealy machines. 2014.

[62] Donald E. Knuth. Computer programming as an art. *Commun. ACM*, 17(12):667–673, December 1974.

[63] Matti Tedre. Computing as a science: A survey of competing viewpoints. *Minds Mach.*, 21(3):361–387, August 2011.

[64] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, June 2012.

[65] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.