UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Una implementación polimórfica de ISWIM

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

P R E S E N T A :

Diego Murillo Albarrán

DIRECTOR DE TESIS:
Dr. Favio Ezequiel Miranda Perea

2017

Ciudad Universitaria, Cd. Mx.

1. Datos del Alumno
Murillo
Albarrán
Diego
55 2325 2202
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
411001896

2. Datos del Tutor
Dr.
Favio Ezequiel
Miranda
Perea

3. Datos del sinodal 1
Dra.
Lourdes del Carmen
González
Huesca

4. Datos del sinodal 2
M. en I.
Karla
Ramírez
Pulido

5. Datos del sinodal 3
M. en C.
Pilar Selene
Linares
Arévalo

6. Datos del sinodal 4
M. en C.
Noé Salomón
Hernandéz
Sanchéz

7. Datos del trabajo escrito
Una implementación polimórfica de ISWIM
161 p
2017

Agradecimientos

A mi familia y amigos, a mis profesores y compañeros, y sobre todo a la Universidad y a la Patria.

Índice general

Introducción

Escribir programas de computadora para resolver cierta tarea de manera eficiente y correcta es una de las labores finales en el proceso creativo de un científico de la computación, y desde los inicios de las primeras computadoras electrónicas en la década de 1950 se hizo notoria la necesidad de diseñar e implementar lenguajes de programación que faciliten esta tarea. Inicialmente el desarrollo de estos lenguajes se limitó al diseño de ensambladores que transformaran reglas mnemotécnicas a su correspondiente código de máquina; sin embargo, esta manera de ensamblar programas resultó problemática al requerir conocimiento profundo de la arquitectura de cada máquina por parte del programador. El siguiente paso natural era diseñar un lenguaje de alto nivel que permitiera al programador olvidarse de las características físicas de la máquina, y en el que fuera posible convertir un programa escrito en este lenguaje en una secuencia de instrucciones de máquina semánticamente equivalente mediante el uso de un metaprograma conocido como compilador. Inicialmente existía la duda de que fuera posible diseñar un compilador capaz de generar código tan eficiente como el que escribiría un programador experto mediante el uso de ensambladores; fue a partir de este problema que el diseño e implementación de lenguajes de programación tomó relevancia como área de estudio de las ciencias de la computación.

A finales de la década de 1960 surgió el paradigma de programación estructurada a partir de la invención del concepto de *subrutinas*, que en cierta medida emulan al concepto de *función matemática*. Ambos conceptos permiten la reutilización del código al definir un mecanismo mediante el cual se puede invocar la ejecución de ciertas intrucciones sobre algunos argumentos de entrada con la finalidad de devolver un valor como resultado. La diferencia entre estas dos nociones es que una función matemática es *pura* en el sentido de que siempre que ésta se *mande llamar* con una combinación particular de argumentos devolverá exactamente el mismo resultado; mientras que en una subrutina esto no es necesariamente cierto ya que existe un estado global que puede ser modificado mediante efectos secundarios de la misma u otra subrutina y del cual dependa el valor del resultado final, por lo que invocar dos veces a la misma subrutina con los mismos argumentos de entrada puede devolver resultados distintos.

La existencia de efectos secundarios oculta el flujo de datos en un programa por lo que también dificulta el análisis matemático de éste; es por esta razón que se propone al paradigma funcional puro como una alternativa superior a la programación estructurada con efectos secundarios.

En este trabajo retomamos ISWIM, la familia de lenguajes de programación propuesta por Landin en 1969 [21], para desarrollar e implementar una instancia polimórfica de tal familia. El objetivo de este trabajo es exponer de manera precisa y clara las nociones fundamentales referentes al cálculo-λ puro y sus extensiones pertinentes, en aras de culminar en un cálculo y un sistema de tipos que puedan formar la base de un lenguaje de programación estáticamente tipado mediante inferencia polimórfica de tipos.

En la primera parte de este trabajo desarrollaremos las nociones básicas referentes al cálculo-λ mostrando su sintáxis y semántica, así como su poder computacional al mostrar cómo definir funciones recursivas que operen sobre representaciones de números naturales; terminaremos esta primera parte con la introducción de ISWIM. En la segunda parte desarrollaremos incrementalmente un sistema de tipos que proporcione polimorfismo paramétrico al cálculo mediante inferencia de tipos, por lo que no será necesario incluir anotaciones de tipos en un programa de ISWIM polimórfico. En la tercera parte de este trabajo desarrollaremos una serie de máquinas abstractas, cada vez más eficientes, para evaluar el cálculo de ISWIM polimórfico de manera eficiente y correcta.

En el capítulo final implementaremos ISWIM *polimórfico*, mediante el lenguaje de programación funcional Haskell [24]. Con esta implementación esperamos mostrar la eficacia de este paradigma para codificar casi de manera directa las funciones matemáticas descritas a lo largo de este trabajo. Además, implementaremos la inferencia de tipos mediante el estilo de programación monádica, que recientemente ha revolucionado el campo de la programación funcional [27, 39, 40, 34].

Esperamos que el presente trabajo sea de utilidad para el estudiante de ciencias de la computación interesado en el diseño de lenguajes de programación desde un enfoque matemático.

Parte I

El cálculo-λ como modelo de cómputo

Capítulo 1

El cálculo-λ puro

A inicio de la década de 1930 Alonzo Church desarrolló la *notación lambda* a partir de su intento de formalizar los fundamentos de las matemáticas; lamentablemente el sistema original resultó ser inconsistente como método de deducción lógica al ser susceptible a la paradoja descubierta por Curry y Rosser [37]. Sin embargo, en 1935 Church depuró el sistema original para conservar únicamente la parte referente al cómputo de aplicación de funciones [9], y poco tiempo después probó junto con Rosser que este nuevo sistema era consistente [11]. En este capítulo describiremos la sintaxis básica del cálculo-λ junto con la regla de reescritura que permite modelar la aplicación de funciones. Terminaremos el capítulo reiterando el resultado de Church y Rosser sobre la consistencia de este sistema formal.

1.1. Sintaxis

Para definir el *cálculo-λ puro* primero necesitamos definir el concepto de *variable*; para esto supondremos que contamos con un conjunto infinito $\mathbb{X}$ de *identificadores*. Usaremos letras minúsculas para denotar elementos pertenecientes a este conjunto.

Definición 1: $\mathbb{X}$ Variables [15]

$$\mathbb{X} \;=\; a \mid b \mid c \mid \ldots \mid x \mid y \mid z \mid \ldots$$

Definiremos el conjunto Λ de todas las posibles expresiones del cálculo-λ a manera de una gramática con tres posibles construcciones donde cada una de ellas está respectivamente etiquetada con `[var]`, `[λx.M]` y `[app]`. Estas etiquetas no forman parte de la sintaxis y solamente nos servirán para referirnos con facilidad a cada una de las posibles producciones de la gramática [15].

Definición 2: Λ Cálculo-λ puro [15]

$$\begin{array}{rcll} \Lambda & = & x & \texttt{[var]} \\ & | & (\lambda x.\, M) & \texttt{[λx.M]} \\ & | & (M\ N) & \texttt{[app]} \end{array}$$

donde $x \in \mathbb{X}$ y $M, N \in \Lambda$

La construcción más simple es la que genera *variables* mediante la regla `[var]`; así tenemos que x, y, y z son elementos atómicos del cálculo-λ. Las últimas dos reglas permiten crear recursivamente nuevos elementos a partir de otros más simples.

La regla `[λx.M]` nos permite construir abstracciones-λ a partir de una *variable* x y otro elemento M del cálculo-λ. Así, por ejemplo, como x y z son *variables*, entonces obtenemos que $(\lambda x.x)$ y $(\lambda x.z)$ son ambos elementos válidos del cálculo-λ. Diremos que en una abstracción-λ de la forma $(\lambda x.\, M)$, x es la *variable ligada* y M el *cuerpo* de la abstracción-λ [15].

A veces nos referiremos a las abstracciones-λ con el nombre de *funciones* ya que existe una correspondencia entre una función matemática de una variable y una abstracción-λ. La diferencia entre estas dos nociones es que en el caso de las funciones matemáticas, éstas disponen de un identificador el cual puede ser usado para invocar a la función en cualquier momento. Por ejemplo, la función $f(x) = x$ puede ser aplicada sobre argumentos distintos mediante la notación $f(0)$ y $f(7)$. En el caso de las abstracciones-λ, éstas no disponen de un identificador por lo que cada que queramos aplicar una abstracción-λ sobre argumentos distintos tendremos que denotarlo escribiendo la abstracción-λ completa en cada caso. Es por esta razón que se dice que las abstracciones-λ son *funciones anónimas*. Este anonimato podría parecer problemático no sólo en cuanto a conveniencia denotacional, sino también en cuanto a la capacidad de autoreferencia dentro de una función lo cual aparentemente nos impediría definir funciones recursivas; afortunadamente, éste no es el caso y veremos cómo solucionar este problema en la Sección 2.3.

Por último, la regla `[app]` construye una *aplicación*. Si tenemos dos elementos del cálculo-λ, digamos z y $(\lambda x.\, x)$, entonces $(z\ (\lambda x.\, x))$ y $((\lambda x.\, x)\ z)$ son ambos elementos válidos del cálculo-λ. La intención de una *aplicación* es proveer de un mecanismo para aplicar una abstracción-λ sobre un argumento. Por ejemplo, $((\lambda x.\, x)\ z)$ denota la aplicación de la función $(\lambda x.\, x)$ sobre el argumento z. En la Sección 1.5 veremos los detalles referentes a la aplicación de funciones.

A partir de ahora cuando $M \in \Lambda$ nos referiremos a M como una *expresión-λ* o simplemente como una *expresión*.

1.2. Convenciones sintácticas

Dado que usaremos ampliamente las expresiones-λ en este trabajo, estableceremos algunas convenciones sintácticas que nos facilitarán su lectura mediante la eliminación de los paréntesis que no sean semánticamente indispensables [15].

Tomemos por ejemplo la siguiente expresión:

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

Si convenimos en que las *aplicaciones* asocian a la izquierda, entonces podemos omitir todos los paréntesis en las *aplicaciones* salvo cuando queramos indicar alguna asociación a la derecha. Por ejemplo, la expresión anterior se simplifica de la siguiente manera:

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

Aquí eliminamos los paréntesis alrededor de $(x\ z)$ porque ahora que estamos de acuerdo en que la *aplicación* asocia a la izquierda, entonces no hay posibilidad de confundir $(x\ z\ (y\ z))$ con $(x\ (z\ (y\ z)))$. Asimismo, podemos omitir los paréntesis exteriores en $(x\ z\ (y\ z))$ y los que encierran a la expresión completa.

De manera similar podemos omitir los paréntesis que rodean a cada abstracción-λ usando la convención de que el alcance de las abstracciones-λ es todo lo que haya a la derecha del punto hasta llegar a algún paréntesis que la encierre. Por ejemplo, la expresión anterior se simplificaría más de la siguiente manera:

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

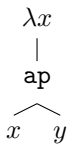
En este punto todos los paréntesis que quedan son necesarios, ya que si quitáramos los paréntesis de $(\lambda x.x)$, el significado de la expresión cambiaría de ser una aplicación de la función $\lambda x.x$ sobre el argumento $\lambda x.\ \lambda y.\ \lambda z.\ x\ \ z\ \ (y\ \ z)$ a ser únicamente una función $\lambda x.\ (x\ (\lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z)))$ que no se aplica a ningún argumento.

1.3. Variables libres y α-equivalencia

Cada expresión del cálculo-λ tiene un *árbol sintáctico* asociado el cual está compuesto por tres tipos de nodos [15].

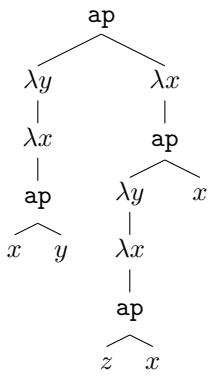
- Las *variables* se representan como hojas del árbol. Son nodos sin hijos que están etiquetados con su respectiva *variable*.
- Una *aplicación* $M\ \ N$ corresponde a un nodo interno denotado por `ap` que tiene a M como hijo izquierdo, y a N como hijo derecho.
- Una abstracción-λ $\lambda x.\ M$ se representa mediante un nodo interno etiquetado con λx y que tiene como único hijo a M.

Por ejemplo, el *árbol sintáctico* correspondiente a $(\lambda x.\ x\ \ y)$ es el siguiente:



Ahora consideremos un árbol más complejo que representa a la expresión

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



Usaremos los árboles sintácticos de una expresión M para clasificar las variables de M como *libres* o *ligadas*.

Definición 3: Variables libres y ligadas [15]

Si $M \in \Lambda$ es una expresión-λ y $x \in \mathbb{X}$ es una variable que figura en M, entonces decimos que x está *ligada en* M si existe un ancestro de x en el árbol sintáctico de M etiquetado con λx. En caso de que no exista tal ancestro diremos que la variable x se encuentra *libre en* M.

En caso de que existan varios ancestros de x etiquetados con λx diremos que la variable se *liga* únicamente al de mayor profundidad.

En el ejemplo anterior podemos ver que todas las variables salvo z están *ligadas* ya que no existe un nodo λz que sea ancestro de z. También notemos que las variables x están *ligadas* a nodos distintos por lo que podríamos pensar que a pesar de que llevan el mismo nombre, en realidad se comportan de manera distinta. Más adelante veremos que en efecto su semántica es distinta, por lo que vale la pena hacer la distinción desde ahora.

Definición 4: α-equivalencia [15]

Diremos que dos expresiones-λ M, N son *α-equivalentes*, si sus árboles sintácticos difieren únicamente en los nombres de las variables ligadas, y lo denotaremos mediante $M \equiv_\alpha N$.

En adelante diremos que $M = N$ si ambas expresiones son sintácticamente idénticas, o si $M \equiv_\alpha N$.

Siguiendo con el ejemplo anterior, tenemos entonces que

$$(\lambda y.\ \lambda x.\ x\ \ y)\ \ (\lambda x.\ (\lambda y.\ \lambda x.\ z\ \ x)\ \ x)\ \equiv_\alpha\ (\lambda a.\ \lambda b.\ b\ \ a)\ \ (\lambda c.\ (\lambda d.\ \lambda e.\ z\ \ e)\ \ c)$$

También nos interesará conocer el conjunto de variables libres de una expresión M, el cual podemos calcular usando la siguiente función.

Definición 5: $\mathcal{V}_\mathcal{L}$ Variables libres [15]

$$\begin{aligned} \mathcal{V}_\mathcal{L}(x) &= \{x\} \\ \mathcal{V}_\mathcal{L}(\lambda x.\ M) &= \mathcal{V}_\mathcal{L}(M) \setminus \{x\} \\ \mathcal{V}_\mathcal{L}(M\ \ N) &= \mathcal{V}_\mathcal{L}(M) \cup \mathcal{V}_\mathcal{L}(N) \end{aligned}$$

Si $\mathcal{V}_\mathcal{L}(M) = \varnothing$, entonces decimos que M es una *expresión cerrada*, un *combinador*, o para propósitos de este trabajo, un *programa*.

1.4. Sustitución

Ahora que conocemos el concepto de *variables libres*, definiremos una operación sobre expresiones-λ que nos permitirá empezar la discusión sobre la semántica del cálculo-λ.

La *sustitución* es una operación ternaria, denotada mediante $M[x \leftarrow N]$, donde M, N son expresiones-λ y x es una variable. La sustitución $M[x \leftarrow N]$ reemplaza todas las ocurrencias libres de x en M con la expresión N. Por ejemplo, la sustitución $(f\ x)[f \leftarrow \lambda y.y]$ resulta en la expresión $((\lambda y.y)\ x)$, pero la sustitución $(\lambda f.f\ x)[f \leftarrow \lambda y.y]$ resulta en $(\lambda f.f\ x)$ ya que no hay ocurrencias libres de f en $(\lambda f.f\ x)$.

Al definir la operación de *sustitución* $M[x \leftarrow N]$ debemos tratar a las variables libres en N de manera cuidadosa; específicamente queremos asegurar que si y es una variable libre en N, entonces y permanece libre en $M[x \leftarrow N]$. Por ejemplo, no queremos que la sustitución $(\lambda f.fx)[x \leftarrow f\ y]$ resulte en $\lambda f.f\ (f\ y)$ ya que si fuera así, habríamos introducido un ligado espurio sobre la variable f. Para evitar esto usaremos α-equivalencia antes de aplicar la sustitución de manera que cambiemos la variable ligada en una abstracción-λ por una variable nueva que no hayamos usado antes. De esta manera tendríamos que $(\lambda f.fx)[x \leftarrow f\ y]$ resulta en $\lambda g.g\ (f\ y)$, donde f permanece libre. A continuación mostramos la definición formal de esta operación.

Definición 6: Sustitución [15]

$$x[y \leftarrow N] = \begin{cases} N & \text{si } x = y \\ x & \text{si } x \neq y \end{cases}$$

$$(\lambda x.\, M)[y \leftarrow N] = \begin{cases} (\lambda x.\, M) & \text{si } x = y \\ (\lambda z.\, M[x \leftarrow z][y \leftarrow N]) & \text{si } x \neq y \end{cases}$$
$$\text{donde} \quad z \notin \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N)$$

$$(M_1\ M_2)[y \leftarrow N] = (M_1[y \leftarrow N]\ M_2[y \leftarrow N])$$

A veces representaremos al operador de sustitución $[x \leftarrow M]$ mediante la letra σ.

Convenimos que en caso de tener varias sustituciones seguidas sobre la misma expresión, éstas se efectúan una por una de izquierda a derecha. Es decir,

$$M\sigma_1\sigma_2 = ((M\sigma_1)\sigma_2)$$

La *sustitución* queda definida de manera muy intuitiva para el caso de *variables* y *aplicaciones*. Solamente en el caso de las abstracciones-λ es donde la definición se complica ligeramente para evitar ligar variables libres.

Si nos encontramos con el caso $(\lambda x.\, M)[x \leftarrow N]$ donde la variable que queremos sustituir es precisamente la variable que la abstracción-λ está ligando, entonces por la definición de $\mathcal{V}_{\mathcal{L}}$ sabemos que no es posible que haya instancias *libres* de x en M, por lo que el resultado es la misma expresión $(\lambda x.\, M)$.

Ahora que si tenemos el caso $(\lambda x.\, M)[y \leftarrow N]$, donde $x \neq y$, deberíamos descender recursivamente sobre M aplicando $M[y \leftarrow N]$; sin embargo, si hiciéramos esto y x aparece libre en N, entonces ésta quedaría ligada en el resultado de la sustitución. Para evitar que esto ocurra nos apoyaremos de la α-equivalencia, y elegiremos una nueva variable z de manera que ésta no aparezca libre en M ni en N. Ahora podemos cambiar la expresión $\lambda x.M$ por la expresión α-equivalente $\lambda z.M[x \leftarrow z]$. De esta manera podemos aplicar la sustitución $[y \leftarrow N]$ sobre esta nueva expresión, sin temor de que alguna variable libre de N quede ligada a la abstracción-λ ya que z no figura en N.

Ahora que conocemos la operación de sustitución, empezaremos con la discusión sobre la semántica del cálculo-λ y su correspondencia con el cómputo de la aplicación de funciones.

1.5. β-reducción

El cálculo-λ modela *funciones de una variable* mediante expresiones de la forma $\lambda x.M$ donde x funge como la variable que representa el argumento de la función. Para asociarle un valor a la variable x es necesario *aplicar* la función sobre alguna otra expresión formando una expresión de la forma $(\lambda x.M)\ N$.

Definición 7: β-redex [15]

Les damos el nombre de *β-redex* a las expresiones de la forma

$$(\lambda x.M)\ N$$

A veces las llamaremos simplemente *redex*. El nombre viene de la contracción de ***red**ucible **exp**ression* que se traduce como «expresión reducible».

El concepto de *β-redex* es fundamental en el cálculo-λ pues permite definir la regla de *β-reducción*, denotada $\underset{\beta}{\rightarrow}$, que reduce un *$\beta$-redex* a una expresión más sencilla.

Definición 8: $\underset{\beta}{\rightarrow}$ β-reducción [15]

La regla $\underset{\beta}{\rightarrow}$ opera sobre un *β-redex* de la siguiente manera.

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

Es decir, al *aplicar* la abstracción-λ $\lambda x.M$ sobre una expresión N, se obtiene el cuerpo de la abstracción-λ donde cada ocurrencia de x fue sustituida por N.

De esta manera se captura la noción de que las abstracciones-λ representan funciones de una variable. Por ejemplo $\lambda x.x$ representa a la función identidad, pues el resultado de aplicarla sobre una expresión M resulta en la misma M.

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

Podemos ver que $(\lambda x.x)$ y $(\lambda z.z)$ operan exactamente de la misma manera ya que son expresiones α-equivalentes.

También podemos representar funciones constantes mediante $\lambda x.M$ cuando x no figura en M como variable libre. Por lo que al aplicar la función obtenemos M independientemente de cuál haya sido el argumento. Observemos algunos ejemplos de β-reducción:

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

1.6. Currificación

La sintaxis del cálculo-λ solamente permite definir funciones de una variable. Por supuesto las funciones más interesantes operan sobre más de una variable, y afortunadamente existe una técnica conocida como *currificación*, popularizada por Haskell Curry[1] [12], la cual nos permite expresar estas funciones dentro del cálculo-λ tal y como lo hemos definido hasta ahora.

La currificación consiste en transformar una función de varias variables en una secuencia de funciones de una variable anidadas de manera que cada una reciba un solo argumento, y que devuelva como resultado una función que espera el siguiente argumento [12].

Por ejemplo, la función de tres variables $comp(f, g, x) = f(g(x))$, puede ser definida equivalentemente en el cálculo-λ mediante la expresión $\lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x)$ en la cual tenemos tres funciones anidadas donde la primera se aplica a un solo argumento, y que devuelve otra función que a su vez espera un solo argumento. Una vez que los tres argumentos han sido aplicados obtenemos el valor que esperábamos, tal y como se muestra en la siguiente reducción:

$$\begin{aligned}
(\lambda f.\, \lambda g.\, \lambda x.\, f\ (g\ x))\ f'\ g'\ x' &\underset{\beta}{\rightarrow} (\lambda g.\, \lambda x.\, f'\ (g\ x))\ g'\ x' \\
&\underset{\beta}{\rightarrow} (\lambda x.\, f'\ (g'\ x))\ x' \\
&\underset{\beta}{\rightarrow} f'\ (g'\ x')
\end{aligned}$$

[1] A pesar de que la currificación lleva el nombre de Haskell Curry, él no la inventó sino que la aprendió de Moses Schönfinkel quien la usó en 1924, e incluso ya había sido usada por Gottlob Frege en 1893 [38, 7].

Con el ejemplo anterior, resaltamos una de las características principales de la programación funcional, y esto es que las funciones se consideran *ciudadanas de primera clase*. Esto significa que una función puede recibir como argumento otra función, así como ser devuelta como resultado de otra función. A las funciones que aceptan o devuelven otras funciones se les conoce como *funciones de orden superior*, en contraste con las *funciones de primer orden* que no reciben ni devuelven funciones [1].

Las funciones de orden superior son muy útiles para abstraer ciertos comportamientos comunes. Por ejemplo, si queremos aplicar una función sobre una lista de elementos, podemos invocar a la función *map* que recibe una función y una lista, y devuelve la lista donde a cada elemento se le aplicó la función. Todavía más útil, es el concepto de *aplicación parcial* que consiste en aplicar una función de n variables sobre menos de n argumentos, para obtener una función parcialmente aplicada. Por ejemplo, si tenemos una lista de números y una función $\lambda a.\lambda b.M$ que devuelve la suma de dos números[2], entonces podemos usar la función *map* $((\lambda a.\lambda b.M)\ 5)$ para sumarle 5 a cada elemento de una lista.

El concepto de currificación forma uno de los primeros indicios que nos puede llevar a pensar que, en efecto, el cálculo-λ puede ser usado como fundamento para un lenguaje de programación de alto nivel.

1.7. Formas normales

La idea central del cálculo-λ es tomar una expresión M y reducirla hasta cierto punto donde consideremos que la nueva expresión sea el resultado de la evaluación de M. Una primera aproximación al concepto de *resultado de una evaluación* es una expresión que no pueda ser reducida más allá de su forma actual.

Definición 9: Forma normal [15]

Decimos que una expresión M está en su *forma normal*, si ninguna de sus subexpresiones es un β-*redex*. Estas expresiones no pueden simplificarse más.

Los siguientes ejemplos son expresiones que están en su *forma normal* y que no pueden reducirse más allá de ella.

- x

[2] En el próximo capítulo veremos cómo usar el cálculo-λ para definir números y operaciones sobre ellos.

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

Es importante mencionar que no todas las expresiones se pueden reducir a una forma normal. Un ejemplo de esto es el *combinador* Ω.

$$\Omega \stackrel{\text{def}}{=} (\lambda x.x\ \ x)\ \ (\lambda x.x\ \ x)$$

Veamos lo que sucede al aplicar $\underset{\beta}{\rightarrow}$ sobre Ω

$$\begin{aligned}(\lambda x.x\ \ x)\ \ (\lambda x.x\ \ x) &\underset{\beta}{\rightarrow} (x\ \ x)[x \leftarrow \lambda x.x\ \ x] \\ &= (\lambda x.x\ \ x)\ \ (\lambda x.x\ \ x) \\ &\stackrel{\text{def}}{=} \Omega\end{aligned}$$

Obtuvimos que $\Omega \underset{\beta}{\rightarrow} \Omega$, es decir, si intentamos encontrar una *forma normal* para Ω estaremos reduciéndolo a sí mismo hasta el fin de los tiempos. Este comportamiento es comparable con el de un programa de computadora cuya ejecución jamás termina.

1.8. Cerradura compatible y confluencia

Consideremos la expresión $\lambda z.\underline{(\lambda x.\ y)\ \ (\underline{(\lambda x.\ x)\ \ \lambda x.\ x})}$. Esta expresión no está en su forma normal ya que contiene dos subexpresiones (subrayadas) que forman un β-*redex*. Sin embargo, formalmente no podemos reducir esta expresión ya que la regla $\underset{\beta}{\rightarrow}$ solamente opera sobre expresiones de la forma $(\lambda x.M)\ N$, y esta expresión es de la forma $\lambda x.M$. Para solucionar este problema definiremos la *cerradura compatible* de $\underset{\beta}{\rightarrow}$, denotada mediante $\hookrightarrow_\beta$.

Definición 10: $\hookrightarrow_\beta$ Cerradura compatible de $\underset{\beta}{\rightarrow}$ [15]

La *cerradura compatible de* $\underset{\beta}{\rightarrow}$ opera sobre cualquier subexpresión de M que forme un β-*redex*. Si M contiene al β-*redex* L, donde $L \underset{\beta}{\rightarrow} L'$, entonces diremos que $M \hookrightarrow_\beta M'$ cuando la única diferencia entre M y M' sea que en el segundo L' aparece en lugar de L.

Volviendo al ejemplo anterior, podemos encontrar su forma normal usando la regla $\hookrightarrow_\beta$ para reducir alguno de sus β-*redex*. Sin embargo, la regla $\hookrightarrow_\beta$ no especifica cuál de los dos redex se debe reducir primero. Si primero reducimos el β-*redex* más superficial, entonces obtenemos inmediatamente que

$$\lambda z.\ \underline{(\lambda x.\ y)\ \ ((\lambda x.\ x)\ \ \lambda x.\ x)} \hookrightarrow_\beta \lambda z.\ y$$

Pero si empezamos con el β-*redex* más profundo obtenemos que

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

Usando esta otra ruta hemos llegado al mismo resultado, excepto que lo hicimos en dos pasos en lugar de uno.

Afortunadamente, no es coincidencia que en el ejemplo anterior hayamos obtenido el mismo resultado sin importar qué redex hayamos elegido para iniciar la evaluación. El cálculo-λ cumple con la propiedad conocida como *confluencia* que dice que si es posible reducir una expresión a dos distintas expresiones mediante secuencias distintas del uso de $\hookrightarrow_\beta$, entonces existe una tercera expresión a la cual ambas expresiones pueden ser reducidas. A esta propiedad también se le conoce como la propiedad de Church-Rosser, en honor a Alonzo Church y John Barkley Rosser quienes demostraron que esta propiedad se cumple en el cálculo-λ puro [11].

Escribiremos $M \overset{n}{\hookrightarrow}_\beta N$, para denotar que llegamos a N desde M usando n pasos de $\hookrightarrow_\beta$ sin tener que mostrar los pasos intermedios. Cuando decimos que $M \overset{0}{\hookrightarrow}_\beta N$, significa que hicimos cero reducciones, y que por lo tanto $M = N$. A veces usamos $M \overset{*}{\hookrightarrow}_\beta N$ para denotar que es posible llegar a N desde M en cero o cualquier otra cantidad finita de pasos. Le llamaremos a $\overset{*}{\hookrightarrow}_\beta$ la *cerradura transitiva reflexiva* de $\hookrightarrow_\beta$.

Teorema 1.8.1 (Church-Rosser)**.** *Si* $M \overset{*}{\hookrightarrow}_\beta L_1$ *y* $M \overset{*}{\hookrightarrow}_\beta L_2$*, entonces existe un* N *tal que* $L_1 \overset{*}{\hookrightarrow}_\beta N$ *y que* $L_2 \overset{*}{\hookrightarrow}_\beta N$*.*

Demostración. La prueba se aleja de los objetivos de este trabajo y se omite, pero puede ser consultada en [11]. □

La propiedad de confluencia es importante porque a partir de ella podemos probar como corolario que si una expresión tiene una forma normal, entonces ésta necesariamente es única.

Corolario 1.8.1 (Unicidad de la forma normal)**.** *Si* M *tiene una forma normal* M_f*, entonces ésta es única.*

Demostración. Supongamos que M tiene dos formas normales M_f y M'_f. Por la propiedad de Church-Rosser, existe una N tal que $M_f \overset{*}{\hookrightarrow}_\beta N$ y $M'_f \overset{*}{\hookrightarrow}_\beta N$. Como M_f y M'_f, están en forma normal, no pueden reducirse más allá, por lo que entonces $M_f \overset{0}{\hookrightarrow}_\beta N$ y $M'_f \overset{0}{\hookrightarrow}_\beta N$; y por lo tanto $M_f = M'_f$.

□

Con los resultados anteriores esperamos haber mostrado que el cálculo-λ forma un sistema formal robusto que permite el estudio sistematizado de las nociones básicas de la aplicación de funciones. Hemos ignorado algunos conceptos como la η-reducción, la lógica combinatoria y otros temas que históricamente han sido fundamentales en el estudio del cálculo-λ ya que no consideramos que sean relevantes para propósitos de este trabajo. Para un tratamiento detallado sobre estos conceptos consultar la siguiente bibliografía [5, 4, 17, 12]. Para una versión detallada del desarrollo histórico del cálculo-λ consultar [7].

Capítulo 2

Computando con el cálculo-λ puro

En el capítulo anterior vimos que el cálculo-λ consiste de tres construcciones sintácticas y una regla de reescritura. En este capítulo mostraremos cómo usar el cálculo-λ para codificar algunos tipos de datos fundamentales en lenguajes de programación como son los valores booleanos y números naturales, así como las estructuras de datos como tuplas. También mostraremos algunos mecanismos para definir funciones recursivas que operen sobre las codificaciones anteriores. El propósito de este capítulo es resaltar el poder computacional inherente del cálculo-λ puro y su potencial para ser usado como un lenguaje de programación de alto nivel.

2.1. Codificaciones de Church

Una de las debilidades aparentes del cálculo-λ es que al estar construido solamente por variables, abstracciones-λ, y aplicaciones, no hay forma de representar objetos matemáticos como números o valores booleanos. Si queremos convertir el cálculo-λ en un lenguaje de programación, será necesario disponer de estos objetos para luego poder definir funciones que los manipulen.

Es posible codificar estos tipos de datos como elementos del cálculo-λ, de manera que estas estructuras puedan ser recibidas y transformadas por otras funciones de forma que implementen las operaciones habituales. Church introdujo las codificaciones para números naturales en [8] donde también definió una función que calcula la suma de dos naturales codificados de esta manera. Esta mis-

ma intuición nos permite definir otros tipos de datos como booleanos e incluso estructuras de datos como tuplas, listas y árboles; además, por supuesto de funciones que las manipulen.

2.1.1. Booleanos

Observemos las siguientes dos funciones.

$$\lambda a.\lambda b.a \qquad\qquad \lambda a.\lambda b.b$$

Podemos empezar por notar que no son α-equivalentes entre sí, por lo que es natural esperar que tengan comportamientos distintos. La primera función toma dos argumentos, y devuelve el primero que recibió. La segunda función también recibe dos argumentos, pero devuelve el último que recibió.

Diremos que estas dos funciones son las representaciones del cálculo-λ para los valores booleanos de `true` y `false` [15].

- `true` $\stackrel{\text{def}}{=} \lambda a.\lambda b.a$
- `false` $\stackrel{\text{def}}{=} \lambda a.\lambda b.b$

Los símbolos de `true` y `false` se encuentran en el dominio de lo que llamaremos *azúcar sintáctica*[1]. Es decir, no son elementos sintácticos válidos en el cálculo-λ pero los usaremos ampliamente para facilitar su lectura. Una expresión que contenga *azúcar sintáctica* deberá *desazucararse*, esto es remplazar cada alias por sus correspondientes expresiones, antes de ser manipulada formalmente.

Ahora podemos definir una función `if` que reciba un booleano, y otros dos argumentos que dependiendo del valor booleano, devuelva el primero en caso de ser `true`, y el segundo en caso de ser `false`. Esta función se comporta como una sentencia `if-then-else` común en los lenguajes de programación.

$$\texttt{if} \stackrel{\text{def}}{=} \lambda b.\lambda t.\lambda f.b\ \ t\ \ f$$

[1] El término *azúcar sintáctica* fue acuñado por Peter Landin en [20] al introducir la sentencia M `where` $x = N$ que es equivalente a la aplicación $(\lambda x.M)\ N$. El propósito del azúcar sintáctica es facilitar la lectura y redacción de programas sin modificar la estructura del cálculo inherente. Vale la pena recalcar que el uso de azúcar sintáctica no aumenta la expresividad de un lenguaje, ya que éste sigue reconociendo el mismo conjunto de programas. Estas nuevas sentencias no afectan en ninguna manera al poder de cómputo inherente del lenguaje. En el Capítulo 6 retomaremos esta misma pieza de azúcar sintáctica, pero lo escribiremos como `let` $x = N$ `in` M por razones meramente cosméticas.

Es fácil ver que se cumplen las siguientes propiedades sobre `if`

- `if true` $M\ N \overset{*}{\hookrightarrow}_{\beta} M$
- `if false` $M\ N \overset{*}{\hookrightarrow}_{\beta} N$

También podemos aprovecharnos de la expresividad de la función `if` para definir los operadores booleanos `not`, `and` y `or` como se muestra a continuación:

- `not` $\overset{\text{def}}{=} \lambda x.$`if` x `false true`
- `and` $\overset{\text{def}}{=} \lambda x.\lambda y.$`if` $x\ y$ `false`
- `or` $\overset{\text{def}}{=} \lambda x.\lambda y.$`if` x `true` y

Debería ser fácil verificar que estos operadores funcionan tal y como se espera.

2.1.2. Tuplas

En lenguajes de programación modernos existe el tipo de dato *tupla* que permite crear colecciones finitas y ordenadas de elementos como $\langle 1, 5\rangle$ o $\langle 1, \texttt{true}, 5\rangle$.

Si tenemos un par ordenado $\langle M, N\rangle$, quisiéramos tener funciones *selectoras* `fst` y `snd` que cumplan:

- `fst` $\langle M, N\rangle \overset{*}{\hookrightarrow}_{\beta} M$
- `snd` $\langle M, N\rangle \overset{*}{\hookrightarrow}_{\beta} N$

Podemos definir $\langle M, N\rangle$ usando azúcar sintáctica como una función que toma M y N como argumentos, y que devuelve una función que espera un valor booleano que usaremos para extraer M o N de la tupla.

Los selectores `fst` y `snd` son funciones que reciben una tupla (una función que espera un valor booleano) y que la aplica respectivamente a `true` o `false`.

- $\langle M, N\rangle \overset{\text{def}}{=} \lambda m.\lambda n.\ \lambda s.$`if` $s\ m\ n$
- `fst` $\overset{\text{def}}{=} \lambda t.t$ `true`

- $\texttt{snd} \stackrel{\text{def}}{=} \lambda t.t\ \texttt{false}$

Una vez definido un par ordenado $\langle M, N\rangle$ podemos usarlo para generar de manera inductiva tuplas de orden superior como $\langle L, \langle M, N\rangle\rangle$, el cual denotaremos, por conveniencia, mediante $\langle L, M, N\rangle$. Definimos entonces los selectores para una tupla de aridad 3 mediante la composición de selectores para las tuplas de aridad 2:

- $\texttt{fst}_3 \stackrel{\text{def}}{=} \texttt{fst}$
- $\texttt{snd}_3 \stackrel{\text{def}}{=} \lambda t.\texttt{fst}\ (\texttt{snd}\ t)$
- $\texttt{thrd}_3 \stackrel{\text{def}}{=} \lambda t.\texttt{snd}\ (\texttt{snd}\ t)$

2.1.3. Números naturales

Podemos representar a los números naturales mediante *combinadores* de la forma $\lambda s.\lambda z.M$, donde M varía para representar cada natural. Llamaremos a estas representaciones *numerales de Church*, en honor a Alonzo Church quien los definió en [8, 10] como:

- $\hat{0} \stackrel{\text{def}}{=} \lambda s.\lambda z.z$
- $\hat{1} \stackrel{\text{def}}{=} \lambda s.\lambda z.s\ z$
- $\hat{2} \stackrel{\text{def}}{=} \lambda s.\lambda z.s\ (s\ z)$
- $\hat{3} \stackrel{\text{def}}{=} \lambda s.\lambda z.s\ (s\ (s\ z))$

  $\vdots$
- $\hat{n} \stackrel{\text{def}}{=} \lambda s.\lambda z.\underbrace{s\ (\ldots(s}_{n \text{ veces}}\ z)\ldots)$

Podemos ver a un *numeral* $\hat{n}$ como una función que toma como argumentos a una función s y un elemento z; y devuelve el resultado de aplicar n veces s sobre z. En general, un numeral de Church, además de ser una codificación para un número natural, es un mecanismo mediante el cual podemos iterar una función n veces sobre un elemento arbitrario. Este mecanismo es una de las herramientas fundamentales que nos permitirán definir funciones más interesantes. Por ejemplo, podemos definir la función `esPar` que toma un *numeral* y devuelva un booleano que indique si el número representado es par o impar.

$$\texttt{esPar} \stackrel{\text{def}}{=} \lambda n.n\ \texttt{not}\ \texttt{true}$$

Si evaluamos `esPar` sobre un numeral par, entonces iteraremos la función `not` sobre `true` una cantidad par de veces, devolviendo como resultado `true`. Si el numeral fuera impar entonces iteraríamos `not` una cantidad impar de veces, devolviendo así `false`.

También podemos definir una función `isZero` que reciba un *numeral* y devuelva `true` si el argumento es $\hat{0}$ y `false` en otro caso. Esto lo hacemos aplicando n veces la función constante $\lambda x.\,\texttt{false}$ sobre `true`. Si la aplicamos cero veces, entonces obtenemos `true` y para todos los demás numerales conseguimos un `false` [15].

$$\texttt{isZero} \stackrel{\text{def}}{=} \lambda n.n\ (\lambda x.\,\texttt{false})\ \texttt{true}$$

Usando la misma idea podemos definir la función `suc` que toma un *numeral* y devuelve su sucesor. Para lograrlo se aplica n veces s sobre z, y luego se aplica s una vez más.

$$\texttt{suc} \stackrel{\text{def}}{=} \lambda n.\lambda s.\lambda z.s\ \ (n\ \ s\ \ z)$$

Sumar n y m es igualmente sencillo. Aplicamos m veces s sobre z, y a lo que resulte le aplicamos s otras n veces.

$$\texttt{suma} \stackrel{\text{def}}{=} \lambda n.\lambda m.\lambda s.\lambda z.n\ \ s\ \ (m\ \ s\ \ z)$$

Incluso podemos definir el producto de dos *numerales* usando la intuición de que un producto es «*una suma abreviada*» para sumar n veces m sobre cero.

$$\texttt{mult} \stackrel{\text{def}}{=} \lambda n.\lambda m.\lambda s.\lambda z.n\ \ (\texttt{suma}\ \ m)\ \ \hat{0}$$

Si quisiéramos definir una función que calcule el predecesor de un *numeral* nos toparíamos con el problema de que no tenemos ningún mecanismo para remover aplicaciones de funciones, es decir que no podríamos pasar de $\lambda s.\lambda z.s\ z$ a sólo $\lambda s.\lambda z.z$. Henk Barendregt cuenta en [4] que Church estaba dispuesto a abandonar la idea de poder definir una función predecesor en el cálculo-λ hasta que Kleene, uno de sus estudiantes, en una visita al dentista dio con una definición que se apoyaba de tuplas para registrar una bandera booleana para indicar que

cuando sea verdadera se debe omitir una aplicación de s. Definiendo así las siguientes funciones.

- $\texttt{wrap} \stackrel{\text{def}}{=} \lambda f.\lambda t.\langle\, \texttt{false},\ \ \texttt{if}\ (\texttt{fst}\ t)\ (\texttt{snd}\ t)\ (f\ \ (\texttt{snd}\ t))\,\rangle$
- $\texttt{pred} \stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.\texttt{snd}\ \ (n\ \ (\texttt{wrap}\ \ f)\ \ \langle \texttt{true},\ \ x\rangle)$

Para calcular el predecesor de $\hat{n}$, aplicamos n veces $\texttt{wrap}\ f$ sobre la tupla $\langle \texttt{true},\ \ x\rangle$. La primera aplicación de $\texttt{wrap}\ f$, devuelve una tupla con `false` como primer elemento, y para calcular el segundo elemento verifica si la bandera booleana de la tupla que recibió como argumento es `true` o `false`. Si la bandera es `true`, entonces devuelve x tal cual. Si la bandera fuera `false` entonces devuelve $f\ x$. Solamente la primera aplicación tiene `true` en la bandera, por lo que aplicará la función $n-1$ veces, construyendo así el predecesor de n. Aquí un ejemplo:

$$
\begin{aligned}
\texttt{pred}\ \hat{2} &\hookrightarrow_\beta \lambda f.\lambda x.\texttt{snd}\ (\underline{\hat{2}\ (\texttt{wrap}\ \ f)}\ \langle \texttt{true},\ \ x\rangle)\\
&\hookrightarrow_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \underline{(\lambda z.\texttt{wrap}\ f\ (\texttt{wrap}\ f\ z)\ )}\ \langle \texttt{true},\ x\rangle)\\
&\hookrightarrow_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \texttt{wrap}\ f\ (\underline{\texttt{wrap}\ f\ \langle \texttt{true},\ x\rangle})\ )\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \texttt{wrap}\ f\ \langle\ \texttt{false},\ \texttt{if}\ (\underline{\texttt{fst}\ \langle \texttt{true},\ x\rangle})\ (\underline{\texttt{snd}\ \langle \texttt{true},\ x\rangle})\ (f\ (\underline{\texttt{snd}\ \langle \texttt{true},\ x\rangle}))\ \rangle\ )\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \texttt{wrap}\ f\ \langle\ \texttt{false},\ \underline{\texttt{if true}\ \ x\ \ (f\ x)}\ \rangle\ )\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \underline{\texttt{wrap}\ f\ \langle\ \texttt{false},\ x\ \rangle}\ )\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \langle\ \texttt{false},\ \texttt{if}\ (\underline{\texttt{fst}\ \langle\ \texttt{false},\ x\ \rangle})\ (\underline{\texttt{snd}\ \langle\ \texttt{false},\ x\ \rangle})\ (f\ (\underline{\texttt{snd}\ \langle\ \texttt{false},\ x\ \rangle}))\ \rangle\ )\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.\texttt{snd}\ (\ \langle\ \texttt{false},\ \ \underline{\texttt{if false}\ \ x\ \ (f\ \ x)}\ \rangle\ )\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.\underline{\texttt{snd}\ (\ \langle\ \texttt{false},\ \ f\ x\ \rangle\ )}\\
&\hookrightarrow^*_\beta \lambda f.\lambda x.f\ x\ \ \equiv_\alpha\ \ \lambda s.\,\lambda z.s\ z\ \ \equiv\ \ \hat{1}
\end{aligned}
$$

Finalmente definimos una función `resta` $n\ m$ que devuelva $n-m$ al aplicar m veces la función `pred` sobre n.

$$\texttt{resta} \stackrel{\text{def}}{=} \lambda n.\lambda m.m\ \texttt{pred}\ n$$

En esta sección hemos definido algunos de los objetos matemáticos fundamentales para realizar operaciones aritméticas y booleanas. En la siguiente sección veremos cómo definir funciones recursivas para realizar cómputos más complejos.

2.2. Recursión

Muchas veces es más sencillo definir funciones recursivas que iterativas. Afortunadamente, es posible definir funciones recursivas en el cálculo-λ de manera sencilla. En esta sección mostraremos algunos mecanismos para dar tales definiciones.

Tomemos, por ejemplo, la función factorial. Una primer definición sería como la siguiente.

$$\texttt{fact}_0 \stackrel{\text{def}}{=} \lambda n.\texttt{if}\ (\texttt{isZero}\ n)\ \ \hat{1}\ \ (\texttt{mult}\ n\ (\texttt{fact}_0\ (\texttt{pred}\ n)))$$

Sin embargo, esta definición no tiene sentido ya que hemos asignado nombres a las funciones usando *azúcar sintáctica*. Si quisiéramos evaluar la función $\texttt{fact}_0$ como la hemos definido arriba, entonces como primer paso debemos *desazucarar* su definición; esto es expandir los nombres de las funciones `if`, `isZero`, `mult`, $\texttt{fact}_0$ y `pred`, por sus correspondientes expresiones *desazucaradas*. Sin embargo, al intentar remplazar $\texttt{fact}_0$ por su definición nos encontramos con que volvemos a introducir $\texttt{fact}_0$, por lo que este proceso se ciclaría infinitamente; y por lo tanto, no nos sirve para definir funciones recursivas.

Lo que necesitamos es una manera de referirnos a la función dentro de su definición sin recurrir al nombre que le asignamos mediante azúcar sintáctica, cosa que ya hemos hecho. Recordemos al combinador $(\lambda x.x\ x)\ (\lambda x.x\ x)$. Al aplicar $\omega \stackrel{\text{def}}{=} (\lambda x.x\ x)$ sobre ella misma obtenemos ω dentro de la definición de ω misma. La clave aquí es la *autoaplicación*.

Podemos extender $\texttt{fact}_0$ de manera que sea una función que tome un parámetro adicional f para referirse a ella misma, y que se *autoaplique* en su definición.

$$\texttt{fact}_1 \stackrel{\text{def}}{=} \underline{\lambda f.}\ \lambda n.\texttt{if}\ (\texttt{isZero}\ n)\ \ \hat{1}\ \ (\texttt{mult}\ n\ (\underline{f\ f}\ (\texttt{pred}\ n)))$$

Hemos subrayado las partes en las que $\texttt{fact}_1$ difiere de $\texttt{fact}_0$. Podemos ver que en esta nueva definición no hacemos mención al nombre de la función en su definición, y en lugar de eso ponemos la autoaplicación $f\ f$. Para calcular el factorial de n, basta calcular $\texttt{fact}_1\ \texttt{fact}_1\ n$. Veamos un ejemplo.

$$
\begin{aligned}
\underline{\texttt{fact}_1\ \texttt{fact}_1\ \hat{2}} &\hookrightarrow^*_\beta\ \texttt{if}\ (\underline{\texttt{isZero}\ \hat{2}})\ \hat{1}\ (\texttt{mult}\ \hat{2}\ (\texttt{fact}_1\ \texttt{fact}_1\ (\underline{\texttt{pred}\ \hat{2}})))\\
&\hookrightarrow^*_\beta\ \underline{\texttt{if false}\ \hat{1}\ (\texttt{mult}\ \hat{2}\ (\texttt{fact}_1\ \texttt{fact}_1\ \hat{1}))}\\
&\hookrightarrow^*_\beta\ \texttt{mult}\ \hat{2}\ (\underline{\texttt{fact}_1\ \texttt{fact}_1\ \hat{1}})\\
&\hookrightarrow^*_\beta\ \texttt{mult}\ \hat{2}\ (\texttt{if}\ (\underline{\texttt{isZero}\ \hat{1}})\ \hat{1}\ (\texttt{mult}\ \hat{1}\ (\texttt{fact}_1\ \texttt{fact}_1\ (\underline{\texttt{pred}\ \hat{1}}))))\\
&\hookrightarrow^*_\beta\ \texttt{mult}\ \hat{2}\ (\underline{\texttt{if false}\ \hat{1}\ (\texttt{mult}\ \hat{1}\ (\texttt{fact}_1\ \texttt{fact}_1\ \hat{0}))})\\
&\hookrightarrow^*_\beta\ \texttt{mult}\ \hat{2}\ (\texttt{mult}\ \hat{1}\ (\texttt{if}\ (\underline{\texttt{isZero}\ \hat{0}})\ \hat{1}\ (\texttt{mult}\ \hat{0}\ (\texttt{fact}_1\ \texttt{fact}_1\ (\texttt{pred}\ \hat{0})))))\\
&\hookrightarrow^*_\beta\ \texttt{mult}\ \hat{2}\ (\texttt{mult}\ \hat{1}\ (\underline{\texttt{if true}\ \hat{1}\ (\texttt{mult}\ \hat{0}\ (\texttt{fact}_1\ \texttt{fact}_1\ (\texttt{pred}\ \hat{0})))}))\\
&\hookrightarrow^*_\beta\ \underline{\texttt{mult}\ \hat{2}\ (\texttt{mult}\ \hat{1}\ \hat{1})}\\
&\hookrightarrow^*_\beta\ \hat{2}
\end{aligned}
$$

La estrategia que acabamos de usar funciona para el caso general. Empezamos con una función mal definida que ocupe el nombre de la función en su cuerpo.

$$\texttt{foo}_0 \stackrel{\text{def}}{=} \ldots \texttt{foo}_0 \ldots \texttt{foo}_0 \ldots$$

Agregamos un parámetro f a la función y cambiamos cada referencia al nombre por su autoaplicación.

$$\texttt{foo}_1 \stackrel{\text{def}}{=} \lambda f.\ \ldots (f\ \ f) \ldots (f\ \ f) \ldots$$

Finalmente aplicamos la función anterior sobre ella misma para obtener la función recursiva que tiene el mismo comportamiento que esperábamos de la definición original, con la sutil diferencia que esta nueva función sí es válida en el cálculo-λ.

$$\texttt{foo} \stackrel{\text{def}}{=} (\texttt{foo}_1\ \ \texttt{foo}_1)$$

El mecanismo que acabamos de desarrollar, a pesar de ser correcto es tedioso de utilizar y propenso a errores. Una manera más eficiente y elegante de definir funciones recursivas es mediante el uso de un *combinador de punto fijo*.

2.3. Combinadores de punto fijo

En las matemáticas convencionales decimos que x es un *punto fijo* de la función f si $x = f(x)$. En el cálculo-λ se ha descubierto una serie de *combinadores* que reciben como parámetro una función f del cálculo-λ, y que devuelven como resultado el punto fijo de f. A éstos se les conoce como *combinadores de punto fijo*.

Si Y es un *combinador de punto fijo* y f es cualquier función del cálculo-λ, entonces $(Y\ f)$ devuelve el punto fijo de f; y entonces tenemos que

$$Y\ f = f\ (Y\ f)$$

Esta familia de combinadores tiene una amplia relevancia en el estudio del cálculo-λ, y en su mayoría queda fuera del alcance de este trabajo; sin embargo, hay un aspecto de ellos que nos interesa y éste es que un combinador que un combinador de punto fijo nos permite implementar la recursión de una manera más práctica que la estrategia que desarrollamos en la sección anterior.

El uso de un combinador de punto fijo, nos permite encapsular el concepto de autoaplicación en el combinador, en lugar de hacerlo en la definición de la función. Por ejemplo podemos definir la función `fact` de la siguiente manera.

$$\texttt{fact} \stackrel{\text{def}}{=} \underline{Y}\ \lambda f.\lambda n.\texttt{if}\ (\texttt{isZero}\ n)\ \ \hat{1}\ \ (\texttt{mult}\ n\ (\underline{f}\ (\texttt{pred}\ n)))$$

Hemos subrayado las diferencias entre esta definición y la de $\texttt{fact}_1$, que como ya dijimos consiste en introducir el combinador Y para poder eliminar la autoaplicación $f\ f$. Notemos que esta función ya está lista para ser aplicada sobre un numeral, y que no tenemos que aplicarla primero a ella misma, como hicimos en la sección anterior.

Podemos ver un ejemplo de cómo se comporta esta función, suponiendo que Y es en efecto un combinador de punto fijo, y que por lo tanto satisface $Y\ f = f\ (Y\ f)$. Para esto usaremos un poco más de *azúcar sintáctica* para facilitar la lectura del siguiente ejemplo.

$$\Sigma \stackrel{\text{def}}{=} \lambda n.\texttt{if}\ (\texttt{isZero}\ n)\ \ \hat{1}\ \ (\texttt{mult}\ n\ (f\ (\texttt{pred}\ n)))$$

También redefinimos `fact` como

$$\texttt{fact} \stackrel{\text{def}}{=} Y\ \lambda f.\Sigma$$

Entonces tenemos que `fact` $\hat{1}$ se reduce de la siguiente manera

$$
\begin{aligned}
(\underline{Y\ \ \lambda f.\Sigma})\ \hat{1} &\overset{\text{def}}{=} (\underline{\lambda f.\Sigma\ \ \texttt{fact}})\ \hat{1} \\
&\overset{*}{\hookrightarrow}_{\beta}\ \underline{(\lambda n.\texttt{if}\ (\texttt{isZero}\ n)\ \ \hat{1}\ \ (\texttt{mult}\ n\ (\texttt{fact}\ (\texttt{pred}\ n))))}\ \ \hat{1} \\
&\overset{*}{\hookrightarrow}_{\beta}\ \texttt{if}\ (\underline{\texttt{isZero}\ \hat{1}})\ \ \hat{1}\ \ (\texttt{mult}\ \hat{1}\ \ (\texttt{fact}\ (\underline{\texttt{pred}\ \hat{1}}))) \\
&\overset{*}{\hookrightarrow}_{\beta}\ \underline{\texttt{if false}\ \ \hat{1}\ \ (\texttt{mult}\ \hat{1}\ \ (\underline{\texttt{fact}}\ \hat{0}))} \\
&\overset{*}{\hookrightarrow}_{\beta}\ \texttt{mult}\ \hat{1}\ \ (\underline{Y\ \ \lambda f.\Sigma}\ \ \hat{0}) \\
&\overset{\text{def}}{=}\ \texttt{mult}\ \hat{1}\ \ (\underline{\lambda f.\Sigma\ \ \texttt{fact}}\ \ \hat{0}) \\
&\overset{*}{\hookrightarrow}_{\beta}\ \texttt{mult}\ \hat{1}\ \ (\underline{(\lambda n.\texttt{if}\ (\texttt{isZero}\ n)\ \ \hat{1}\ \ (\texttt{mult}\ n\ (\texttt{fact}\ (\texttt{pred}\ n))))}\ \ \hat{0}) \\
&\overset{*}{\hookrightarrow}_{\beta}\ \texttt{mult}\ \hat{1}\ \ (\texttt{if}\ (\underline{\texttt{isZero}\ \hat{0}})\ \ \hat{1}\ \ (\texttt{mult}\ \ \hat{0}\ \ (\texttt{fact}\ (\texttt{pred}\ \hat{0})))) \\
&\overset{*}{\hookrightarrow}_{\beta}\ \texttt{mult}\ \hat{1}\ \ (\underline{\texttt{if true}\ \ \hat{1}\ \ (\texttt{mult}\ \hat{0}\ \ (\texttt{fact}\ (\texttt{pred}\ \hat{0})))}) \\
&\overset{*}{\hookrightarrow}_{\beta}\ \texttt{mult}\ \hat{1}\ \ \hat{1} \\
&\overset{*}{\hookrightarrow}_{\beta}\ \hat{1}
\end{aligned}
$$

Los pasos cruciales son en los que $(Y\ \ \lambda f.\Sigma)$ se reduce a $(\lambda f.\Sigma\ \ (Y\ \ \lambda f.\Sigma))$, lo cual es válido ya que supusimos que Y era un combinador de punto fijo.

Es así como podemos desarrollar una nueva estrategia general para definir funciones recursivas. Empezando con una definición mal formada en la que el nombre de la función aparezca en su cuerpo.

$$\texttt{bar}_0 \overset{\text{def}}{=} \ldots \texttt{bar}_0 \ldots \texttt{bar}_0 \ldots \texttt{bar}_0 \ldots$$

Le agregamos un parámetro f a la función, cambiando cada ocurrencia del nombre de la función por la aplicación de f. Finalmente le aplicamos un combinador de punto fijo a eso.

$$\texttt{bar} \overset{\text{def}}{=} Y\ \ \lambda f. \ldots . f \ldots f \ldots f \ldots$$

Ahora solamente nos hace falta mostrar la existencia de un combinador de punto fijo. En realidad hay muchos de ellos, pero el más conocido es el que descubrió Haskell Curry y es el que usualmente tiene el privilegio de llamarse Y.

Definición 11: El combinador de punto fijo Y [15]

$$Y \overset{\text{def}}{=} \lambda f.\ \ (\lambda x.\ f\ (x\ x))\ \ \ \lambda x.\ f\ (x\ x)$$

Teorema 2.3.1. *Y es un combinador de punto fijo.*

Demostración. Para probar esto, debemos mostrar que Y satisface la propiedad $Y\ f = f\ (Y\ f)$ para toda función f del cálculo-λ^2 .

$$\begin{aligned} Y\ f' &\stackrel{\text{def}}{=} \underline{(\lambda f.\ (\ (\lambda x.f\ (x\ x))\ \ (\lambda x.f\ (x\ x))\ ))\ \ f'} \\ &\hookrightarrow_\beta \underline{(\lambda x.f'\ (x\ x))\ \ (\lambda x.f'\ (x\ x))} \\ &\hookrightarrow_\beta f'\ \underbrace{((\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))}_{Y\ f'} \end{aligned}$$

□

2.4. Semántica operacional

Hemos visto que por la propiedad de confluencia siempre que una expresión tenga una forma normal llegaremos a ella bajo cualquier secuencia de reducciones; y hasta ahora hemos aplicado la regla de reescritura $\underset{\beta}{\rightarrow}$ de manera arbitraria donde mejor nos haya parecido mediante el uso de $\hookrightarrow_\beta$. En esta sección desarrollaremos estrategias de evaluación deterministas que nos permitirán implementar un evaluador que reduzca mecánicamente una expresión-λ a un *valor*.

2.4.1. Valores

Al evaluar una expresión-λ quisiéramos obtener un *valor*, esto es una expresión cuyo significado sea atómico y en correspondencia con algún objeto matemático que consideremos significativo; y que por lo tanto no debe de ser evaluada más allá de su forma actual. Hemos hablado ya de formas normales, que parecen ser buenos candidatos a *valores*; sin embargo, esta correspondencia no es del todo correcta. Por ejemplo $(x\ \ \lambda x.x)$ está en su forma normal, pero tenemos dificultades para adjudicarle un significado concreto a esta expresión. En general las aplicaciones no pueden ser valores porque implican que hace falta realizar más pasos de evaluación para llegar a un resultado significativo.

La elección de qué tipo de expresiones pueden ser consideradas *valores* es un tanto arbitraria. Muchos lenguajes de programación no consideran a las funciones como *valores* lo cual usualmente impide pasar a una función como argumento

[2]Para probar formalmente que $Y\ f = f\ (Y\ f)$ hace falta definir la igualdad entre expresiones-λ, la cual es equivalente a la cerradura simétrica de $\hookrightarrow_\beta$.

a otra función, o incluso regresar una función como resultado de la evaluación de otra función. En el ámbito de la programación funcional, las funciones son consideradas como *ciudadanas de primera clase*, y por lo tanto son consideradas *valores*.

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

Definición 12: Valores del cálculo-λ [15]

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

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

Vale la pena notar que una función $\lambda x.M$ es un valor a pesar de que no necesariamente esté en su forma normal ya que nada impide que M contenga algún β-*redex* como subexpresión.

Ahora que convenimos sobre qué tipo de expresiones se consideran valores, desarrollaremos un par de estrategias de evaluación que nos permitan convertir una expresión en un valor de manera determinista.

2.4.2. Estrategias de evaluación

En esta sección desarrollaremos dos estrategias de evaluación para el cálculo-λ que operan de manera determinista. A estas estrategias se les conoce tradicionalmente como *llamada por valor* y *llamada por nombre*, y para cada una consideraremos sus ventajas y desventajas sobre la otra. Conveniremos en cuál de las estrategias elegir para desarollar el resto de este trabajo. Es importante formalizar la estrategia de evaluación para mantener homogeneidad entre las implementaciones del lenguaje así como para poder garantizar ciertas propiedades de la evaluación.

Esencialmente la diferencia entre *llamada por valor* y *llamada por nombre* radica en la manera en la que se evalúan las aplicaciones $(\lambda x.M)\ N$. La estrategia de *llamada por valor* requiere que antes de aplicar la β-reducción, primero se evalúe

N hasta convertirlo en un valor. En la estrategia de *llamada por nombre*, no se contempla esta restricción y los β-*redex* se reducen sin importar si N es un valor o no [33, 15].

Formalizaremos cada estrategia de evaluación mediante la notación de *reglas de inferencia*. Una *regla de inferencia* se denota con una linea horizontal que en la parte superior tiene una serie de precondiciones o premisas y en la parte inferior tiene una o varias conclusiones. Si mostramos que las precondiciones se cumplen, entonces podemos usar la regla de inferencia para deducir que la conclusión también es válida.

Primero definiremos la llamada por valor que consta de tres reglas de inferencia.

Definición 13: $\rightarrow_v$ Llamada por valor [15]

$$\frac{M \rightarrow_v M'}{(M\ N) \rightarrow_v (M'\ N)}\ \texttt{[ev-app-i]}$$

$$\frac{v \in \mathcal{V} \qquad N \rightarrow_v N'}{(v\ N) \rightarrow_v (v\ N')}\ \texttt{[ev-app-d]}$$

$$\frac{v \in \mathcal{V}}{(\lambda x.M)\ v \rightarrow_v M[x \leftarrow v]}\ \texttt{[ev-red]}$$

Podemos leer la regla `[ev-app-i]` como: «si M se evalúa a M', entonces la aplicación $M\ \ N$ se evalúa a $M'\ \ N$». Esta regla captura la noción de que en una aplicación, primero se evalúa la expresión de la izquierda hasta que se reduzca a un valor.

La regla `[ev-app-d]` dice que si v es un valor (una variable o una abstracción-λ) y N se evalúa a N', entonces la aplicación $v\ \ N$ se evalúa a $v\ \ N'$. Con esta regla y la anterior dejamos claro que en una aplicación la evaluación se realiza de izquierda a derecha.

La regla `[ev-red]` especifica que la β-reducción solamente se efectúa cuando se aplica una función a un valor.

Alternativamente, podemos definir la llamada por nombre usando solamente dos reglas de inferencia, y en la cual nos deshacemos de la restricción que nos pide reducir el argumento de una aplicación a un valor antes de realizar de β-reducción.

Definición 14: $\rightarrow_n$ Llamada por nombre

$$\frac{M \rightarrow_n M'}{(M\ N) \rightarrow_n (M'\ N)}$$

$$\frac{}{(\lambda x.M)\ N \rightarrow_n M[x \leftarrow N]}$$

Notemos que a diferencia de la regla $\hookrightarrow_\beta$ ambas estrategias de evaluación son deterministas por lo que podríamos elegir cualquiera de éstas para implementar un evaluador del cálculo-λ; sin embargo, debemos de elegir sólo una de éstas para desarrollar el resto de este trabajo. Para ello consideraremos las debilidades y fortalezas de ambas estrategias. Un tratamiento detallado sobre la relación entre el evaluador inducido por ambas estrategias puede encontrarse en el trabajo de Plotkin [33].

La estrategia de *llamada por nombre* tiene una fuerte desventaja sobre la *llamada por valor*, ya que en una aplicación $(\lambda x.M)\ N$ el argumento N puede ser usado varias veces en el cuerpo de la función, y si éste se recibe sin evaluar entonces eventualmente tendremos que reducir N a un valor tantas veces como x aparezca libre en M. En cambio si primero reducimos el argumento a un valor, entonces no hay necesidad de volver a calcular varias veces la misma expresión [33, 15].

Sin embargo, la *llamada por valor* tiene el inconveniente de que si la función es una función constante, entonces habremos calculado el valor del argumento innecesariamente. Aun así preferiremos esta estrategia ya que los problemas son menores y son más fáciles de evitar por el programador, y también es sencillo de realizar la optimización adecuada durante el proceso de compilación. En adelante quedará implícito que la estrategia de evaluación de la que hablamos es mediante *llamada por valor* y en lugar de escribir $\rightarrow_v$ escribiremos simplemente $\rightarrow$.

Vale la pena notar que hay expresiones para las cuales no está definida su semántica operacional, por ejemplo, $(y\ \ \lambda x.x)$; esto se debe a que no podemos encontrar una interpretación adecuada para lo que significa aplicar una variable a una función. En el Capítulo 4 desarrollaremos un sistema que permita clasificar este tipo de expresiones como sin sentidos que no deben ser evaluadas.

2.4.3. Recursión por valor

Ahora que nos hemos comprometido con la *llamada por valor* deberemos volver a hablar sobre combinadores de punto fijo ya que al introducir esta estrategia de evaluación hemos roto el mecanismo mediante el cual definíamos funciones recursivas [15].

Recordemos cómo hemos definido a Y

$$Y \stackrel{\text{def}}{=} \lambda f.\ (\lambda x.f\ (x\ x))\quad \lambda x.f\ (x\ x)$$

Ahora analicemos lo que sucede al evaluar $(Y\ \ f)$ bajo $\rightarrow_v$

$$\begin{aligned} Y\ \ f\ \ &= \quad (\lambda f.\ (\lambda x.f\ (x\ x))\quad \lambda x.f\ (x\ x))\ \ f \\ &\rightarrow_v \ \ (\lambda x.f\ (x\ x))\quad \lambda x.f\ (x\ x) \\ &\rightarrow_v \ \ f\ \ \underbrace{((\lambda x.f\ (x\ x))\ \ \lambda x.f\ (x\ x))}_{\text{no es un valor}} \end{aligned}$$

En el primer paso eliminamos la azúcar sintáctica para expandir la definición de Y y luego usamos dos veces la regla `[ev-red]` para hacer β-reducción sobre el β-*redex* más superficial, tal y como la regla indica.

Para continuar la evaluación tendríamos que aplicar la regla `[ev-app-d]` para reducir $(\lambda x.f\ \ (x\ x))\ \ \lambda x.f\ \ (x\ x)$ a un valor; sin embargo, en el segundo paso obtuvimos exactamente esta expresión la cual se redujo a una expresión cuya reducción implica reducir la misma expresión. Podemos ver que este proceso se cicla infinitamente y por lo tanto no nos sirve para definir funciones recursivas, pues naturalmente queremos que su evaluación termine eventualmente.

El problema está en que $(\lambda x.f\ (x\ x))\ \ \lambda x.f\ (x\ x)$ nunca se reducirá a un valor, y por lo tanto el error está en pedir tal cosa. Sin embargo, podemos tomar otro combinador de punto fijo en el cual no se presente este problema.

Definición 15: Y_v Combinador de punto fijo por valor [15]

$$Y_v \stackrel{\text{def}}{=} \lambda f.(\lambda y.f\ (\lambda z.y\ y\ z))\ (\lambda y.f\ (\lambda z.y\ y\ z))$$

Teorema 2.4.1. *Y_v es un combinador de punto fijo.*

Demostración. Para mostrar que Y_v es un combinador de punto fijo nos hace falta definir formalmente la igualdad entre expresiones-λ; la cual se define como la cerradura simétrica de $\hookrightarrow_\beta$. Consideramos que estos detalles son de poca importancia para el presente trabajo y se omiten. □

Con este capítulo esperamos haber mostrado el potencial del cálculo-λ para ser usado como un lenguaje de programación de alto nivel. En el siguiente capítulo daremos el primer paso para lograr este objetivo al extender el cálculo-λ puro con un par de construcciones adicionales que permitirán la inclusión de constantes y operadores primitivos.

Capítulo 3

ISWIM

En el capítulo anterior mostramos que es posible representar números y valores booleanos en el cálculo-λ puro, y aunque pudimos definir ciertas operaciones sobre ellos, nos queda el inconveniente de que la complejidad de calcular el resultado de estas operaciones escala linealmente sobre el valor de sus argumentos; y por lo tanto resultaría ineficiente usarlas como parte del núcleo de nuestro lenguaje de programación.

Podemos evitarnos esta complicación al incluir dentro del cálculo un conjunto primitivo de constantes y operadores que las manipulen. De esta manera podríamos agregar booleanos, números enteros o de punto flotante y funciones específicas que implementen por ejemplo la lógica booleana, operaciones aritméticas o métodos numéricos. Podemos elegir qué constantes y operadores incluir según sea necesario para la tarea que queramos resolver.

La ventaja de tomar este enfoque es que, como Landin señaló en [21], «la mayoría de los lenguajes de programación son en parte un conjunto primordial de cosas, y en parte una manera de expresar cosas en términos de otras cosas»; y que el atractivo de un lenguaje de programación radica en el conjunto primordial de cosas, esto es, lo que uno puede calcular con el lenguaje. Ignorar esta observación tiene como consecuencia que exista un sinfín de lenguajes de programación orientados a distintas tareas en los cuales se presta más atención a la forma de expresar cosas en términos de otras cosas que a lo que realmente el lenguaje puede hacer, de esto se sigue que la mayoría de la especificación de un lenguaje es irrelevante para los problemas que se supone que debe resolver.

Landin propone un sistema de propósito general que permite separar los elementos primordiales de la manera de expresar otros elementos en términos de los primeros. Así podemos enfocarnos en lo segundo y extender lo primero según sea

necesario para cada área de aplicación. De esta manera queda definida una familia de lenguajes de programación a la cual jocosamente llamó ISWIM, acrónimo de *If you See What I Mean*.

En este capítulo nos dedicaremos a desarrollar el cálculo de ISWIM y en el resto de este trabajo extenderemos su expresividad sin preocuparnos demasiado por las funcionalidades primitivas del lenguaje.

3.1. Constantes y operadores primitivos

Como primer paso definiremos $\mathcal{C}_\mathcal{P}$, un conjunto de constantes primitivas.

Definición 16: $\mathcal{C}_\mathcal{P}$ Constantes primitivas [15]

Consideraremos un conjunto de constantes primitivas al cual denotaremos mediante $\mathcal{C}_\mathcal{P}$. En este conjunto vivirán todas las constantes que consideremos necesarias según los problemas que queramos resolver con cada instancia particular de ISWIM.

Distinguiremos a las constantes de las variables poniendo un punto sobre las constantes; así $\dot{a}$ será una constante y b una variable.

De aquí en adelante supondremos que el conjunto $\mathcal{C}_\mathcal{P}$ incluye por lo menos a las constantes booleanas `true` y `false` así como a los números naturales. También necesitaremos de un mecanismo que nos permita efectuar operaciones sobre estas constantes; para ello definiremos al conjunto $\mathcal{O}_\mathcal{P}$ que contiene al menos a los operadores primitivos que implementarán las operaciones habituales de álgebra booleana y aritmética.

Definición 17: $\mathcal{O}_\mathcal{P}$ Operadores primitivos [15]

Consideraremos al conjunto $\mathcal{O}_\mathcal{P}$ que contiene a los operadores primitivos. Diremos que $o^{(n)} \in \mathcal{O}_\mathcal{P}$ es un operador primitivo de aridad n, esto es que para ser evaluado espera recibir n argumentos que se evalúen cada uno a una constante primitiva.

Con la inclusión de los conjuntos $\mathcal{C}_\mathcal{P}$ y $\mathcal{O}_\mathcal{P}$, podemos denotar la aplicación de un operador primitivo sobre una serie de constantes, pero aún nos hace falta definir un mecanismo que efectivamente compute el resultado de esta aplicación; para esto definiremos la función δ.

Definición 18: $\delta : \mathcal{O}_\mathcal{P} \times [\mathcal{C}_\mathcal{P}] \to \mathcal{C}_\mathcal{P}$ **Evaluación de operadores [15]**

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

Para definir una instancia de la familia de ISWIM habrá que dar especificaciones de los conjuntos $\mathcal{C}_\mathcal{P}$, $\mathcal{O}_\mathcal{P}$ y de la función δ. Ya hemos dicho que de aquí en adelante supondremos que $\mathcal{C}_\mathcal{P}$ incluye por lo menos a las constantes booleanas así como a los números naturales, y por lo mismo $\mathcal{O}_\mathcal{P}$ deberá contener operadores primitivos para las operaciones habituales sobre estas constantes. También tendremos que especificar el comportamiento de la función δ de manera que implemente fielmente estos operadores.

3.2. El cálculo de ISWIM

Expandiremos el cálculo-λ puro con nuevas construcciones para formar el cálculo de ISWIM, al cual denotaremos mediante Λ_s. Incorporaremos las constantes primitivas a Λ_s con la regla `[cte]`, y los operadores primitivos con la regla `[op[M]]`.

También extenderemos el cálculo de ISWIM agregando una sentencia condicional `if-then-else` como elemento primitivo del lenguaje. Este condicional podría ser considerado como parte de $\mathcal{O}_\mathcal{P}$, pero ya que hemos elegido comprometernos con la evaluación mediante llamada por valor, esto nos obligaría a evaluar ambos casos del condicional hasta convertirlos en un valor, para posteriorment elegir cuál de estos dos regresar. La elección de incluir este condicional como elemento primitivo nos permite hacer una declaración *ad hoc* de la semántica operacional para evitar evaluar ambos casos, ya que solamente uno de ellos nos interesa, además de que si tuviéramos que evaluar ambos casos, este proceso jamás terminaría para las funciones recursivas. Agregaremos este condicional al cálculo mediante la regla `[if]`.

Definición 19: Λ_s ISWIM simple [15]

$$
\begin{array}{rcll}
\Lambda_s & = & x & \texttt{[var]} \\
& | & \dot{c} & \texttt{[cte]} \\
& | & (\lambda x.\, M_1) & \texttt{[}\lambda\texttt{x.M]} \\
& | & (M_1\ M_2) & \texttt{[app]} \\
& | & (\ o^n\ [\ M_i\ ]_{i=1}^{n}\ ) & \texttt{[op[M]]} \\
& | & (\texttt{if}\ \ M_1\ \ M_2\ \ M_3) & \texttt{[if]}
\end{array}
$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad o^n \in \mathcal{O}_{\mathcal{P}} \quad M_i \in \Lambda_s$

3.2.1. Variables libres y sustitución para Λ_s

Ahora que hemos modificado la sintaxis de nuestro cálculo tendremos que extender adecuadamente todas las funciones que definimos para el cálculo-λ puro. Afortunadamente, todas estas funciones se extienden de manera trivial.

Definición 20: $\mathcal{V}_{\mathcal{L}}$ para ISWIM simple [15]

$$
\begin{array}{rcl}
\mathcal{V}_{\mathcal{L}}(x) & = & \{x\} \\
\mathcal{V}_{\mathcal{L}}(\dot{c}) & = & \varnothing \\
\mathcal{V}_{\mathcal{L}}(\lambda x.\, M) & = & \mathcal{V}_{\mathcal{L}}(M) \setminus \{x\} \\
\mathcal{V}_{\mathcal{L}}(M\ \ N) & = & \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N) \\
\mathcal{V}_{\mathcal{L}}(\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ ) & = & \cup_{i=1}^{n}\ \mathcal{V}_{\mathcal{L}}(M_i) \\
\mathcal{V}_{\mathcal{L}}(\texttt{if}\ \ M\ \ N\ \ L) & = & \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N) \cup \mathcal{V}_{\mathcal{L}}(L)
\end{array}
$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad M, N, L \in \Lambda_s$

Definición 21: Sustitución para ISWIM simple [15]

$$x[y \leftarrow N] = \begin{cases} N & \text{si } x = y \\ x & \text{si } x \neq y \end{cases}$$

$$\dot{c}[y \leftarrow N] = \dot{c}$$

$$(\lambda x.\, M)[y \leftarrow N] = \begin{cases} \lambda x.\, M & \text{si } x = y \\ \lambda z.\, M[x \leftarrow z][y \leftarrow N] & \text{si } x \neq y \end{cases}$$
$$\text{donde} \quad z \notin \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N)$$

$$(M_1\ M_2)[y \leftarrow N] = M_1[y \leftarrow N]\ M_2[y \leftarrow N]$$
$$(\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ )[y \leftarrow N] = o^{(n)}\ [\ M_i[y \leftarrow N]\ ]_{i=1}^{n}$$
$$(\texttt{if}\ M_1\ M_2\ M_3)[y \leftarrow N] = \texttt{if}\ M_1[y \leftarrow N]\ M_2[y \leftarrow N]\ M_3[y \leftarrow N]$$

$$\text{donde } x, y \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad M, N \in \Lambda_s$$

3.3. Semántica operacional

Ahora que contamos con constantes primitivas en Λ_s resulta natural adjudicarles el estatus de *valor*, por lo que agregaremos a las constantes primitivas al conjunto de valores de ISWIM.

Definición 22: $\mathcal{V}_s$ Valores de ISWIM simple [15]

$$\begin{aligned} \mathcal{V}_s &= x \\ &\mid \dot{c} \\ &\mid \lambda x.\, M \end{aligned}$$

$$\text{donde } x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad M \in \Lambda_s$$

Para extender la semántica operacional tendremos que agregar reglas para lidiar con las construcciones `[if]` y `[op[M]]`.

Las reglas `[ev-if-cnd]`, `[ev-if-tr]` y `[ev-if-fl]` dicen que para evaluar (`if` M N L) primero tenemos que evaluar M hasta que se convierta a un valor booleano, y dependiendo de si éste es `true` o `false`, devolvemos como resultado N o L respectivamente.

Para lidiar con la construcción `[op[M]]` hemos definido las reglas `[ev-`o^n`-v]` y `[ev-`o^n`-`δ`]`, para las cuales hemos definido la función auxiliar

$$sel([M_i]) = \text{mín}(\{i \in \mathbb{N} | M_i \notin \mathcal{C}_\mathcal{P}\} \cup \{n+1\})$$

Esta función recibe como argumento una lista de elementos de Λ_s indexados con la variable i que va desde 1 hasta n, y como resultado devuelve el mínimo valor de i para el cual M_i no es una constante primitiva. Este valor lo usamos en la regla `[ev-`o^n`-v]` para determinar qué expresión evaluar a manera de que cada expresión en la lista de argumentos se evalúe de izquierda a derecha hasta que se conviertan en constantes primitivas. Si el valor que devuelve esta función fuera $n+1$ entonces sabremos que las n expresiones ya han sido reducidas a constantes primitivas por lo que podemos aplicar la función δ para obtener el valor del operador.

Definición 23: Semántica operacional para ISWIM simple [15]

$$\frac{M \to M'}{(M\ N) \to (M'\ N)}\ \texttt{[ev-app-i]}$$

$$\frac{v \in \mathcal{V} \qquad N \to N'}{(v\ N) \to (v\ N')}\ \texttt{[ev-app-d]}$$

$$\frac{v \in \mathcal{V}}{(\lambda x.\, M)\ \ v \to M[x \leftarrow v]}\ \texttt{[ev-red]}$$

$$\frac{M \to M'}{\texttt{if}\ \ M\ \ N\ \ L \to \texttt{if}\ \ M'\ \ N\ \ L}\ \texttt{[ev-if-cnd]}$$

$$\frac{}{\texttt{if true}\ \ N\ \ L \to N}\ \texttt{[ev-if-tr]}$$

$$\frac{}{\texttt{if false}\ \ N\ \ L \to L}\ \texttt{[ev-if-fl]}$$

Definición 23: Semántica operacional para ISWIM simple [15] (cont.)

$$sel([M_i]) = \text{mín}(\{i \in \mathbb{N} | M_i \notin \mathcal{C}_\mathcal{P}\} \cup \{n+1\})$$

$$\frac{j = sel(\,[M_i]\,) \quad j \leq n \quad M_j \to M'_j}{o^{(n)}\ [\ M_i\ ]_{i=1}^{n} \to o^{(n)}\ [M_1\ \ldots\ M'_j\ \ldots\ M_n]}\ \texttt{[ev-}o^n\texttt{-v]}$$

$$\frac{sel(\,[M_i]\,) = n+1}{o^{(n)}\ [\ M_i\ ]_{i=1}^{n} \to \delta(o^{(n)},\ [\ M_i\ ]_{i=1}^{n})}\ \texttt{[ev-}o^n\texttt{-}\delta\texttt{]}$$

Observemos nuevamente que todavía hay expresiones como $(y\ \lambda x.x)$ para las cuales no existe una regla en la semántica operacional que nos permita reducirlas más allá de su forma actual. En el próximo capítulo mostraremos una manera de clasificar cualquier expresión como *correcta* o *incorrecta* mediante un análisis sintáctico de manera que sepamos que las expresiones incorrectas representan cómputos sin sentido y que por lo tanto su evaluación quedará indefinida. En los Capítulos 5 y 6 refinaremos este sistema para lograr la misma seguridad en el lenguaje sin sufrir de los compromisos que inicialmente estableceremos.

Parte II

Sistemas de tipos

Capítulo 4

Iswim simplemente tipado

La sintaxis de Λ y Λ_s que hemos manejado hasta ahora ha resultado ser demasiado permisiva. Nos permite crear expresiones que para propósitos de nuestro trabajo carecen de significado. Por ejemplo, para ninguno de los siguientes programas existe una regla de la semántica operacional que nos permita reducirlos más allá de su forma actual.

- $42\ \lambda x.\, x$
- `if` 1 2 3
- `not` $[\lambda x.x]$

El problema con nuesto cálculo es que nos permite incrustar cualquier expresión como subexpresión de una más compleja; nada nos prohibe construir una aplicación de una constante primitiva sobre un valor, lo cual no tiene sentido. Lo que haremos en este capítulo será asociar a cada expresión un *tipo* si y sólo si es *correcta*. Al mostrar que a una expresión se le puede asociar un tipo, podremos asegurar que la ejecución de tal expresión no podrá fallar más que de exactamente una manera, tal y como veremos en la Sección 7.3.

El objetivo principal de este capítulo será desarrollar una serie de reglas de inferencia que nos permitirán decidir algorítmicamente mediante un análisis sintáctico, al cual llamaremos *verificación de tipos*, si una expresión tiene sentido o no, de manera que un evaluador del cálculo rechace a las expresiones incorrectas. La ventaja de este proceso es que le permitirá a un programador descubrir errores en su código de manera inmediata sin tener que ejecutarlo, dotando así al lenguaje con una seguridad inherente poco común en los lenguajes de programación contemporáneos.

4.1. Tipos

El *tipo de una expresión* es el concepto clave que nos permitirá obtener seguridad en el lenguaje [32]. La intuición detrás de esto es muy simple; particionaremos al conjunto de valores mediante tipos de manera que, por ejemplo, el tipo de un natural sea incompatible con el de un booleano. Con esto podremos rechazar expresiones como `if 1 2 3` donde se espera que la condición sea de tipo booleano y no de tipo numérico. Ya que las funciones también son valores, deberemos de asignarles un tipo a éstas tomando en cuenta el dominio y contradominio de cada función. A continuación definimos el conjunto de tipos $\mathcal{T}$.

Definición 24: $\mathcal{T}$ Tipos [15]

$$\begin{aligned} \mathcal{T} = \quad & \texttt{Bool} \\ | \quad & \texttt{Nat} \\ | \quad & T \to S \end{aligned}$$

donde $T, S \in \mathcal{T}$

Usaremos las letras mayúsculas T y S para denotar *tipos*, y escribiremos $M : T$ para decir que la expresión M es de tipo T.

El conjunto $\mathcal{T}$ ha quedado definido recursivamente. Los primeros dos tipos son los que usaremos para representar a las constantes booleanas y naturales respectivamente. El tercer tipo es que el que le asignaremos a las abstracciones-λ y diremos que es el *tipo función* [15, 32].

El tipo función está formado por otros dos tipos y una flecha que los separa. El tipo $\texttt{Nat} \to \texttt{Bool}$ corresponde al tipo de una función que recibe como argumento un natural y devuelve un booleano. Para representar funciones de varios argumentos usaremos currificación, por lo que diremos que el tipo de una función que recibe dos booleanos y devuelve un natural es $\texttt{Bool} \to (\texttt{Bool} \to \texttt{Nat})$. Usaremos la convención de que el tipo función asocia a la derecha y por lo tanto escribiremos $\texttt{Bool} \to \texttt{Bool} \to \texttt{Nat}$ en lugar de $\texttt{Bool} \to (\texttt{Bool} \to \texttt{Nat})$. Usaremos paréntesis para denotar que uno de los argumentos de la función es otra función, por ejemplo el tipo $\texttt{Nat} \to (\texttt{Nat} \to \texttt{Bool}) \to \texttt{Bool}$ corresponde al tipo de una función que toma como argumentos un natural y una función de naturales a booleanos, y que devuelve como resultado un booleano.

Para poder determinar el tipo de una expresión necesitaremos cierta información

a partir de la cual podamos deducir el resto. Primero necesitaremos de la función $\mathcal{C}_\mathcal{T}$ que toma una constante primitiva y devuelve su tipo. Ésta deberá estar definida para cada instancia de ISWIM junto con los conjuntos $\mathcal{C}_\mathcal{P}$, $\mathcal{O}_\mathcal{P}$ y la función Δ que definimos en seguida.

Definición 25: $\mathcal{C}_\mathcal{T} : \mathcal{C}_\mathcal{P} \to \mathcal{T}$ **Tipo de constantes [15]**

$$\mathcal{C}_\mathcal{T}(\dot{c}) = T \text{ tal que } \dot{c} : T$$

La función $\mathcal{C}_\mathcal{T}$ toma una constante primitiva $\dot{c} \in \mathcal{C}_\mathcal{P}$ como argumento y devuelve un tipo $T \in \mathcal{T}$ de manera que $\dot{c}{:}T$

También necesitaremos de la función Δ que toma como argumentos un operador $o^{(n)}$ y una lista de n tipos, y que devuelve como resultado el tipo correspondiente a la constante que resultaría de evaluar δ sobre $o^{(n)}$ y n constantes primitivas de los tipos requeridos.

Definición 26: $\Delta : \mathcal{O}_\mathcal{P} \times [\mathcal{T}] \to \mathcal{T}$ **Tipo de operadores [15]**

Dados un operador $o^{(n)}$ y una lista $[T_i]$ de n tipos, la función $\Delta(o^{(n)}, [T_i])$ devuelve un tipo T que corresponde al tipo de la constante primitiva que resultaría de evaluar $\delta(o^{(n)}, [\dot{c}_i])$ donde $[\dot{c}_i]$ es una lista de n constantes primitivas donde para cada una se tiene que $\dot{c}_i{:}T_i$. Es decir, que para que las funciones δ, $\mathcal{C}_\mathcal{T}$ y Δ estén en concordancia deberá de cumplirse que

$$\Delta(o^{(n)}, [T_i]) = \mathcal{C}_\mathcal{T}(\delta(o^{(n)}, [\dot{c}_i]))$$

También necesitaremos saber qué tipo de dato espera cada función. Esta información tendrá que ser proporcionada por el programador mediante anotaciones de tipo en las abstracciones-λ [15, 32]. Es por esto que tendremos que cambiar la sintaxis de las abstracciones-λ, y ahora serán de la forma

$$\lambda x{:}T.\, M \qquad \text{con } T \in \mathcal{T}$$

Tendremos que escribir $\lambda x{:}\texttt{Nat}.M$ para funciones que esperan que su argumento x sea de tipo `Nat`. Así mismo tendríamos que escribir $\lambda f : \texttt{Bool} \to \texttt{Nat}.\, M$ para funciones que esperan que el argumento f sea una función de booleanos a naturales.

Daremos reglas de inferencia para especificar las reglas de tipado[1] para este nuevo cálculo al cual llamaremos Λ_t. Empezaremos por la regla que nos permite tipar constantes primitivas [15, 32].

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\Gamma \vdash \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c})}\ \texttt{[t-cte]}$$

Esta regla debe leerse como, «si $\dot{c}$ es una constante primitiva, entonces $\dot{c}$ tiene tipo $\mathcal{C}_{\mathcal{T}}(\dot{c})$ bajo el contexto de tipado Γ». El *contexto de tipado* Γ no es relevante para esta regla y puede omitirse en su lectura. Sin embargo, para las reglas subsecuentes será importante hacer uso de él, así que su definición se da más adelante.

El tipado de las funciones es un poco más complicado. Sabemos que las funciones tienen tipo $T_1 \to T_2$, donde T_1 es el tipo del argumento que espera y T_2 es el tipo del valor que devuelve. Con las anotaciones de tipo ya sabemos cuál es el tipo de su argumento, por lo que solamente nos hace falta conocer el tipo de su resultado. Éste lo podemos encontrar al tipar M bajo la suposición de que cada instancia libre de x es de tipo T_1. Estas suposiciones forman lo que se conoce como el *contexto de tipado*.

Definición 27: Γ Contexto de tipado [32, 15]

Decimos que un *contexto de tipado* denotado por Γ es un conjunto de variables anotadas con un tipo.

$$\Gamma = \{x_1 : T_1,\ \ x_2 : T_2,\ \ \ldots,\ \ x_n : T_n\}$$

El contexto vacío lo denotaremos mediante $\varnothing$ aunque preferiremos simplemente omitirlo cuando sea vacío, escribiendo $\vdash M : T$ en lugar de $\varnothing \vdash M : T$.

A veces requeriremos *extender* el contexto de tipado con la asociación $x : T$. Esto lo denotaremos mediante

$$\Gamma,\ x : T \stackrel{\text{def}}{=} \Gamma \cup \{x : T\}$$

También usaremos a Γ como una función para obtener el tipo de alguna

[1] El vocablo *tipar* viene de la castellanización del verbo inglés *to type*. Otra popular castellanización de este verbo es *tipificar*. En este trabajo hemos optado por usar la primera opción.

Definición 27: Γ Contexto de tipado [32, 15] (cont.)

variable en el contexto de manera que

$$\Gamma(x_i) = T_i$$

Para que Γ quede bien definida como función debemos tener cuidado de tratar de manera correcta una situación como la siguiente:

Supongamos que queremos tipar la función $\lambda x : \texttt{Nat}.\lambda x : \texttt{Bool}.x$, ésta es una función que toma un valor de tipo `Nat`, uno de tipo `Bool` e ignora el primero para devolver el segundo. Como veremos a continuación, para tipar esta función deberemos de encontrar un tipo T tal que $\{x : \texttt{Bool}, x : \texttt{Nat}\} \vdash x : T$, donde $T = \Gamma(x)$. En este caso $\Gamma(x)$ no está bien definida a pesar de que quisiéramos que $\Gamma(x) = \texttt{Bool}$. Para lograr esto tenemos dos opciones:

1. Asegurando que si $\Gamma = \{x_1 : T_1, \ldots, x_n : T_n\}$, entonces $x_i \neq x_j$ siempre que $i \neq j$. Esto lo podemos lograr si en lugar de realizar la verificación de tipos sobre la expresión original M lo hacemos sobre una expresión α-equivalente, donde todas las variables ligadas han sido renombradas de manera que no haya dos variables con el mismo nombre que estén ligadas a nodos distintos. Por lo tanto, optando por este camino tiparíamos la expresión $\lambda x : \texttt{Nat}.\lambda y : \texttt{Bool}.y$ y entonces tendríamos que $\{y : \texttt{Bool}, x : \texttt{Nat}\} \vdash y : T$, donde ahora sí $T = \Gamma(y)$ está bien definida.
2. Modificando la estructura de datos de Γ, en lugar de usar un conjunto podemos usar una estructura de datos que recuerde el orden en el cual se han añadido elementos a Γ, de manera que responda a la consulta $\Gamma(x)$ devolviendo la asociación más reciente para x.

Ya que ambos enfoques funcionan, no nos precuparemos por convenir en uno de ellos y de ahora en adelante solamente supondremos que $\Gamma(x)$ está bien definida.

Ahora procedemos a dar la regla para tipar funciones [15, 32].

$$\frac{\Gamma,\ x : T_1 \ \vdash \ M : T_2}{\Gamma \ \vdash \ \lambda x : T_1.\ M \ : \ T_1 \to T_2}\ \texttt{[t-λx]}$$

Esta regla debe leerse como «si al expandir el contexto Γ con la suposición de que x es de tipo T_1, podemos concluir que M es de tipo T_2; entonces bajo el contexto Γ original $\lambda x : T_1.\ M$ tiene tipo $T_1 \to T_2$ ».

Cuando estemos intentando encontrar el tipo del cuerpo de una función nos encontraremos con variables libres, cuyo tipo podemos encontrar usando la siguiente regla [15, 32].

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\,\texttt{[t-var]}$$

Esta regla puede fallar en caso de que x no se encuentre en el contexto, lo cual solamente pasaría si x no está ligada en la expresión. Siempre que ocurra un error de tipo será porque la expresión no tiene sentido y no debe ser evaluada. En este caso el problema es que la expresión no es cerrada, o sea que no es un programa válido.

Para tipar una aplicación $M\ N$ primero necesitamos que M sea de tipo función y que N esté en el dominio de la función. En caso de que no se cumplan estas condiciones entonces reportaremos un error de tipos [15, 32].

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1}{\Gamma \vdash M\ \ N : T_2}\,\texttt{[t-app]}$$

La regla para tipar operadores primitivos depende de la función Δ [15].

$$\frac{\Gamma \vdash M_i : T_i \quad 1 \leq i \leq n}{\Gamma \vdash o^{(n)}\ [\ M_i\ ]_{i=1}^{n} : \Delta(o^{(n)}, [T_i])}\,\texttt{[t-op]}$$

Si Δ recibe argumentos inesperados entonces devolverá un error, por lo que diremos que ha ocurrido un error de tipos. Esto ocurre cuando hemos querido usar una operación primitiva sobre tipos de datos que no le corresponden.

Finalmente, para tipar una expresión condicional deberemos asegurarnos de que la condición sea de tipo `Bool` y que ambos posibles casos tengan el mismo tipo. Es necesario pedir que ambos posibles casos sean del mismo tipo ya que de lo contrario solamente podríamos saber el tipo de la expresión si conociéramos de antemano el valor de la condición booleana, lo cual es imposible a menos que mezclemos la verificación de tipos con la evaluación del cálculo [15, 32].

$$\frac{\Gamma \vdash M : \texttt{Bool} \qquad \Gamma \vdash N : T \qquad \Gamma \vdash L : T}{\Gamma \vdash \texttt{if}\ \ M\ \ N\ \ L : T}\,\texttt{[t-if]}$$

4.2. Recursión

Al introducir el sistema de tipos también introdujimos un problema, pues sin querer hemos eliminado la posibilidad de hacer recursión mediante combinadores de punto fijo. En particular hemos prohibido la autoaplicación.

Recordemos al combinador $\omega \stackrel{\text{def}}{=} \lambda x.\, x\ x$. La autoaplicación en ω fue la clave que nos permitió implementar la recursión en el cálculo-λ puro. Si quisiéramos introducir ω en nuestro nuevo cálculo con anotaciones de tipo nos encontraríamos con el siguiente problema.

Primero vemos que la aplicación $x\ x$ pide que x como función sea de tipo $T_1 \to T_2$, pero también necesitamos que el tipo de x como argumento sea T_1 para que sea compatible con el dominio de la función. Por lo que, entonces, tendríamos que x necesita ser de tipo T_1 y $T_1 \to T_2$ al mismo tiempo; lo cual es imposible. Este mismo problema aparecería al intentar tipar cualquier combinador de punto fijo [15].

La solución que tomaremos será introducir el combinador Y como un elemento primitivo del cálculo extendido, y modelar su comportamiento a través de la semántica operacional. Extenderemos la sintaxis del cálculo con el constructor `Fix` para representar a un combinador de punto fijo [15].

$$\texttt{Fix}\ M$$

La semántica operacional de `Fix` modela precisamente el comportamiento del combinador Y. Solamente queda definida para expresiones que tengan la forma $\texttt{Fix}\ \lambda f{:}T.M$. La variable f ocurre en el cuerpo de M para señalar una llamada recursiva a la misma función. Este tipo de expresiones se reducen al sustituir f en M por la misma expresión $\texttt{Fix}\ \lambda f{:}T.M$ [15].

$$\frac{}{\texttt{Fix}\ \lambda f{:}T.\, M \ \ \to \ \ M[f \leftarrow \texttt{Fix}\ \lambda f{:}T.\, M]}\ \texttt{[ev-fix]}$$

Ya que $\texttt{Fix}\ M$ solamente tiene sentido cuando M es una función, entonces deberemos verificar que M sea de tipo función, y también deberá cumplir que el tipo de su resultado sea compatible con el tipo de su argumento. Es decir, que tiene que tener el tipo $T \to T$. Si es así, entonces diremos que la expresión $\texttt{Fix}\ M$ tiene tipo T [15].

$$\frac{\Gamma \vdash M{:}T \to T}{\Gamma \vdash \texttt{Fix}\ M{:}T}\ \texttt{[t-fix]}$$

Es posible que ocurra una expresión como `Fix` $(M\ N)$, donde la aplicación $M\ N$ se reduzca a una función. Es por esto que también tendremos que considerar la siguiente regla dentro de la semántica operacional [15].

$$\frac{M \to M'}{\texttt{Fix}\ M \to \texttt{Fix}\ M'}\ \texttt{[ev-fix-b]}$$

4.3. Un ejemplo de una función recursiva

Con esto hemos terminado la especificación formal de ISWIM simplemente tipado. En el Apéndice A se encuentran las definiciones relevantes para el cálculo de Λ_t, así como su sistema de tipos y semántica operacional. Antes de mostrar los resultados relevantes a la seguridad del lenguaje, retomaremos la función factorial que definimos usando el cálculo-λ puro. En esta ocasión la definiremos usando ISWIM simplemente tipado y usaremos las reglas de tipado para elaborar una prueba de que la expresión puede ser tipada correctamente.

Definiremos la función factorial en ISWIM simplemente tipado de la siguiente manera

$$\texttt{fact} \stackrel{\text{def}}{=} \texttt{Fix}\ \lambda f{:}\texttt{Nat} \to \texttt{Nat}.\ \lambda n{:}\texttt{Nat}.\ \texttt{if}\ (\texttt{isZero}\ [n])\ 1\ (\texttt{*}\ [n, (f\ (\texttt{-}\ [n, 1]))])$$

Para este ejemplo supondremos que nuestro conjunto de operadores primitivos contiene a `isZero` que toma un natural y devuelve `true` si el valor es igual a cero y `false` en cualquier otro caso. También usamos `*` y `-` para representar multiplicación y resta de naturales respectivamente en notación prefija. Supondremos que Δ está definida de la siguiente manera para estos operadores.

$$\begin{aligned}\Delta(\texttt{isZero}, [\texttt{Nat}]) &= \texttt{Bool}\\ \Delta(\texttt{*}, [\texttt{Nat}, \texttt{Nat}]) &= \texttt{Nat}\\ \Delta(\texttt{-}, [\texttt{Nat}, \texttt{Nat}]) &= \texttt{Nat}\end{aligned}$$

Usaremos las anotaciones de tipos en las funciones para determinar nuestro contexto de tipado.

$$\Gamma = \{f{:}\texttt{Nat} \to \texttt{Nat},\ \ n{:}\texttt{Nat}\}$$

Tiparemos cada uno de los argumentos del `if`, empezando por la subexpresión `isZero` $[n]$. Para esto primero usaremos `[t-var]` para decir que n tiene tipo `Nat` y luego usaremos `[t-op]` para concluir que `isZero` $[n]$ es de tipo `Bool`.

$$\dfrac{\dfrac{}{\Gamma \vdash n : \texttt{Nat}}\,\texttt{[t-var]}}{\Gamma \vdash \texttt{isZero}\ [n] : \texttt{Bool}}\,\texttt{[t-op]}$$

Para tipar el segundo argumento del `if` basta con reiterar que la constante 1 tiene tipo `Nat` usando `[t-cte]`. El tercer argumento es un poco más complicado. Primero debemos decir que n y 1 tienen tipo `Nat`.

$$\dfrac{}{\Gamma \vdash n : \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{}{\Gamma \vdash 1 : \texttt{Nat}}\,\texttt{[t-cte]}$$

Ahora podemos combinar estos dos resultados para decir que $\texttt{-}\ [n, 1]$ también tiene tipo `Nat`.

$$\dfrac{\dfrac{}{\Gamma \vdash n : \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{}{\Gamma \vdash 1 : \texttt{Nat}}\,\texttt{[t-cte]}}{\Gamma \vdash \texttt{-}\ [n, 1] : \texttt{Nat}}\,\texttt{[t-op]}$$

Para probar que $f(\texttt{-}\ [n, 1])$ tiene tipo `Nat` primero debemos reiterar que bajo la suposición de que f tiene tipo $\texttt{Nat} \to \texttt{Nat}$, f tiene precisamente ese tipo. Luego usando usando esta información junto con el resultado que acabamos de obtener podemos ver que el tipo del paramétro es compatible con el del dominio de f. Usaremos `[t-app]` para probar que la aplicación tiene tipo `Nat`.

$$\dfrac{\dfrac{}{\Gamma \vdash f : \texttt{Nat} \to \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{\dfrac{}{\Gamma \vdash n : \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{}{\Gamma \vdash 1 : \texttt{Nat}}\,\texttt{[t-cte]}}{\Gamma \vdash (\texttt{-}\ [n, 1]) : \texttt{Nat}}\,\texttt{[t-op]}}{\Gamma \vdash f\ \ (\texttt{-}\ [n, 1]) : \texttt{Nat}}\,\texttt{[t-app]}$$

Finalmente, probamos que $(\texttt{*}\ [n, f\ \ (\texttt{-}\ [n, 1])])$ tiene tipo `Nat`.

$$\dfrac{\dfrac{}{\Gamma \vdash n : \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{\dfrac{}{\Gamma \vdash f : \texttt{Nat} \to \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{\dfrac{}{\Gamma \vdash n : \texttt{Nat}}\,\texttt{[t-var]} \qquad \dfrac{}{\Gamma \vdash 1 : \texttt{Nat}}\,\texttt{[t-cte]}}{\Gamma \vdash (\texttt{-}\ [n, 1]) : \texttt{Nat}}\,\texttt{[t-op]}}{\Gamma \vdash f\ \ (\texttt{-}\ [n, 1]) : \texttt{Nat}}\,\texttt{[t-app]}}{\Gamma \vdash (\texttt{*}\ [n, f\ \ (\texttt{-}\ [n, 1])]) : \texttt{Nat}}\,\texttt{[t-op]}$$

Ahora que hemos probado que el primer argumento del `if` es de tipo `Bool` y los otros dos son de tipo `Nat`, podemos concluir que el `if` es de tipo `Nat`.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \texttt{isZero}\ [n] : \texttt{Bool}} \qquad \dfrac{}{1 : \texttt{Nat}}\,\texttt{[t-cte]} \qquad \dfrac{\vdots}{\Gamma \vdash (\texttt{*}\ [n, f\ \ (\texttt{-}\ [n, 1])]) : \texttt{Nat}}}{\Gamma \vdash \texttt{if}\ \ (\texttt{isZero}\ \ n)\ \ 1\ \ (\texttt{*}\ [n, (f\ \ (\texttt{-}\ [n, 1]))]) : \texttt{Nat}}\,\texttt{[t-if]}$$

A partir de este punto debemos empezar a reducir el contexto de tipado, las suposiciones que hemos hecho quedarán plasmadas en las anotaciones de tipo de las abstracciones-λ.

$$\dfrac{\dfrac{\dfrac{\vdots}{\{f:\texttt{Nat}\to\texttt{Nat},\ n:\texttt{Nat}\}\vdash \texttt{if (isZero } n\texttt{) 1 (*}\,[n,(f\ \texttt{(-}\,[n,1]\texttt{))}]\texttt{)}:\texttt{Nat}}\,\texttt{[t-}\lambda\texttt{x]}}{\{f:\texttt{Nat}\to\texttt{Nat}\}\vdash \lambda n{:}\texttt{Nat.if (isZero } n\texttt{) 1 (*}\,[n,(f\ \texttt{(-}\,[n,1]\texttt{))}]\texttt{)}:\texttt{Nat}\to\texttt{Nat}}\,\texttt{[t-}\lambda\texttt{x]}}{\varnothing\vdash \lambda f{:}\texttt{Nat}\to\texttt{Nat}.\lambda n{:}\texttt{Nat.if (isZero } n\texttt{) 1 (*}\,[n,(f\ \texttt{(-}\,[n,1]\texttt{))}]\texttt{)}:(\texttt{Nat}\to\texttt{Nat})\to(\texttt{Nat}\to\texttt{Nat})}\,\texttt{[t-}\lambda\texttt{x]}$$

Con esto hemos obtenido que la expresión tiene tipo $(\texttt{Nat}\to\texttt{Nat})\to(\texttt{Nat}\to\texttt{Nat})$, el cual es un tipo poco intuitivo para la función factorial. Esto es porque aún nos hace falta considerar el constructor `Fix` que envuelve a toda la expresión.

$$\dfrac{\dfrac{\vdots}{\varnothing\vdash \lambda f{:}\texttt{Nat}\to\texttt{Nat}.\lambda n{:}\texttt{Nat.if (isZero } n\texttt{) 1 (*}\,[n,(f\ \texttt{(-}\,[n,1]\texttt{))}]\texttt{)}:(\texttt{Nat}\to\texttt{Nat})\to(\texttt{Nat}\to\texttt{Nat})}\,\texttt{[t-fix]}}{\varnothing\vdash \texttt{Fix }\lambda f{:}\texttt{Nat}\to\texttt{Nat}.\lambda n{:}\texttt{Nat.if (isZero } n\texttt{) 1 (*}\,[n,(f\ \texttt{(-}\,[n,1]\texttt{))}]\texttt{)}:\texttt{Nat}\to\texttt{Nat}}\,\texttt{[t-fix]}$$

Con esto hemos obtenido que la función `fact` tiene el tipo que esperaríamos de ella, esto es, que recibe un natural y devuelve otro natural. Al llegar a una conclusión de la forma $\varnothing\vdash M:T$, podemos estar seguros de que el programa tiene un tipo adecuado, lo cual nos asegura que su ejecución no puede quedar indefinida en ningún momento.

4.4. Seguridad

En esta sección mostraremos los resultados principales que nos proporciona el sistema de tipos para brindarle seguridad al lenguaje.

Los primeros lemas que estableceremos dicen que las reglas de tipado y las de la semántica operacional son bidireccionales en el sentido de que si la conclusión es cierta, entonces necesariamente las premisas también lo son.

Lema 4.4.1 (Inversión de tipos [32])**.** *Para cada relación de tipado podemos invertir el juicio a manera de que si la conclusión es cierta, entonces los antecedentes también lo son, es decir:*

1. *Si* $\Gamma\vdash x{:}T$*, entonces* $x\in\Gamma$ *y* $\Gamma(x)=T$
2. *Si* $\Gamma\vdash \dot{c}{:}T$*, entonces* $\dot{c}\in\mathcal{C}_{\mathcal{P}}$ *y* $\mathcal{C}_{\mathcal{T}}(\dot{c})=T$
3. *Si* $\Gamma\vdash \lambda x{:}T_1.M:T$*, entonces* $T=T_1\to T_2$*, donde* $\Gamma,x{:}T_1\vdash M:T_2$

4. *Si* $\Gamma \vdash M\ \ N : T$, *entonces existe* T_1 *tal que* $\Gamma \vdash M : T_1 \to T$ *y* $\Gamma \vdash N : T_1$
5. *Si* $\Gamma \vdash o^{(n)}[\ M_i\ ] : T$, *entonces* $\Gamma \vdash M_i : T_i \quad \forall i \in \{1 \dots n\}$ *tal que* $\Delta(o^{(n)}, [T_i]) = T$
6. *Si* $\Gamma \vdash \texttt{if}\ M\ N\ L : T$, *entonces* $\Gamma \vdash M : \texttt{Bool}$, $\Gamma \vdash N : T$ *y* $\Gamma \vdash L : T$
7. *Si* $\Gamma \vdash \texttt{Fix}\ M : T$, *entonces* $\Gamma \vdash M : T \to T$

Demostración. Directamente de las reglas de tipado. □

Lema 4.4.2 (Inversión de la semántica operacional [15])**.** *Para cada regla de la semántica operacional podemos invertir el juicio a manera de que si la conclusión es cierta, entonces los antecedentes también lo son, es decir:*

1. *Si* $(M\ N) \to (M'\ N)$, *entonces* $M \to M'$
2. *Si* $(v\ N) \to (v\ N')$, *entonces* $v \in \mathcal{V}$ *y* $N \to N'$
3. *Si* $(\lambda x{:}T.M)\ v \to M[x \leftarrow v]$, *entonces* $v \in \mathcal{V}$
4. *Si* $\texttt{if}\ M\ N\ L \to \texttt{if}\ M'\ N\ L$, *entonces* $M \to M'$
5. *Si* $\texttt{if}\ v\ N\ L \to N$, *entonces* $v = \texttt{true}$
6. *Si* $\texttt{if}\ v\ N\ L \to L$, *entonces* $v = \texttt{false}$
7. *Si* $o^{(n)}\ [M_1 \dots M_j \dots M_n] \to o^{(n)}\ [M_1 \dots M'_j \dots M_n]$, *entonces* $M_j \to M'_j$
8. *Si* $o^{(n)}\ [M_i] \to \delta(o^{(n)}, [M_i])$, *entonces* $M_i \in \mathcal{C}_\mathcal{P}\ \ \forall i \in \{1 \dots n\}$
9. *Si* $\texttt{Fix}\ M \to \texttt{Fix}\ M'$, *entonces* $M \to M'$

Demostración. Directamente de las reglas de la semántica operacional. □

Lema 4.4.3 (Monotonía [32])**.** *Si* $M \in \Lambda_t$ *y* x *es una variable tal que* $x \notin \mathcal{V}_\mathcal{L}(M)$, *entonces* $\Gamma \vdash M : T$ *si y sólo si* $\Gamma, x{:}T' \vdash M : T$

Demostración. La demostración se hace mediante una inducción sencilla sobre $\Gamma \vdash M : T$ para ambas direcciones de la equivalencia y se omite. Para el sentido inverso será necesario ocupar la hipótesis $x \notin \mathcal{V}_\mathcal{L}(M)$. □

El próximo lema nos permitirá estar seguros de que si hemos logrado tipar una expresión bajo la suposición de que alguna variable tiene cierto tipo, entonces al sustituir la variable por una expresión que tenga el mismo tipo de nuestra suposición no se altera el tipado de la expresión original.

Lema 4.4.4 (Sustitución [32]). *Si* $\Gamma, x : T_1 \vdash M : T_2$ *y* $\Gamma \vdash N : T_1$, *entonces* $\Gamma \vdash M[x \leftarrow N] : T_2$

Demostración. Inducción sobre M.

Para los casos base consideraremos a las variables y constantes.

- `[var]` Si $M = z$ es una variable, entonces consideraremos dos casos: cuando $x = z$ y $x \neq z$
  - Si $x = z$ entonces tenemos por hipótesis que $\Gamma, x : T_1 \vdash x : T_2$. Por inversión de tipos tenemos que $x \in (\Gamma, x : T_1)$ y $\Gamma(x) = T_2$, pero como $x : T_1$ está en el contexto de tipado, tenemos entonces que $T_1 = T_2$.

    Tenemos también que la sustitución $x[x \leftarrow N] = N$ y por hipótesis tenemos que $\Gamma \vdash N : T_1$, pero como $T_1 = T_2$, concluimos que $\Gamma \vdash N : T_2$, que es lo que queríamos demostrar.
  - Si $x \neq z$ entonces tenemos por hipótesis que $\Gamma, x : T_1 \vdash z : T_2$. Al aplicar la sustitución obtenemos $z[x \leftarrow N] = z$ porque $z \neq x$. Como x no figura en z, entonces por monotonía de Γ tenemos que $\Gamma \vdash z : T_2$ que es lo queríamos demostrar.
- `[cte]` Si $M = \dot{c}$ es una constante, tenemos que la sustitución $\dot{c}[x \leftarrow N] = \dot{c}$. Por hipótesis tenemos que $\Gamma, x : T_1 \vdash \dot{c} : T_2$ y como x no figura en $\dot{c}$, entonces por monotonía de Γ podemos reducir el contexto de tipado y obtener que $\Gamma \vdash \dot{c} : T_2$.

  Supongamos para el paso inductivo que M' cumple con la propiedad de sustitución.
- `[λx:T.M]` Supongamos que $M = \lambda z : T.M'$. Supongamos sin pérdida de generalidad (por α-equivalencia) que $x \neq z$ y $z \notin \mathcal{V}_{\mathcal{L}}(N)$.

  Tenemos por hipótesis que $\Gamma, x : T_1 \vdash \lambda z : T.M' : T_2$ por lo que $T_2 = T \to T'$ para algún T'.

  Por el lema de monotonía podemos expandir el contexto de tipado con $z : T$ para tipar a M' $\Gamma, x : T_1, z : T \vdash M' : T'$; y para mayor claridad podemos reordenar el contexto de tipado en la sentencia anterior y obtener

  $$\Gamma, z : T, x : T_1 \vdash M' : T'$$

  Usando la hipótesis $\Gamma \vdash N : T_1$ y por el lema de monotonía, podemos expandir su contexto de tipado a

  $$\Gamma, z : T \vdash N : T_1$$

  De los dos juicios anteriores y por hipótesis de inducción en M' podemos concluir que

  $$\Gamma, z : T \vdash M'[x \leftarrow N] : T'$$

De lo anterior y usando la regla de tipado `[t-λx]` llegamos a que

$$\Gamma \vdash \lambda z{:}T.(M'[x \leftarrow N]){:}T \to T'$$

Finalmente, aplicando la definición de sustitución obtenemos que

$$\Gamma \vdash (\lambda z{:}T.M')[x \leftarrow N]{:}T \to T'$$

Recordando que $T_2 = T \to T'$ llegamos a lo que queríamos demostrar.

$$\Gamma \vdash (\lambda z{:}T.M')[x \leftarrow N]{:}T_2$$

- Los demas casos son muy similares a los anteriores y se omiten.

□

El siguiente teorema nos dice que si una expresión puede ser tipada correctamente, y ésta no es un valor, entonces al realizar un paso de evaluación la expresión obtenida preserva el tipo de la anterior.

Teorema 4.4.1 (Preservación de tipos [32])**.** *Si* $\vdash M{:}T$ *y* $M \to N$*, entonces* $\vdash N{:}T$

Demostración. Inducción sobre las reglas de la semántica operacional.

Supongamos por inducción que M y N cumplen la propiedad de preservación.

- `[ev-app-i]` Supongamos que $\vdash (M\ N){:}T$ y que $(M\ N) \to (M'\ N)$.
  Por inversión de tipos existe T_1 tal que $\vdash M{:}T_1 \to T$ y $\vdash N{:}T_1$.
  Por inversión de la semántica operacional $M \to M'$.
  Por hipótesis de inducción sobre M tenemos que $\vdash M'{:}T_1 \to T$, y como $\vdash N{:}T_1$ concluimos por la regla `[t-app]` que $\vdash (M'\ N){:}T$.
- `[ev-app-d]` Supongamos que $\vdash (v\ N){:}T$ y que $(v\ N) \to (v\ N')$.
  Por inversión de tipos existe T_1 tal que $\vdash v{:}T_1 \to T$ y $\vdash N{:}T_1$.
  Por inversión de la semántica operacional $v \in \mathcal{V}$ y $N \to N'$.
  Por hipótesis de inducción sobre N tenemos que $\vdash N'{:}T_1$ y como $\vdash v{:}T_1 \to T$, concluimos por la regla `[t-app]` que $\vdash (v\ N'){:}T$
- `[ev-red]` Supongamos que $\vdash ((\lambda x{:}T_1.M)\ v){:}T$ y $(\lambda x{:}T_1.M)\ v \to M[x \leftarrow v]$.
  Por inversión de tipos $\vdash v{:}T_1$ y $\vdash \lambda x{:}T_1.M{:}T_1 \to T$, y una vez más por inversión de tipos tenemos que $\{x{:}T_1\} \vdash M{:}T$.
  Finalmente, por el lema de sustitución tenemos que $\vdash M[x \leftarrow v]{:}T$

- Los casos restantes son análogos a los anteriores y se omiten.

□

A partir del teorema de preservación podemos concluir que si una expresión tiene tipo T, entonces si al evaluar la expresión logramos reducirla a un valor, entonces necesariamente éste será también de tipo T.

Corolario 4.4.1. *Si* $\vdash M : T$ *y* $M \to^* v$ *donde* $v \in \mathcal{V}$ *es un valor, entonces* $\vdash v : T$

Por último, es importante mencionar que existe un teorema que se conoce como la propiedad de *progreso* y es el que brinda la seguridad al lenguaje diciendo que si un programa M puede ser tipado correctamente, entonces solamente hay tres posibles resultados en su evaluación:

1. La ejecución de M termina eventualmente cuando se reduce a un valor.
2. La ejecución de M falla al evaluar un operador primitivo cuando $\delta(o^{(n)}, [\dot{c}_i])$ no esté definida.
3. La ejecución de M jamás termina y siempre existe un M' tal que $M \to M'$.

Por el momento omitiremos el enunciado y la prueba de este teorema ya que aún nos hace falta desarrollar algunos conceptos necesarios para la prueba. En el Capítulo 6 enunciaremos y probaremos este teorema para un sistema de tipos más sofisticado.

Con los resultados anteriores hemos mostrado que usando el sistema de tipos desarrollado en este capítulo podemos clasificar un programa M como correcto si y sólo si existe un tipo T tal que $\vdash M : T$. En los próximos dos capítulos mejoraremos el sistema de tipos para deshacernos de las anotaciones de tipo en las abstracciones-λ e introducir polimorfismo paramétrico en el lenguaje.

Capítulo 5

Inferencia de tipos

En el capítulo anterior se desarrolló un sistema de tipos para el cálculo de ISWIM que nos permite tener la certeza de que si un programa puede ser tipado, entonces su ejecución no podrá quedar indefinida salvo cuando la función δ no esté definida para alguna combinación particular de operando y operadores. Este sistema de tipos tiene algunos incovenientes que resolveremos en este capítulo.

Recordemos que en el cálculo-λ puro definimos la función identidad como $\lambda x.\, x$. Esta función tiene la propiedad de devolver como resultado exactamente el mismo argumento que recibió [15]. En el capítulo anterior introdujimos anotaciones de tipos en las abstracciones-λ, por lo que si quisiéramos escribir una función identidad en ISWIM simplemente tipado, tendríamos que escribir una función identidad distinta para cada tipo concreto que tengamos, por ejemplo:

$$\lambda x{:}\texttt{Bool}.x$$
$$\lambda x{:}\texttt{Nat}.x$$

Recordando que las funciones pueden recibir como argumento otras funciones, entonces nuestro problema se vuelve infinitamente más complicado pues tendríamos que agregar una función identidad distinta para cada tipo posible. Naturalmente no es razonable definir una cantidad infinita de funciones esencialmente idénticas salvo por sus anotaciones de tipo. La solución que daremos será olvidarnos de las anotaciones de tipos y quedarnos con una sola función como en el cálculo-λ puro cuyo tipo sea $\texttt{A} \to \texttt{A}$, donde $\texttt{A}$ es una *variable de tipo* la cual podamos *instanciar* a un tipo concreto como $\texttt{Bool}$ o $\texttt{Nat} \to \texttt{Nat}$ [1].

[1] A pesar que que introducimos variables de tipo, esto no significa que contemos con polimorfismo paramétrico. Véase el Capítulo 6.

En lugar de depender de las anotaciones de tipos para determinar el tipo del argumento que espera cada función, haremos un análisis del cuerpo de la función para *inferir* esta información. A este proceso se le conoce como *inferencia de tipos* [32].

El propósito de este capítulo es dotar a ISWIM con un sistema de tipos que no tenga injerencia en la sintaxis del cálculo y que nos permita tipar una expresión mediante inferencia de tipos. El sistema de tipos que daremos fue descubierto por Hindley [18] en su estudio de la lógica combinatoria y más tarde fue redescubierto independientemente por Milner en el contexto de los lenguajes de programación. Este sistema de tipos fue retomado por Damas quien lo formalizó en su tesis doctoral [14, 13]; formando así lo que hoy se conoce como el sistema de tipos de Damas–Hindley–Milner. Este sistema de tipos fue fundamental para el desarrollo de los primeros lenguajes de programación funcionales como ML [26] y estableció la base para sistemas de tipos más sofisticados como el de Haskell [24]. En éste y el próximo capítulo desarrollaremos el sistema de tipos de Damas–Hindley–Milner para incorporarlo a ISWIM.

5.1. Variables de tipo

La clave que nos permitirá deshacernos de las anotaciones de tipo en el cálculo es la introducción de *variables de tipo* en el sistema de tipos.

Definición 28: $\mathcal{T}_\nu$ Variables de tipo [32]

Denotaremos al conjunto de *variables de tipo* mediante $\mathcal{T}_\nu$ y usaremos letras mayúsculas como identificadores para cada variable de tipo. Supondremos también que disponemos de una cantidad infinita de identificadores.

$$\mathcal{T}_\nu = \texttt{A} \mid \texttt{B} \mid \texttt{C} \quad \ldots$$

Modificaremos el conjunto de tipos $\mathcal{T}$ para agregar las variables de tipo.

Definición 29: $\mathcal{T}$ Tipos [32]

$$\begin{aligned}\mathcal{T} = \quad & \texttt{Bool} \\ \mid \quad & \texttt{Nat} \\ \mid \quad & \texttt{A} \\ \mid \quad & T_1 \to T_2\end{aligned}$$

donde $T_1, T_2 \in \mathcal{T} \quad \texttt{A} \in \mathcal{T}_\nu$

Si un tipo no contiene variables de tipo diremos que es un *tipo concreto*.

También necesitaremos de la función $\mathcal{V}_\tau$ para extraer el conjunto de las variables de tipo presentes en un tipo T.

Definición 30: $\mathcal{V}_\tau(T)$ Variables de tipo en un tipo [32]

Definimos la función $\mathcal{V}_\tau$ que toma un tipo T y devuelve el conjunto de las variables de tipo contenidas en T de la siguiente manera

$$\begin{aligned}\mathcal{V}_\tau(\texttt{Bool}) &= \varnothing \\ \mathcal{V}_\tau(\texttt{Nat}) &= \varnothing \\ \mathcal{V}_\tau(\texttt{A}) &= \{\texttt{A}\} \\ \mathcal{V}_\tau(T_1 \to T_2) &= \mathcal{V}_\tau(T_1) \cup \mathcal{V}_\tau(T_2)\end{aligned}$$

donde $\texttt{A} \in \mathcal{T}_\nu \quad T_1, T_2 \in \mathcal{T}$

Primero veremos mediante un ejemplo la intuición detrás del algoritmo. Consideremos la expresión $\lambda x.\lambda y.\lambda z.\ (x\ z)\ \ (y\ z)$. Podemos reconstruir el tipo de cada subexpresión desde abajo hacia arriba de manera que cada que tengamos duda del tipo de una expresión, le asignemos una variable de tipo nueva, la cual posteriormente podrá ser instanciada a un tipo concreto.

Empezaremos por la subexpresión $(x\ z)$. No tenemos manera alguna de saber qué tipo tiene esta aplicación ni ninguna de las dos variables que la componen, y por lo tanto, tiparemos cada variable con una variable de tipo distinta. Así diremos que $x : \texttt{A}$ y $z : \texttt{B}$. Tampoco sabemos cuál es el tipo de la aplicación $(x\ z)$, por lo que también le asignaremos una nueva variable de tipo y diremos que

$(x\ z) : \mathtt{C}$. La parte crucial de este proceso es que sabemos que $\mathtt{A}$ tiene que ser de tipo función, y no sólo eso, sino que $\mathtt{A} = \mathtt{B} \to \mathtt{C}$. A esta igualdad le llamaremos una *restricción* sobre los tipos inferidos.

Podemos repetir un proceso similar para la aplicación $(y\ z)$ salvo por el hecho que debemos recordar que ya le hemos asignado un tipo a la variable z, por lo que también en esta subexpresión consideraremos que $z : \mathtt{B}$, y como y es una variable que no habíamos visto, a ésta le daremos un nuevo tipo; $y : \mathtt{D}$. De manera similar diremos que la aplicación $(y\ z) : \mathtt{E}$, y generamos una nueva restricción $\mathtt{D} = \mathtt{B} \to \mathtt{E}$.

Finalmente, aplicamos el mismo proceso una vez más para tipar $(x\ z)\ \ (y\ z) : \mathtt{F}$ generando la restricción $\mathtt{C} = \mathtt{E} \to \mathtt{F}$.

$$\dfrac{\dfrac{x : \mathtt{A} \quad z : \mathtt{B}}{(x\ z) : \mathtt{C}} \qquad \dfrac{y : \mathtt{D} \quad z : \mathtt{B}}{(y\ z) : \mathtt{E}}}{(x\ z)\ \ (y\ z) : \mathtt{F}}$$

Ahora podríamos concluir que $\lambda x.\lambda y.\lambda z.\ (x\ z)\ (y\ z) : \mathtt{A} \to \mathtt{D} \to \mathtt{B} \to \mathtt{F}$, lo cual solamente nos dice que ésta es una función que toma tres cosas y devuelve otra. En realidad esto no es un tipo válido, ya que no podremos usar las reglas de la Definición 71 (Apéndice A) para llegar a la misma conclusión. Sin embargo, hemos acumulado una serie de restricciones que nos ayudarán a reconstuir un tipo que sí sea válido bajo esas reglas.

- $\mathtt{A} = \mathtt{B} \to \mathtt{C}$
- $\mathtt{D} = \mathtt{B} \to \mathtt{E}$
- $\mathtt{C} = \mathtt{E} \to \mathtt{F}$

Podemos sustituir la tercera ecuación en la primera para obtener.

- $\mathtt{A} = \mathtt{B} \to \mathtt{E} \to \mathtt{F}$
- $\mathtt{D} = \mathtt{B} \to \mathtt{E}$

Finalmente podemos sustituir estas ecuaciones en el tipo $\mathtt{A} \to \mathtt{D} \to \mathtt{B} \to \mathtt{F}$, para obtener $(\mathtt{B} \to \mathtt{E} \to \mathtt{F}) \to (\mathtt{B} \to \mathtt{E}) \to \mathtt{B} \to \mathtt{F}$. A este tipo podemos aplicarle un renombre con fines cosméticos para obtener que

$$\lambda x.\lambda y.\lambda z.\ (x\ z)\ \ (y\ z) : (\mathtt{A} \to \mathtt{B} \to \mathtt{C}) \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{A} \to \mathtt{C}$$

Este tipo sí es válido bajo las reglas de la Definición 71 (Apéndice A).

El tipo que obtuvimos está dado en términos de variables de tipos, y le llamaremos el *tipo principal* de la expresión. La idea de esto es que si logramos obtener el tipo principal de una expresión, entonces podemos dar una *sustitución de tipos* con la cual remplacemos las variables de tipo por tipos concretos, de manera que obtengamos una expresión que también esté correctamente tipada. Si tenemos el tipo principal de una expresión, entonces cualquier otro tipo correcto para la expresión puede ser expresado en términos del tipo principal y una sustitución apropiada.

5.2. Sustitución y restricciones de tipos

El objetivo principal de usar variables de tipo es que podamos sustituir una variable de tipo por algún otro tipo de manera que podamos instanciar, por ejemplo, el tipo $\mathtt{A} \to \mathtt{B}$ al tipo concreto $\mathtt{Nat} \to \mathtt{Bool}$ mediante una sustitución de tipos.

Definición 31: σ Sustitución de tipos [32]

Diremos que $[\mathtt{A}_1 \mapsto T_1, \mathtt{A}_2 \mapsto T_2, \ldots, \mathtt{A}_n \mapsto T_n]$, donde $\mathtt{A}_i \in \mathcal{T}_\mathcal{V}$ y $T_i \in \mathcal{T}$ con $\mathtt{A}_i \neq \mathtt{A}_j$ siempre que $i \neq j$, es una *sustitución de tipos* y la abreviaremos mediante la letra σ. Una sustitución de tipos σ puede ser aplicada a un tipo de la siguiente manera:

$$\begin{aligned}
&\sigma(\mathtt{Nat}) = \mathtt{Nat} \\
&\sigma(\mathtt{Bool}) = \mathtt{Bool} \\
&\sigma(\mathtt{A}) = \begin{cases} T & \text{si } (\mathtt{A} \mapsto T) \in \sigma \\ \mathtt{A} & \text{si } (\mathtt{A} \mapsto T) \notin \sigma \end{cases} \\
&\sigma(T \to S) = \sigma(T) \to \sigma(S)
\end{aligned}$$

$$\text{donde } \mathtt{A} \in \mathcal{T}_\mathcal{V} \quad S, T \in \mathcal{T}$$

Definiremos el *dominio* y *rango* de una sustitución σ de la siguiente manera

- $dom([\mathtt{A}_1 \mapsto T_1, \ldots, \mathtt{A}_n \mapsto T_n]) = \{\mathtt{A}_1, \ldots, \mathtt{A}_n\}$
- $ran([\mathtt{A}_1 \mapsto T_1, \ldots, \mathtt{A}_n \mapsto T_n]) = \{T_1, \ldots, T_n\}$

Finalmente, también definiremos el *rango de variables de tipo* de una sustitución como el conjunto de variables de tipo que figuran en el rango de una sustitución

Definición 31: σ **Sustitución de tipos [32] (cont.)**

- $ran_{\mathcal{T}_\mathcal{V}}([\mathtt{A}_1 \mapsto T_1, \ldots, \mathtt{A}_n \mapsto T_n]) = \bigcup_{i=1}^{n} \mathcal{V}_\tau(T_i)$

Más adelante necesitaremos componer dos sustituciones por lo que definiremos esta operación.

Definición 32: $\sigma_2 \circ \sigma_1$ **Composición de sustituciones [32]**

Si σ_1 y σ_2 son sustituciones de tipo de la forma

$$\sigma_1 = [\mathtt{A}_1 \mapsto T_1, \ldots, \mathtt{A}_n \mapsto T_n]$$
$$\sigma_2 = [\mathtt{A}_i \mapsto T'_i, \ldots, \mathtt{A}_j \mapsto T'_j, \mathtt{B}_1 \mapsto S_1, \ldots, \mathtt{B}_m \mapsto S_m]$$

$$\text{donde } dom(\sigma_1) \cap dom(\sigma_2) = \{\mathtt{A}_i, \ldots, \mathtt{A}_j\}$$

Entonces definimos la *composición* $\sigma_2 \circ \sigma_1$ de ambas sustituciones de la siguiente manera

$$\sigma_2 \circ \sigma_1 = [\mathtt{A}_1 \mapsto \sigma_2(T_1), \ldots, \mathtt{A}_n \mapsto \sigma_2(T_n), \mathtt{B}_1 \mapsto S_1, \ldots, \mathtt{B}_m \mapsto S_m]$$

Obsérvese que aplicar la composición $\sigma_2 \circ \sigma_1$ sobre un tipo T tiene el mismo efecto que aplicar σ_2 sobre $\sigma_1(T)$; o sea que

- $(\sigma_2 \circ \sigma_1)(T) = \sigma_2(\sigma_1(T))$

Convenimos también en que la composición de sustituciones asocia hacia la derecha

- $\sigma_3 \circ \sigma_2 \circ \sigma_1 = \sigma_3 \circ (\sigma_2 \circ \sigma_1)$

Diremos que σ_1 y σ_2 son sustituciones *disjuntas* si cumplen con las siguientes propiedades

- $dom(\sigma_1) \cap dom(\sigma_2) = \varnothing$

Definición 32: $\sigma_2 \circ \sigma_1$ **Composición de sustituciones [32] (cont.)**

- $dom(\sigma_1) \cap ran_{\mathcal{T}_\mathcal{V}}(\sigma_2) = \varnothing$
- $dom(\sigma_2) \cap ran_{\mathcal{T}_\mathcal{V}}(\sigma_1) = \varnothing$

Obsérvese que si σ_1 y σ_2 son disjuntas, entonces la composición conmuta

- $\sigma_2 \circ \sigma_1 = \sigma_1 \circ \sigma_2$

Por ejemplo, si tenemos las sustituciones

- $\sigma_1 = [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}]$
- $\sigma_2 = [\mathtt{B} \mapsto \mathtt{Nat} \to \mathtt{Bool}]$

Podemos observar que no son disjuntas ya que $dom(\sigma_2) \cap ran_{\mathcal{T}_\mathcal{V}}(\sigma_1) = \{\mathtt{B}\}$ y por lo tanto su composición no conmuta, como se muestra a continuación

- $\sigma_2 \circ \sigma_1 = [\mathtt{A} \mapsto (\mathtt{Nat} \to \mathtt{Bool}) \to \mathtt{C},\ \mathtt{B} \mapsto \mathtt{Nat} \to \mathtt{Bool}]$
- $\sigma_1 \circ \sigma_2 = [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C},\ \mathtt{B} \mapsto \mathtt{Nat} \to \mathtt{Bool}]$

También definiremos el concepto de restricciones de tipos que mencionamos anteriormente.

Definición 33: $\mathcal{R}$ **Restricciones de tipo [32]**

Diremos que una ecuación de la forma $T = S$ donde $T, S \in \mathcal{T}$ es una *restricción de tipos*.

Un *conjunto de restricciones* es un conjunto de la forma

$$\mathcal{R} = \{T_1 = S_1, \ldots, T_n = S_n\}$$

donde cada ecuación $T_i = S_i$ es una restricción de tipos.

Abusaremos de la notación para definir la función $\mathcal{V}_\tau$ de manera que también aplique sobre restricciones de tipo, devolviendo el conjunto de variables de tipo presentes en un conjunto de restricciones.

Definición 34: $\mathcal{V}_\tau(\mathcal{R})$ Variables de tipo en $\mathcal{R}$ [32]

Si $\mathcal{R} = \{T_1 = S_1, \ldots, T_n = S_n\}$ es un conjunto de restricciones de tipo, entonces definimos la función $\mathcal{V}_\tau(\mathcal{R})$ que devuelve el conjunto de variables de tipo contenidas en $\mathcal{R}$ de la siguiente manera.

$$\mathcal{V}_\tau(\{T_1 = S_1, \ldots, T_n = S_n\}) = \bigcup_{i=1}^{n} (\mathcal{V}_\tau(T_i) \cup \mathcal{V}_\tau(S_i))$$

También nos interesará aplicar una sustitución de tipos sobre otras estructuras por lo que seguiremos abusando de la notación y definiremos la aplicación de sustituciones sobre un contexto de tipado Γ y un conjunto de restricciones $\mathcal{R}$.

Definición 35: $\sigma(\Gamma)$ Sustitución de tipos sobre Γ [32]

Una sustitución de tipos σ también puede ser aplicada sobre un contexto de tipado Γ de la siguiente manera

$$\sigma(\{x_1 : T_1 \ldots x_n : T_n\}) = \{x_1 : \sigma(T_1) \ldots x_n : \sigma(T_n)\}$$

Definición 36: $\sigma(\mathcal{R})$ Sustitución de tipos sobre $\mathcal{R}$ [32]

Una sustitución de tipos $\mathcal{R}$ también puede ser aplicada sobre un conjunto de restricciones $\mathcal{R}$ de la siguiente manera

$$\sigma(\{T_1 = S_1, \ldots, T_n = S_n\}) = \{\sigma(T_1) = \sigma(S_1), \ldots, \sigma(T_n) = \sigma(S_n)\}$$

5.3. Generación de restricciones

A continuación daremos un conjunto de reglas similares a las que damos en la Definición 71 (Apéndice A), pero en lugar de obtener expresiones como $\Gamma \vdash M : T$, obtendremos expresiones de la forma $\Gamma \models M : T|_{\mathcal{R}}$ donde $\mathcal{R}$ es un conjunto de restricciones de tipo. Estas expresiones denotan que en el contexto Γ, M tiene tipo T bajo el conjunto de restricciones $\mathcal{R}$.

Para dar las reglas de este algoritmo consideraremos al conjunto $\mathbb{N}_\mathbb{V}$, el cual

contiene todas la variables de tipo que no hemos usado en ninguna parte de la derivación o en alguna de las subderivaciones. Supondremos que cuando tomemos una variable $\mathtt{Z} \in \mathbb{N}_{\mathbb{V}}$, ésta se elimina del conjunto $\mathbb{N}_{\mathbb{V}}$ por lo que no podremos volver a elegir la misma variable en ningún momento de la derivación. En la Sección 8.2 veremos cómo implementar este algoritmo en Haskell y mostraremos un mecanismo eficiente y elegante para obtener de manera determinista una variable de tipo nueva cada que la necesitemos.

Definición 37: $\mathbb{N}_{\mathbb{V}}$ Variables frescas

Definimos al conjunto $\mathbb{N}_{\mathbb{V}} \subset \mathcal{T}_{\nu}$ como un conjunto infinito de variables de tipo, el cual tiene la propiedad de que cada vez que tomemos una variable $\mathtt{Z} \in \mathbb{N}_{\mathbb{V}}$ ésta se eliminará inmediatamente del conjunto $\mathbb{N}_{\mathbb{V}}$. De esta manera podemos estar seguros de que $\mathtt{Z}$ será usada exactamente una vez.

Definición 38: Generación de restricciones [32]

$$\frac{\Gamma(x) = T}{\Gamma \models x : T \mid \varnothing}\ \texttt{[c-var]}$$

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\Gamma \models \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c}) \mid \varnothing}\ \texttt{[c-cte]}$$

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \Gamma, x : \mathtt{Z} \models M : T \mid \mathcal{R}}{\Gamma \models \lambda x.M : \mathtt{Z} \to T \mid \mathcal{R}}\ \texttt{[c-λx]}$$

$$\frac{\Gamma \models M : T_1 \mid \mathcal{R}_1 \quad \Gamma \models N : T_2 \mid \mathcal{R}_2 \quad \mathtt{Z} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models M\ N : \mathtt{Z} \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = T_2 \to \mathtt{Z}\}}\ \texttt{[c-app]}$$

$$\frac{\Gamma \models M : T_1 \mid \mathcal{R}_1 \quad \Gamma \models N : T_2 \mid \mathcal{R}_2 \quad \Gamma \models L : T_3 \mid \mathcal{R}_3}{\Gamma \models \mathtt{if}\ M\ N\ L : T_2 \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \{T_1 = \mathtt{Bool}, T_2 = T_3\}}\ \texttt{[c-if]}$$

$$\frac{\Gamma \models M : T \mid \mathcal{R} \quad \mathtt{Z} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models \mathtt{Fix}\ M : \mathtt{Z} \mid \mathcal{R} \cup \{T = \mathtt{Z} \to \mathtt{Z}\}}\ \texttt{[c-fix]}$$

En las reglas anteriores haría falta añadir una regla correspondiente con la construcción `[op[M]]`; sin embargo, no podemos dar una única regla que cubra de

manera adecuada cualquier operador primitivo, como veremos a continuación. Una definición desatinada de esta regla sería como la siguiente

$$\frac{\Gamma \models M_i : T_i \mid_{\mathcal{R}_i}}{\Gamma \models o^{(n)}\ [\ M_i\ ]_{i=1}^{n} : \Delta(o^{(n)}, [T_i]) \mid_{(\bigcup_{i=1}^{n} \mathcal{R}_i) \cup \Delta_{\mathcal{R}}(o^{(n)}, [T_i])}}\ \texttt{[c-op]}$$

El problema es que no podemos usar la función $\Delta(o^{(n)}, [T_i])$ (Def. 26) como lo hicimos en la Definición 71 (Apéndice A) para obtener el tipo que regresaría la aplicación del operador $o^{(n)}$. Esto se debe a que es muy posible que alguno de los T_i sea una variable de tipo, por lo que la función Δ no sabría qué regresar en ese caso.

Dado que no podemos dar una única regla para el caso `[op[M]]` daremos múltiples reglas, una para cada operador primitivo. Por ejemplo, para un conjunto conservador de operadores primitivos podríamos dar las siguientes reglas.

$$\frac{\Gamma \models M_1 : T_1 \mid_{\mathcal{R}_1} \quad \Gamma \models M_2 : T_2 \mid_{\mathcal{R}_2}}{\Gamma \models \texttt{+}\ [M_1\ M_2] : \texttt{Nat} \mid_{\mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = \texttt{Nat}, T_2 = \texttt{Nat}\}}}$$

$$\frac{\Gamma \models M_1 : T_1 \mid_{\mathcal{R}_1} \quad \Gamma \models M_2 : T_2 \mid_{\mathcal{R}_2}}{\Gamma \models \texttt{*}\ [M_1\ M_2] : \texttt{Nat} \mid_{\mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = \texttt{Nat}, T_2 = \texttt{Nat}\}}}$$

$$\frac{\Gamma \models M : T \mid_{\mathcal{R}}}{\Gamma \models \texttt{pred}\ [M] : \texttt{Nat} \mid_{\mathcal{R} \cup \{T = \texttt{Nat}\}}}$$

$$\frac{\Gamma \models M : T \mid_{\mathcal{R}}}{\Gamma \models \texttt{not}\ [M] : \texttt{Bool} \mid_{\mathcal{R} \cup \{T = \texttt{Bool}\}}}$$

$$\frac{\Gamma \models M : T \mid_{\mathcal{R}}}{\Gamma \models \texttt{isZero}\ [M] : \texttt{Bool} \mid_{\mathcal{R} \cup \{T = \texttt{Nat}\}}}$$

Retomemos el ejemplo mencionado al inicio de este capítulo y usemos las reglas que desarrollamos en esta sección para tipar la expresión $\lambda x.\lambda y.\lambda z.(x\ z)\ (x\ y)$ bajo un cierto conjunto de restricciones. Para mantener la continuidad con el ejemplo anterior consideraremos al contexto de tipado $\Gamma = \{x : \texttt{A}, y : \texttt{D}, z : \texttt{B}\}$.

Primero tipamos la aplicación $(x\ z)$

$$\frac{\dfrac{\Gamma(x) = \texttt{A}}{\Gamma \models x : \texttt{A} \mid_{\varnothing}}\ \texttt{[c-var]} \quad \dfrac{\Gamma(z) = \texttt{B}}{\Gamma \models z : \texttt{B} \mid_{\varnothing}}\ \texttt{[c-var]} \quad \texttt{C} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models (x\ z) : \texttt{C} \mid_{\{\texttt{A} = \texttt{B} \to \texttt{C}\}}}\ \texttt{[c-app]}$$

Y ahora tipamos la aplicación $(y\ z)$

$$\dfrac{\dfrac{\Gamma(y) = \mathtt{D}}{\Gamma \models y : \mathtt{D} \mid_{\varnothing}}\,\texttt{[c-var]} \quad \dfrac{\Gamma(z) = \mathtt{B}}{\Gamma \models z : \mathtt{B} \mid_{\varnothing}}\,\texttt{[c-var]} \quad \mathtt{E} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models (y\ z) : \mathtt{E} \mid_{\{\mathtt{D}=\mathtt{B}\to\mathtt{E}\}}}\,\texttt{[c-app]}$$

Combinando estos dos resultados tipamos la aplicación $(x\ z)\ (y\ z)$ y, finalmente, descargamos el contexto para tipar a la expresión $\lambda x.\lambda y.\lambda z.(x\ z)\ (x\ y)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \models (x\ z) : \mathtt{C} \mid_{\{\mathtt{A}=\mathtt{B}\to\mathtt{C}\}}} \quad \dfrac{\vdots}{\Gamma \models (y\ z) : \mathtt{E} \mid_{\{\mathtt{D}=\mathtt{B}\to\mathtt{E}\}}}\ \mathtt{F} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models (x\ z)\ (y\ z) : \mathtt{F} \mid_{\{\mathtt{A}=\mathtt{B}\to\mathtt{C},\ \mathtt{D}=\mathtt{B}\to\mathtt{E},\ \mathtt{C}=\mathtt{E}\to\mathtt{F}\}}}\,\texttt{[c-app]}}{\{x : \mathtt{A}, y : \mathtt{D}\} \models \lambda z.(x\ z)\ (y\ z) : \mathtt{B} \to \mathtt{F} \mid_{\{\mathtt{A}=\mathtt{B}\to\mathtt{C},\ \mathtt{D}=\mathtt{B}\to\mathtt{E},\ \mathtt{C}=\mathtt{E}\to\mathtt{F}\}}}\,\texttt{[c-λx]}}{\{x : \mathtt{A}\} \models \lambda y.\lambda z.(x\ z)\ (y\ z) : \mathtt{D} \to \mathtt{B} \to \mathtt{F} \mid_{\{\mathtt{A}=\mathtt{B}\to\mathtt{C},\ \mathtt{D}=\mathtt{B}\to\mathtt{E},\ \mathtt{C}=\mathtt{E}\to\mathtt{F}\}}}\,\texttt{[c-λx]}}{\models \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) : \mathtt{A} \to \mathtt{D} \to \mathtt{B} \to \mathtt{F} \mid_{\{\mathtt{A}=\mathtt{B}\to\mathtt{C},\ \mathtt{D}=\mathtt{B}\to\mathtt{E},\ \mathtt{C}=\mathtt{E}\to\mathtt{F}\}}}\,\texttt{[c-λx]}$$

Podemos observar que obtuvimos el mismo tipo y conjunto de restricciones que en el ejemplo informal al inicio del capítulo. Ahora solamente nos hace falta formalizar el proceso mediante el cual usamos las restricciones acumuladas para obtener un tipo válido para la expresión.

5.4. Unificación de restricciones

Al usar las reglas para la generación de restricciones obtenemos que una expresión M tiene tipo T bajo cierto conjunto de restricciones $\mathcal{R}$. Este tipo T no es válido en el sentido de que no podemos probar que $\vdash M : T$, pues nos hace falta modificar T de cierta manera que refleje las restricciones acumuladas en $\mathcal{R}$.

Lo que haremos a continuación será mostrar que un *unificador* del conjunto de restricciones nos permitirá transformar el tipo intermedio T en uno que sí tenga sentido bajo unas nuevas reglas de tipado similares a las de la Definición 71 (Apéndice A). Estas nuevas reglas son esencialmente idénticas a las de la Definición 71 (Apéndice A) siendo la única diferencia que en la regla `[a-λx]` no se considera a las anotaciones de tipos en la sintaxis del cálculo.

Cuando probamos[2] que $\vdash M : T$ usando las reglas con anotaciones de tipo lo hacemos mediante un proceso de *verificación de tipos*; se realiza entonces el análisis sintáctico de una expresión para verificar que las anotaciones de tipo sean congruentes entre ellas y que nos permita tipar la expresión completa. Al

[2]Decimos que un programa M está correctamente tipado si es posible asignarle un tipo bajo el contexto vacío, i.e. $\vdash M : T$

deshacernos de las anotaciones de tipo, ya no podemos hacer una verificación de tipos. En este caso el proceso mediante el cual probamos que $\vdash M : T$ es mediante *inferencia de tipos*; esto es, hacemos un análisis sintáctico de la expresión para inferir mediante el uso de las aplicaciones el tipo de la expresión. Al hacer inferencia de tipos daremos con el tipo más general de la expresión, el cual puede ser *instanciado* para obtener tipos más particulares.

Mostraremos ahora el cálculo ISWIM sin anotaciones de tipo, denotado Λ_a, y sus reglas de tipado correspondientes.

Definición 39: Λ_a ISWIM sin anotaciones de tipo [32]

$$
\begin{array}{rcll}
\Lambda_a & = & x & \texttt{[var]} \\
& | & \dot{c} & \texttt{[cte]} \\
& | & (\lambda x.\, M_1) & \texttt{[λx:T.M]} \\
& | & (M_1\ M_2) & \texttt{[app]} \\
& | & (\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ ) & \texttt{[op[M]]} \\
& | & (\texttt{if}\ \ M_1\ \ M_2\ \ M_3) & \texttt{[if]} \\
& | & (\texttt{Fix}\ \ M_1) & \texttt{[fix]}
\end{array}
$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad T \in \mathcal{T} \quad o^{(n)} \in \mathcal{O}_{\mathcal{P}} \quad M_i \in \Lambda_a$

Definición 40: Sistema de tipos simple sin anotaciones [32]

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \texttt{[a-var]}$$

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\Gamma \vdash \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c})}\ \texttt{[a-cte]}$$

$$\frac{\Gamma,\ x{:}T_1\ \vdash\ M{:}T_2}{\Gamma\ \vdash\ \lambda x.\, M\ :\ T_1 \to T_2}\ \texttt{[a-λx]}$$

$$\frac{\Gamma \vdash M{:}T_1 \to T_2 \qquad \Gamma \vdash N{:}T_1}{\Gamma \vdash M\ \ N : T_2}\ \texttt{[a-app]}$$

Definición 40: Sistema de tipos simple sin anotaciones [32] (cont.)

$$\frac{\Gamma \vdash M_i : T_i \quad 1 \leq i \leq n}{\Gamma \vdash o^{(n)}\ [\ M_i\ ]_{i=1}^{n} : \Delta(o^{(n)}, [T_i])}\ \texttt{[a-op]}$$

$$\frac{\Gamma \vdash M : \texttt{Bool} \quad \Gamma \vdash N : T \quad \Gamma \vdash L : T}{\Gamma \vdash \texttt{if}\ \ M\ \ N\ \ L : T}\ \texttt{[a-if]}$$

$$\frac{\Gamma \vdash M : T \to T}{\Gamma \vdash \texttt{Fix}\ \ M : T}\ \texttt{[a-fix]}$$

La clave que nos permitirá relacionar las expresiones $\Gamma \models M : T|_{\mathcal{R}}$ y $\Gamma \vdash M : T'$ es el concepto de *unificador* que definimos a continuación.

Definición 41: Unificadores [32]

Una sustitución de tipos σ *unifica* a la restricción de tipos $T = S$ si $\sigma(T) \equiv \sigma(S)$; esto es, al aplicar σ sobre ambos lados de la ecuación obtenemos exactamente el mismo resultado.

También diremos que σ es un *unificador* de $\mathcal{R}$ si σ unifica a todas las restricciones de $\mathcal{R}$.

El siguiente teorema muestra que si logramos encontrar un unificador para las restricciones generadas con el algoritmo de la Definición 76 (Apéndice B), entonces podremos obtener un tipo para la expresión que sea derivable usando las reglas de la Definición 75 (Apéndice B).

Teorema 5.4.1. *Si* $\Gamma \models M : T \mid_{\mathcal{R}}$ *y* μ *es un unificador del conjunto de restricciones* $\mathcal{R}$*, entonces* $\mu(\Gamma) \vdash M : \mu(T)$*.*

Demostración. Inducción sobre la relación $\Gamma \models M : T \mid_{\mathcal{R}}$.

Los casos bases corresponden a las reglas `[c-var]` y `[c-cte]`, para estos el conjunto de restricciones $\mathcal{R}$ es el conjunto vacío por lo que un unificador de ellos sería una sustitución vacía $\mu_{\varnothing}$. Tendríamos entonces que $\mu_{\varnothing}(\Gamma) \equiv \Gamma$ y $\mu_{\varnothing}(T) \equiv T$ por lo que en ambos casos obtendríamos que $\mu_{\varnothing}(\Gamma) \vdash M : \mu_{\varnothing}(T)$.

Tomemos como hipótesis de inducción que si $\Gamma \models M_i : T_i|_{\mathcal{R}_i}$ y μ_i es un unificador de $\mathcal{R}_i$, entonces $\mu_i(\Gamma) \vdash M_i : \mu_i(T_i)$.

- **Caso** `[c-λx]`. Si $\Gamma, x:\mathtt{Z} \models M:T \mid_{\mathcal{R}}$ donde $\mathtt{Z}$ es una variable nueva, entonces $\Gamma \models \lambda x.\, M:\mathtt{Z} \to T \mid_{\mathcal{R}}$. Debemos mostrar que $\mu(\Gamma) \vdash \lambda x.\, M:\mu(\mathtt{Z} \to T)$.

  Sea μ un unificador de $\mathcal{R}$, por hipótesis de inducción tenemos que

  $$\mu(\Gamma, x:\mathtt{Z}) \vdash M:\mu(T)$$

  Aplicando la sustitución sobre el contexto de tipado (Def. 35) tenemos que

  $$\mu(\Gamma), x:\mu(\mathtt{Z}) \vdash M:\mu(T)$$

  Por la regla `[a-λx]` podemos concluir que $\mu(\Gamma) \vdash \lambda x.\, M:\mu(\mathtt{Z}) \to \mu(T)$ y aplicando en sentido contrario la definición de sustitución de tipos (Def. 31) conluimos que

  $$\mu(\Gamma) \vdash \lambda x.\, M:\mu(\mathtt{Z} \to T)$$

- **Caso** `[c-app]` Si $\Gamma \models M_1 : T_1 \mid_{\mathcal{R}_1}$ y $\Gamma \models M_2 : T_2 \mid_{\mathcal{R}_2}$ y $\mathtt{Z}$ es una variable nueva, entonces $\Gamma \models M_1\ M_2 : \mathtt{Z} \mid_{\mathcal{R}}$ donde $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = T_2 \to \mathtt{Z}\}$. Debemos mostrar que $\mu(\Gamma) \vdash M_1\ M_2 : \mu(\mathtt{Z})$.

  Sea μ un unificador de $\mathcal{R}$. Observemos que μ también es unificador de $\mathcal{R}_1$ y $\mathcal{R}_2$, pues estos están contenidos en $\mathcal{R}$.

  Por hipótesis de inducción tenemos que

  $$\mu(\Gamma) \vdash M_1 : \mu(T_1)$$

  Como μ unifica a $\mathcal{R}$, entonces tenemos que $\mu(T_1) \equiv \mu(T_2 \to \mathtt{Z})$, por lo que de lo anterior obtenemos que

  $$\mu(\Gamma) \vdash M_1 : \mu(T_2 \to \mathtt{Z})$$

  Ahora aplicamos la sustitución de tipos para concluir que

  $$\mu(\Gamma) \vdash M_1 : \mu(T_2) \to \mu(\mathtt{Z})$$

  Por hipótesis de inducción también tenemos que

  $$\mu(\Gamma) \vdash M_2 : \mu(T_2)$$

  Finalmente, combinamos estos dos últimos resultados con la regla `[a-app]` para concluir que

  $$\mu(\Gamma) \vdash M_1\ M_2 : \mu(\mathtt{Z})$$

- **Caso** `[c-fix]` Si $\Gamma \models M:T \mid_{\mathcal{R}_1}$ y $\mathtt{Z}$ es una variable nueva, entonces tenemos que $\Gamma \models \mathtt{Fix}\ M : \mathtt{Z} \mid_{\mathcal{R}}$ donde $\mathcal{R} = \mathcal{R}_1 \cup \{T = \mathtt{Z} \to \mathtt{Z}\}$. Debemos mostrar que $\mu(\Gamma) \vdash \mathtt{Fix}\ M : \mu(\mathtt{Z})$.

  Sea μ un unificador de $\mathcal{R}$. Por hipótesis de inducción tenemos que

  $$\mu(\Gamma) \vdash M : \mu(T)$$

Como μ es unificador de $\mathcal{R}$, entonces $\mu(T) \equiv \mu(\mathtt{Z}) \to \mu(\mathtt{Z})$ por lo que de lo anterior obtenemos que

$$\mu(\Gamma) \vdash M : \mu(\mathtt{Z}) \to \mu(\mathtt{Z})$$

Finalmente aplicando la regla `[a-fix]` obtenemos lo que queríamos mostrar

$$\mu(\Gamma) \vdash \mathtt{Fix}\ M : \mu(\mathtt{Z})$$

- Los demás casos son muy similares a los anteriores y se omiten.

□

5.5. El unificador principal

En esta sección mostraremos un algoritmo que dado un conjunto $\mathcal{R}$ de restricciones, encuentre el unificador más general, al cual llamaremos el *unificador principal*.

Definición 42: μ Unificador principal [32]

Decimos que una sustitución μ es *más general* que otra sustitución σ, denotado $\sigma \sqsubseteq \mu$, si existe una sustitución σ', tal que $\sigma' \circ \mu = \sigma$.

Decimos que un unificador μ de $\mathcal{R}$ es un *unificador principal* si éste es el unificador más general posible. Esto es que para todo σ que también sea un unificador de $\mathcal{R}$ se cumple que $\sigma \sqsubseteq \mu$.

Usaremos la letra μ para denotar a una sustitución que sea un unificador principal.

A continuación damos el algoritmo $\mathcal{U}$ de Martelli y Montanari [25], el cual es una mejora al algoritmo original de Robinson [36, 35], que dado un conjunto de restricciones devuelve ya sea un unificador principal o que falla cuando sea imposible encontrarlo. Este algoritmo recibe una lista de restricciones la cual denotamos, al estilo de Haskell, como $(h : \mathcal{R})$ donde h es el elemento a la cabeza de la lista y $\mathcal{R}$ es el resto de ella. El algoritmo está definido por casos donde cada uno revisa si la lista corresponde con un patrón en particular. Cada patrón es revisado uno por uno de arriba a abajo; cuando uno no encaja con la lista o no se cumplen las condiciones adicionales, entonces se prueba con el siguiente patrón. Si ninguno de los patrones encaja con la lista, entonces se devuelve un

error que corresponde con el hecho de que no es posible encontrar un unificador para el conjunto dado. Representamos a la sustitución vacía mediante [ ] y $\mathcal{V}_{\mathcal{T}_\mathcal{L}}$ es la función que devuelve las variables de tipo que figuren en un tipo T.

Definición 43: $\mathcal{U}$ **Unificación [25]**

$$
\begin{aligned}
&\mathcal{U}([\ ]) && = [\ ] \\
&\mathcal{U}(\ (S = T) : \mathcal{R}\ ) && = \mathcal{U}(\mathcal{R}) \quad \text{si } S \equiv T \\
&\mathcal{U}(\ (S_1 \to S_2 = T_1 \to T_2) : \mathcal{R}\ ) && = \mathcal{U}(\ (S_1 = T_1, S_2 = T_2) : \mathcal{R}\ ) \\
&\mathcal{U}(\ (\mathtt{A} = T) : \mathcal{R}\ ) && = \mathcal{U}(\ [\mathtt{A} \mapsto T]\mathcal{R}\ ) \circ [\mathtt{A} \mapsto T] \\
& && \quad \text{si } \mathtt{A} \notin \mathcal{V}_{\mathcal{T}_\mathcal{L}}(T) \\
&\mathcal{U}(\ (T = \mathtt{A}) : \mathcal{R}\ ) && = \mathcal{U}(\ (\mathtt{A} = T) : \mathcal{R}\ )
\end{aligned}
$$

donde $S, T \in \mathcal{T}$, $\mathtt{A} \in \mathcal{T}_\mathcal{V}$, y $\mathcal{R}$ es un conjunto de restricciones.

Teorema 5.5.1 (Unificador principal [25])**.** *El algoritmo de unificación* $\mathcal{U}$ *cumple con las siguientes propiedades.*

- $\mathcal{U}(\mathcal{R})$ *siempre termina ya sea devolviendo una sustitución o porque falló devolviendo un error.*
- *Si* $\mathcal{U}(\mathcal{R}) = \mu$*, entonces* μ *es un unificador principal para* $\mathcal{R}$*.*

Demostración. La prueba puede ser consultada en el trabajo de Martelli y Montanari [25]. □

Concluimos este capítulo mostrando que para encontrar el *tipo principal* de una expresión basta combinar el algoritmo de generación de restricciones (Def. 76) y el algoritmo para calcular el unificador principal (Def. 42).

Definición 44: Tipo principal [32]

Decimos que T es el tipo principal de M si $\Gamma \vdash M : T$ y para todo tipo T' tal que $\Gamma' \vdash M : T'$ existe una sustitución de tipos σ donde $T' = \sigma(T)$ y $\Gamma' = \sigma(\Gamma)$.

Teorema 5.5.2. *Si* $\Gamma \models M : T|_\mathcal{R}$ *y* μ *es el unificador principal de* $\mathcal{R}$*, entonces se tiene que* $\mu(\Gamma) \vdash M : \mu(T)$*, donde* $\mu(T)$ *es el tipo principal de* M*.*

Demostración. La prueba queda fuera del alcance de este trabajo y se omite. Ésta puede ser consultada en [13]. □

Finalmente, retomamos el ejemplo anterior donde concluimos que

$$\models \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) : \mathtt{A} \to \mathtt{D} \to \mathtt{B} \to \mathtt{F}|_{\{\mathtt{A}=\mathtt{B}\to\mathtt{C},\mathtt{D}=\mathtt{B}\to\mathtt{E},\mathtt{C}=\mathtt{E}\to\mathtt{F}\}}$$

Usaremos el algoritmo $\mathcal{U}$ para encontrar el unificador principal μ para el conjunto de restricciones y así dar con el tipo principal de la expresión.

$$\begin{aligned}
\mu &= \mathcal{U}((\mathtt{A} = \mathtt{B} \to \mathtt{C}) : \{\mathtt{D} = \mathtt{B} \to \mathtt{E}, \mathtt{C} = \mathtt{E} \to \mathtt{F}\}) \\
&= \mathcal{U}([\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}]\{\mathtt{D} = \mathtt{B} \to \mathtt{E}, \mathtt{C} = \mathtt{E} \to \mathtt{F}\}) \circ [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}] \\
&= \mathcal{U}((\mathtt{D} = \mathtt{B} \to \mathtt{E}) : \{\mathtt{C} = \mathtt{E} \to \mathtt{F}\}) \circ [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}] \\
&= \mathcal{U}([\mathtt{D} \mapsto \mathtt{B} \to \mathtt{E}]\{\mathtt{C} = \mathtt{E} \to \mathtt{F}\}) \circ [\mathtt{D} \mapsto \mathtt{B} \to \mathtt{E}] \circ [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}] \\
&= \mathcal{U}((\mathtt{C} = \mathtt{E} \to \mathtt{F}) : []) \circ [\mathtt{D} \mapsto \mathtt{B} \to \mathtt{E}] \circ [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}] \\
&= [\mathtt{C} \mapsto \mathtt{E} \to \mathtt{F}] \circ [\mathtt{D} \mapsto \mathtt{B} \to \mathtt{E}] \circ [\mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}] \\
&= [\mathtt{C} \mapsto \mathtt{E} \to \mathtt{F}] \circ [\mathtt{D} \mapsto \mathtt{B} \to \mathtt{E},\ \ \mathtt{A} \mapsto \mathtt{B} \to \mathtt{C}] \\
&= [\mathtt{C} \mapsto \mathtt{E} \to \mathtt{F},\ \ \mathtt{D} \mapsto \mathtt{B} \to \mathtt{E},\ \ \mathtt{A} \mapsto \mathtt{B} \to \mathtt{E} \to \mathtt{F}]
\end{aligned}$$

Por lo tanto tenemos que $\vdash \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) : \mu(\mathtt{A} \to \mathtt{D} \to \mathtt{B} \to \mathtt{F})$, es decir

$$\vdash \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) : (\mathtt{B} \to \mathtt{E} \to \mathtt{F}) \to (\mathtt{B} \to \mathtt{E}) \to \mathtt{B} \to \mathtt{F}$$

Finalmente aplicamos un renombre de las variables con fines cosméticos y obtenemos el resultado que esperábamos

$$\vdash \lambda x.\lambda y.\lambda z.(x\ z)\ (y\ z) : (\mathtt{A} \to \mathtt{B} \to \mathtt{C}) \to (\mathtt{A} \to \mathtt{B}) \to \mathtt{A} \to \mathtt{C}$$

En este capítulo hemos mostrado cómo mantener la seguridad del sistema de tipos sin obligar al programador a incluir anotaciones de tipo en cada abstracción-λ. En el próximo capítulo mostraremos cómo modificar el sistema de tipos para incluir polimorfismo paramétrico en el lenguaje.

Capítulo 6

Iswim polimórfico

Al inicio del capítulo anterior mencionamos que nuestro problema principal era que no podíamos usar una misma función identidad que fuera capaz de lidiar con cualquier tipo de dato sobre el cual la quisiéramos aplicar, y que necesitábamos definir una cantidad infinita de funciones esencialmente idénticas salvo por sus anotaciones de tipo para solventar esto. Cuando un lenguaje de programación permite definir funciones que operen de la misma manera sobre una variedad de tipos se dice que cuenta con *polimorfismo paramétrico* [32]. Algunos lenguajes de programación implementan un tipo de polimorfismo más débil que el paramétrico mediante la definición de varias instancias de una misma función donde cada una recibe argumentos de distintos tipos; a esta clase de polimorfismo se le conoce como *ad hoc* y es análoga a nuesta solución del capítulo anterior. Por ejemplo, el lenguaje de programación Java usó polimorfismo *ad hoc* a partir del año 2004 [30], cuando se introdujeron los tipo de datos génericos a consecuencia del trabajo de Wadler y otros [29, 6]; y aún así se mostró en 2016 que su sistema de tipos no garantiza la seguridad de tipos en tiempo de compilación [3].

En el capítulo anterior logramos deshacernos de las anotaciones de tipos para conseguir tipar a $\lambda x.x$ con el tipo $\mathtt{A} \to \mathtt{A}$, lo cual aparentemente significa que podemos usar la misma función sobre cualquier tipo de dato que queramos y obtener el resultado que naturalmente esperaríamos. Veremos que esto todavía no es cierto.

6.1. Expresiones `let`

En esta sección definiremos el enunciado `let` $x = N$ `in` M usando azúcar sintáctica de la siguiente manera

$$\texttt{let}\ x = N\ \texttt{in}\ M \stackrel{\text{def}}{=} (\lambda x.\, M)\ N$$

Con este nuevo enunciado podemos asignarle un nombre a una pieza de código a la cual nos podemos referir e invocar en cualquier punto subsecuente del programa mediante el nombre que hayamos elegido. De esta manera podemos considerar un programa de ISWIM como una serie de `let`s anidados donde cada uno define alguna función o constante que podemos utilizar en el cuerpo de la sentencia principal M refiriéndonos a ellas mediante la variable que le hayamos asignado.

$$\begin{array}{l} \texttt{let}\ f_1 = N_1 \quad \texttt{in} \\ \quad \texttt{let}\ f_2 = N_2 \quad \texttt{in} \\ \qquad \vdots \\ \qquad \texttt{let}\ f_n = N_n\ \texttt{in}\ M \end{array}$$

En el siguiente programa usamos `let` para asignarle el nombre id a la función $\lambda x.x$, la cual quisiéramos poder aplicar indistintamente sobre valores de tipo `Nat` y `Bool`. Observemos que si en efecto pudiéramos hacer esto entonces el programa estaría correctamente tipado ya que la condición del `if` sería de tipo `Bool` y ambos posibles casos serían de tipo `Nat`, por lo que el tipo de la expresión sería `Nat`.

$$\texttt{let}\ \ id = \lambda x.x\ \ \texttt{in}\ \ \texttt{if}\ (id\ \texttt{true})\ (id\ 1)\ 2$$

La versión desazucarada de este programa se ve de la siguiente manera.

$$(\lambda id.\ \texttt{if}\ (id\ \texttt{true})\ (id\ 1)\ 2)\ \ \lambda x.x$$

Intentaremos tipar esta expresión usando la inferencia de tipos que desarrollamos en la sección anterior. Para esto primero generaremos las restricciones de tipo y después intentaremos unificarlas para dar con su tipo principal. Consideremos $\Gamma = \{id{:}\texttt{A}\}$ y generemos las restricciones para la condición $(id\ \texttt{true})$ y el caso verdadero $(id\ 1)$ del `if`.

$$\dfrac{\dfrac{}{\Gamma \vdash id : \mathtt{A} \mid_{\varnothing}}\,\texttt{[c-var]} \quad \dfrac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool} \mid_{\varnothing}}\,\texttt{[c-cte]}}{\Gamma \vdash id\ \mathtt{true} : \mathtt{B} \mid_{\mathtt{A}=\mathtt{Bool}\to\mathtt{B}}}\,\texttt{[c-app]}$$

$$\dfrac{\dfrac{}{\Gamma \vdash id : \mathtt{A} \mid_{\varnothing}}\,\texttt{[c-var]} \quad \dfrac{}{\Gamma \vdash 1 : \mathtt{Nat} \mid_{\varnothing}}\,\texttt{[c-cte]}}{\Gamma \vdash id\ 1 : \mathtt{C} \mid_{\mathtt{A}=\mathtt{Nat}\to\mathtt{C}}}\,\texttt{[c-app]}$$

Para el caso falso tendríamos que $\Gamma \vdash 2 : \mathtt{Nat} \mid_{\varnothing}$ por la regla `[c-cte]`. Ahora tipamos la expresión `if`

$$\dfrac{\Gamma \vdash id\ \mathtt{true} : \mathtt{B} \mid_{\mathtt{A}=\mathtt{Bool}\to\mathtt{B}} \quad \Gamma \vdash id\ 1 : \mathtt{C} \mid_{\mathtt{A}=\mathtt{Nat}\to\mathtt{C}} \quad \Gamma \vdash 2 : \mathtt{Nat} \mid_{\varnothing}}{\Gamma \vdash \mathtt{if}\ (id\ \mathtt{true})\ (id\ 1)\ 2 : \mathtt{C} \mid_{\mathcal{R}}}\,\texttt{[c-if]}$$

Donde $\mathcal{R}$ contiene las siguientes restricciones de tipo

- $\mathtt{A} = \mathtt{Bool} \to \mathtt{B}$
- $\mathtt{A} = \mathtt{Nat} \to \mathtt{C}$
- $\mathtt{Bool} = \mathtt{Bool}$
- $\mathtt{C} = \mathtt{Nat}$

Para terminar la derivación consideremos que $\Gamma' = \{x : \mathtt{D}\}$

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{if}\ (id\ \mathtt{true})\ (id\ 1)\ 2 : \mathtt{C} \mid_{\mathcal{R}}}{\vdash \lambda id.\ \mathtt{if}\ (id\ \mathtt{true})\ (id\ 1)\ 2 : \mathtt{A} \to \mathtt{C} \mid_{\mathcal{R}}}\,\texttt{[c-}\lambda\texttt{x]} \quad \dfrac{\Gamma' \vdash x : \mathtt{D} \mid_{\varnothing}}{\vdash \lambda x.x : \mathtt{D} \to \mathtt{D} \mid_{\varnothing}}\,\texttt{[c-}\lambda\texttt{x]}}{\vdash (\lambda id.\ \mathtt{if}\ (id\ \mathtt{true})\ (id\ 1)\ 2)\ \ \lambda x.x : \mathtt{E} \mid_{\mathcal{R}\cup\{\mathtt{A}\to\mathtt{C}=(\mathtt{D}\to\mathtt{D})\to\mathtt{E}\}}}\,\texttt{[c-app]}$$

Ahora solamente nos hace falta encontrar un unificador μ para el conjunto de restricciones que obtuvimos y en caso de lograrlo reportaríamos que el tipo del programa es $\mu(\mathtt{E})$.

- $\mathtt{A} = \mathtt{Bool} \to \mathtt{B}$
- $\mathtt{A} = \mathtt{Nat} \to \mathtt{C}$
- $\mathtt{Bool} = \mathtt{Bool}$
- $\mathtt{C} = \mathtt{Nat}$

- $\texttt{A} \to \texttt{C} = (\texttt{D} \to \texttt{D}) \to \texttt{E}$

Si de verdad pudiéramos unificar estas restricciones, entonces de las primeras dos ecuaciones obtendríamos que $\texttt{Bool} \to \texttt{B} = \texttt{Nat} \to \texttt{C}$, lo cual implicaría que $\texttt{B} = \texttt{C}$ y $\texttt{Bool} = \texttt{Nat}$; sin embargo, no es posible satisfacer esta última restricción bajo ninguna sustitución, por lo que no es posible unificar el conjunto de restricciones y por lo tanto no podemos tipar este programa.

6.2. Polimorfismo mediante sustitución

El problema viene de que supusimos que *id* es una función de tipo `A` y usamos esta misma función sobre dos tipos que son incompatibles entre ellos. Lo que necesitaríamos sería una manera de asignarle una variable de tipo distinta a $\lambda x.x$ en cada una de las aplicaciones. Esto lo podemos solucionar muy fácilmente si antes de tipar un enunciado `let` primero hacemos un paso de β-reducción, y sólo entonces intentamos tipar la expresión reducida en lugar de la expresión original. O sea que en lugar de intentar tipar $(\lambda id.\ \texttt{if}\ (id\ \texttt{true})\ (id\ 1)\ 2)\ \ \lambda x.x$, esto lo reduciríamos a $\texttt{if}\ ((\lambda x.x)\ \texttt{true})\ ((\lambda x.x)\ 1)\ 2$ y tiparíamos esta expresión. Ahora que cada instancia de *id* ha sido sustituida por su definición, hemos roto la conexión entre cada una de estas funciones permitiendo que sean tipadas de manera independiente.

El problema de esto es que tendríamos que mezclar ejecución con la verificación de tipos, pero nos resultaría imposible distinguir entre un redex correspondiente a un enunciado `let` y uno correspondiente a una aplicación regular. Para poder hacer esta distinción tendremos que agregar la construcción `let` como elemento primitivo del cálculo

$$\texttt{let}\ \ x = N\ \ \texttt{in}\ \ M$$

y así podríamos dar una regla de tipado apropiada.

$$\frac{\Gamma \vdash M[x \leftarrow N] : T_1 \mid_{\mathcal{R}_1} \qquad \Gamma \vdash N : T_2 \mid_{\mathcal{R}_2} \qquad \mathcal{R}_2 \text{ es unificable}}{\Gamma \vdash \texttt{let}\ \ x = N\ \ \texttt{in}\ \ M : T_1 \mid_{\mathcal{R}_1}}$$

La primera condición dice que si al sustituir x por N en M, obtenemos que su tipo es T_1 bajo las restricciones $\mathcal{R}_1$, entonces ése es el tipo y las restricciones correspondientes a la construcción $\texttt{let}\ \ x = N\ \ \texttt{in}\ \ M$. Las siguientes dos condiciones sirven para asegurarnos de que N puede ser tipado correctamente con un tipo principal. Si omitiéramos estas últimas dos condiciones entonces seríamos capaces de tipar correctamente expresiones en las cuales x no figure como variable libre en M y N no tenga ningún sentido, por ejemplo $\texttt{let}\ \ x = 42\ \lambda x.x\ \ \texttt{in true}$

A pesar de que este enfoque funciona correctamente y permite definir funciones polimórficas, en la práctica puede llegar a ser muy ineficiente debido a que deberemos verificar el tipo de N por cada ocurrencia de x en M. En la siguiente sección mostraremos cómo verificar el tipo de N una sola vez independientemente de cuántas veces sea usado en el cuerpo de M.

6.3. Polimorfismo mediante generalización de variables

En esta sección desarrollaremos un método más elegante y eficiente para implementar el polimorfismo paramétrico en el sistema de tipos. Para esto primero recordemos que el enunciado $\texttt{let}\ \ x = N\ \ \texttt{in}\ \ M$ ya es una construcción primitiva de nuestro cálculo y retomemos el ejemplo anterior

$$\texttt{let}\ \ id = \lambda x.x\ \ \texttt{in}\ \ \texttt{if}\ (id\ \texttt{true})\ (id\ 1)\ 2$$

Para tipar $\texttt{let}\ \ x = N\ \ \texttt{in}\ \ M$ primero deberemos obtener el *tipo principal* de N; en este caso $\lambda x.x$, para el cual sabemos que su tipo principal es $\texttt{A} \to \texttt{A}$. Una vez que tenemos el tipo principal de N lo *generalizaremos* a un *esquema de tipos*.

Definición 45: $\mathcal{E}$ Esquema de tipos [32]

Un *esquema de tipos*, denotado mediante $\mathcal{E}$, es una expresión la forma

$$\forall \texttt{A}_1 \ldots \texttt{A}_n.\, T$$

donde $\texttt{A}_i \in \mathcal{T}_\mathcal{V}$ $\quad T \in \mathcal{T}$ y cada $\texttt{A}_i \in \mathcal{V}_\mathcal{T}(T)$. Además todas las $\texttt{A}_i$ deberán ser distintas entre ellas, y es posible que $n = 0$, por lo que $\forall.T$ sería también un esquema de tipos válido.

Por ejemplo, sabemos que el tipo principal de la función identidad $\lambda x.x$ es $\texttt{A} \to \texttt{A}$. En este caso podemos obtener su esquema de tipos al cuantificar universalmente todas las variables de tipo contenidas en el tipo principal, por lo que obtendríamos que $\forall \texttt{A}.\texttt{A} \to \texttt{A}$ es el esquema de tipos asociado a $\lambda x.x$.

La idea de generalizar un tipo a un esquema de tipos es que reemplacemos nuestros contextos de tipado Γ con nuevos *contextos de tipado generalizados* $\overline{\Gamma}$, donde en lugar de que cada variable x_i esté anotada con un tipo, éstas estén anotadas con un esquema de tipos.

Definición 46: $\bar{\Gamma}$ Contextos de tipado generalizados [32]

Un *contexto de tipado generalizado* es una secuencia de variables a las cuales se les asocia un esquema de tipos.

$$\bar{\Gamma} = \{x_1 : \mathcal{E}_1, \ldots, x_n : \mathcal{E}_n\}$$

donde $x_i \in \mathbb{X}$, $\mathcal{E}_i$ es un esquema de tipos, y $x_i \neq x_j$ siempre que $i \neq j$.

Si tuviéramos un contexto de tipado simple como $\Gamma = \{x : \texttt{Nat},\ f : \texttt{A} \to \texttt{B}\}$, su generalización podría ser de la forma $\bar{\Gamma} = \{x : \forall.\texttt{Nat},\ f : \forall\, \texttt{B.A} \to \texttt{B}\}$. Obsérvese que el tipo `Nat` no contiene variables de tipo, por lo cual no hay ninguna variable de tipo cuantificada en su esquema de tipos correspondiente. De manera similar observemos que a pesar de que el tipo $\texttt{A} \to \texttt{B}$ tiene dos variables de tipo, es posible que en su esquema de tipos correspondiente solamente una de ellas (o ninguna) esté cuantificada.

Con la introducción de los esquemas de tipo y los contextos de tipado generalizados necesitaremos de una función $\mathcal{V}_{\mathcal{T}_\mathcal{L}}$ para cada una de estas estructuras que nos devuelva el conjunto de variables de tipo libres, es decir aquellas que no estén cuantificadas con un $\forall$.

Definición 47: $\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\mathcal{E})$ Variables de tipo libres para $\mathcal{E}$ [32]

Definimos la función $\mathcal{V}_{\mathcal{T}_\mathcal{L}}$ que toma un esquema de tipos $\mathcal{E}$ y devuelve el conjunto de variables de tipo libres en $\mathcal{E}$

$$\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\forall \texttt{A}_1 \ldots \texttt{A}_n.T) = \mathcal{V}_\tau(T) \setminus \{\texttt{A}_1 \ldots \texttt{A}_n\}$$

Definición 48: $\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$ Variables de tipo libres para $\bar{\Gamma}$ [32]

Definimos la función $\mathcal{V}_{\mathcal{T}_\mathcal{L}}$ que toma un contexto de tipado generalizado $\bar{\Gamma}$ y devuelve el conjunto de variables de tipo libres en $\bar{\Gamma}$

$$\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\{x_1 : \mathcal{E}_1 \ldots x_n : \mathcal{E}_n\}) = \bigcup_{i=1}^{n} \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\mathcal{E})$$

Por ejemplo, retomando el ejemplo anterior, calcularemos el conjunto $\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$

$$\begin{aligned}
\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma}) &= \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\{x{:}\forall.\mathtt{Nat},\ f{:}\forall\,\mathtt{B}.\mathtt{A}\to\mathtt{B}\}) \\
&= \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\forall.\mathtt{Nat}) \cup \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\forall\,\mathtt{B}.\mathtt{A}\to\mathtt{B}) \\
&= (\mathcal{V}_{\mathcal{T}}(\mathtt{Nat}) \setminus \varnothing) \cup (\mathcal{V}_{\mathcal{T}}(\mathtt{A}\to\mathtt{B}) \setminus \{\mathtt{B}\}) \\
&= (\varnothing \setminus \varnothing) \cup (\{\mathtt{A},\mathtt{B}\} \setminus \{\mathtt{B}\}) \\
&= \varnothing \cup \{\mathtt{A}\} \\
&= \{\mathtt{A}\}
\end{aligned}$$

Además deberemos de definir la aplicación de una sustitución de tipos σ sobre estas dos estructuras.

Definición 49: $\sigma(\mathcal{E})$ Sustitución sobre $\mathcal{E}$ [32]

Una sustitución de tipos σ también puede ser aplicada sobre un esquema de tipos $\mathcal{E}$ de la siguiente manera

$$\sigma(\forall \mathtt{A}_1 \ldots \mathtt{A}_n.T) = \forall \mathtt{A}_1 \ldots \mathtt{A}_n.\sigma_{\neg\vec{\mathtt{A}}}(T)$$

donde $\sigma_{\neg\vec{\mathtt{A}}}$ es la sustitución de tipos que resulta de quitarle a σ las asociaciones $\mathtt{A}_i \mapsto T_i$ para $1 \leq i \leq n$

$$\sigma_{\neg\vec{\mathtt{A}}} \stackrel{\text{def}}{=} \sigma \setminus \{\mathtt{A}_1 \mapsto T_1, \ldots, \mathtt{A}_n \mapsto T_n\}$$

Alternativamente también usaremos la notación $\sigma_{\neg\mathtt{ABC}}$ para denotar a la sustitución $\sigma \setminus \{\mathtt{A} \mapsto T_1, \mathtt{B} \mapsto T_2, \mathtt{C} \mapsto T_3\}$

Definición 50: $\sigma(\bar{\Gamma})$ Sustitución sobre $\bar{\Gamma}$ [32]

Una sustitución de tipos σ también puede ser aplicada sobre un contexto de tipado generalizado $\bar{\Gamma}$ de la siguiente manera

$$\sigma(\{x_1{:}\mathcal{E}_1, x_2{:}\mathcal{E}_2, \ldots, x_n{:}\mathcal{E}_n\}) = \{x_1{:}\sigma(\mathcal{E}_1), x_2{:}\sigma(\mathcal{E}_2), \ldots, x_n{:}\sigma(\mathcal{E}_n)\}$$

Siguiendo con los ejemplos anteriores apliquemos $\sigma(\bar{\Gamma})$ donde

$$\sigma = \{\mathtt{A} \mapsto \mathtt{Nat}\to\mathtt{Nat},\ \mathtt{B} \mapsto \mathtt{C}\to\mathtt{C},\ \mathtt{C} \mapsto \mathtt{Bool}\}$$

$$\begin{aligned}\sigma(\bar{\Gamma}) &= \sigma(\{x{:}\forall.\mathtt{Nat},\ f{:}\forall\,\mathtt{B}.\mathtt{A}\to\mathtt{B}\})\\ &= \{x{:}\sigma(\forall.\mathtt{Nat}),\ f{:}\sigma(\forall\,\mathtt{B}.\mathtt{A}\to\mathtt{B})\}\\ &= \{x{:}\forall.\sigma(\mathtt{Nat}),\ f{:}\forall\,\mathtt{B}.\sigma_{\neg\mathtt{B}}(\mathtt{A}\to\mathtt{B})\}\\ &= \{x{:}\forall.\mathtt{Nat},\ f{:}\forall\,\mathtt{B}.\sigma_{\neg\mathtt{B}}(\mathtt{A})\to\sigma_{\neg\mathtt{B}}(\mathtt{B})\}\\ &= \{x{:}\forall.\mathtt{Nat},\ f{:}\forall\,\mathtt{B}.(\mathtt{Nat}\to\mathtt{Nat})\to\mathtt{B}\}\end{aligned}$$

La notación $\sigma_{\neg\vec{\mathtt{A}}}$ y $\sigma_{\neg\mathtt{BC}}$ será importante más adelante para algunas demostraciones.

Ahora definiremos la *generalización de variables de tipo*, el cual es el concepto clave que nos permitirá introducir el polimorfismo paramétrico en el lenguaje.

Definición 51: $\mathtt{gen}(\bar{\Gamma},T)$ Generalización de variables [32]

Sea $\bar{\Gamma}$ un contexto de tipado generalizado y T un tipo. La *generalización* de T con respecto a $\bar{\Gamma}$ es un esquema de tipos en el cual cada variable de tipo en T, exceptuando a las variables que aparecen libres en $\bar{\Gamma}$, es sustituida por una variable fresca que queda cuantificada en el esquema de tipos; y se define de la siguiente manera

$$\begin{aligned}\mathtt{gen}(\bar{\Gamma},T) &\stackrel{\text{def}}{=} \forall\vec{\mathtt{Z}}.[\vec{\mathtt{A}}\mapsto\vec{\mathtt{Z}}](T)\\ &\stackrel{\text{def}}{=} \forall\mathtt{Z}_1\ldots\mathtt{Z}_n.[\mathtt{A}_1\mapsto\mathtt{Z}_1,\ldots,\mathtt{A}_n\mapsto\mathtt{Z}_n](T)\end{aligned}$$

Donde $\{\mathtt{A}_1,\ldots,\mathtt{A}_n\}=\mathcal{V}_\tau(T)\setminus\mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$ son las variables de tipo que figuran en T menos las variables de tipo libres en $\bar{\Gamma}$; y $\{\mathtt{Z}_1,\ldots,\mathtt{Z}_n\}\subset\mathbb{N}_\mathbb{V}$ es un conjunto de n variables frescas.

Con estas definiciones podemos volver al ejemplo

$$\mathtt{let}\ \ id=\lambda x.x\ \ \mathtt{in}\ \ \mathtt{if}\ (id\,\mathtt{true})\ (id\ 1)\ 2$$

que para tiparlo generalizaremos el tipo principal de $\lambda x.x$ al esquema de tipos $\forall\mathtt{A}.\mathtt{A}\to\mathtt{A}$ y agregaremos la asociación $id:\forall\mathtt{A}.\mathtt{A}\to\mathtt{A}$ al contexto de tipado generalizado $\bar{\Gamma}$. De esta manera cuando queramos tipar $(id\,\mathtt{true})$ o $(id\ 1)$, buscaremos el esquema de tipos asociado al identificador id en $\bar{\Gamma}$ y lo *instanciaremos* a un tipo T según la siguiente definición.

Definición 52: $\mathtt{finst}(\mathcal{E})$ Instanciación de variables frescas [32]

Si $\mathcal{E}$ es un esquema de tipos, definimos la operación $\mathtt{finst}(\mathcal{E})$ de la siguiente manera:

$$\mathtt{finst}(\forall\mathtt{A}_1\ldots\mathtt{A}_n.\,T)=\sigma(T)$$

Definición 52: $\texttt{finst}(\mathcal{E})$ Instanciación de variables frescas [32] (cont.)

donde σ es una sustitución $[\mathtt{A}_1 \mapsto \mathtt{Z}_1 \ldots \mathtt{A}_n \mapsto \mathtt{Z}_n]$ y donde cada $\mathtt{Z}_i \in \mathbb{N}_\mathbb{V}$ es una variable de tipo nueva. Si $n = 0$, entonces aplicaremos la sustitución vacía sobre T y devolveremos T.

De esta manera al tipar $(id\ \texttt{true})$ instanciaríamos el esquema $\forall\mathtt{A}.\mathtt{A} \to \mathtt{A}$ al tipo $\mathtt{B} \to \mathtt{B}$ y al instanciar el mismo esquema al intentar tipar $(id\ 1)$ obtendríamos que id es de tipo $\mathtt{C} \to \mathtt{C}$. Con esto hemos logrado romper la conexión entre ambos usos de la función id permitiendo así que pueda ser usada sobre tipos distintos.

Ahora deberemos de adaptar las reglas para la generación de restricciones que dimos en la Definición 76 para que funcionen polimórficamente mediante la generalización de los contextos de tipado que desarrollamos en esta sección; por lo que la principal diferencia entre estas definiciones es que donde aparecía un contexto de tipado Γ, ahora aparecerá un contexto de tipado generalizado $\bar{\Gamma}$. A continuación señalaremos las diferencias menos sutiles.

Al tipar una variable x ahora necesitaremos instanciar el esquema de tipos asociado a la variable por lo que cambiaremos la regla

$$\frac{\Gamma(x) = T}{\Gamma \models x{:}T \mid \varnothing}\ \texttt{[c-var]}$$

por la regla

$$\frac{\bar{\Gamma}(x) = \mathcal{E}}{\bar{\Gamma} \models x : \texttt{finst}(\mathcal{E}) \mid \varnothing}\ \texttt{[p-var]}$$

Para las abstracciones $\lambda x.M$ necesitaremos generar $\mathtt{Z}$, una variable de tipo nueva, que denote al tipo del argumento x. Antes extendíamos el contexto $\Gamma, x : \mathtt{Z}$ con la nueva variable pero ahora lo extenderemos el contexto $\bar{\Gamma}, x : \forall.\mathtt{Z}$ con un esquema de tipos en el cual $\mathtt{Z}$ se mantiene libre. Cambiaremos entonces la regla

$$\frac{\mathtt{Z} \in \mathbb{N}_\mathbb{V} \quad \Gamma, x{:}\mathtt{Z} \models M{:}T \mid \mathcal{R}}{\Gamma \models \lambda x.M : \mathtt{Z} \to T \mid \mathcal{R}}\ \texttt{[c-}\lambda\texttt{x]}$$

por la regla

$$\frac{\mathtt{Z} \in \mathbb{N}_\mathbb{V} \quad \bar{\Gamma}, x{:}\forall.\mathtt{Z} \models M{:}T \mid \mathcal{R}}{\bar{\Gamma} \models \lambda x.M : \mathtt{Z} \to T \mid \mathcal{R}}\ \texttt{[p-}\lambda\texttt{x]}$$

Finalmente presentaremos la regla para la nueva sentencia $\texttt{let}\ x = N\ \texttt{in}\ M$

$$\bar{\Gamma}' \models N{:}T_1' \mid \mathcal{R}_1$$

$$\frac{\begin{array}{c}\mu = \mathcal{U}(\mathcal{R}_1) \\ \bar{\Gamma} = \mu(\bar{\Gamma}') \\ T_1 = \mu(T_1') \\ \bar{\Gamma}, x{:}\mathtt{gen}(\bar{\Gamma}, T_1) \models M : T_2 \mid_{\mathcal{R}_2}\end{array}}{\bar{\Gamma} \models \mathtt{let}\ \ x = N\ \ \mathtt{in}\ \ M : T_2 \mid_{\mathcal{R}_2}}\ \texttt{[p-let]}$$

Lo que esta regla dice es que primero deberemos calcular un tipo T' y un conjunto de restricciones $\mathcal{R}_1$ a partir de algún contexto $\bar{\Gamma}'$ para la expresión N. Luego obtendremos el unificador μ de $\mathcal{R}_1$ para con él conseguir el tipo principal $T = \mu(T')$ de N. También usaremos el mismo unificador μ para obtener un nuevo contexto de tipado $\bar{\Gamma} = \mu(\bar{\Gamma}')$ el cual usaremos para el resto de la derivación. Finalmente extenderemos este nuevo contexto $\bar{\Gamma}, x{:}\mathtt{gen}(\bar{\Gamma}, T)$ con la generalización del tipo principal de N para obtener un tipo T_2 y restricciones $\mathcal{R}_2$ para la expresión M, con lo cual podemos concluir que $\bar{\Gamma} \vdash \mathtt{let}\ \ x = N\ \ \mathtt{in}\ \ M : T_2 \mid_{\mathcal{R}_2}$.

Con esto concluimos el desarrollo de ISWIM polimórfico. En el Apéndice C se podrán encontrar las definiciones pertinentes al cálculo Λ_p, así como las reglas de tipado y generación de restricciones polimórficas. A continuación probaremos los resultados relevantes a la seguridad del lenguaje.

6.4. Seguridad

En esta sección daremos una teorema análogo al Teorema 5.4.1 en cual relacionamos la inferencia de tipos de la Definición 79 con las reglas de tipado de la Definición 78, para lo cual primero daremos algunos lemas.

Lema 6.4.1 (Partición)**.** *Si $\bar{\Gamma} \models M : T|_{\mathcal{R}}$ y $\mathtt{X} \in \mathcal{T}_{\mathcal{V}}$ es una variable de tipo tal que $\mathtt{X} \in \mathcal{V}_{\mathcal{T}_{\mathcal{L}}}(\bar{\Gamma})$, entonces $\bar{\Gamma} = \bar{\Gamma}', x{:}\forall.\mathtt{X}$, para alguna $x \in \mathbb{X}$; y además se cumple que $\mathtt{X} \notin \mathcal{V}_{\mathcal{T}}(\bar{\Gamma}')$.*

Demostración. La única regla que genera esquemas de tipo con variables libres es `[p-λx]`; y cuando lo hace son de la forma $\forall.\mathtt{Z}$, donde `Z` es una variable de tipo que nunca se había usado, por lo tanto no está contenida en $\mathcal{V}_{\mathcal{T}}(\bar{\Gamma}')$. □

Lema 6.4.2 (Distribución)**.** *Si $T \in \mathcal{T}$ es un tipo y σ es una sustitución de tipos tal que para toda $\mathtt{X} \in \mathcal{V}_{\mathcal{T}}(T)$ $\mathtt{X} \notin dom(\sigma)$ $\mathtt{X} \notin ran_{\mathcal{T}_{\mathcal{V}}}(\sigma)$, entonces se cumple que*

$$\sigma(\mathtt{gen}(\bar{\Gamma}, T)) = \mathtt{gen}(\sigma(\bar{\Gamma}), \sigma(T)))$$

Demostración. La demostración de este lema se basa en la siguiente ecuación:

$$\mathcal{V}_{\mathcal{T}}(T) \setminus \mathcal{V}_{\mathcal{T}_{\mathcal{L}}}(\bar{\Gamma}) = \mathcal{V}_{\mathcal{T}}(\sigma(T)) \setminus \mathcal{V}_{\mathcal{T}_{\mathcal{L}}}(\sigma(\bar{\Gamma}))$$

Para probar esta ecuación primero observemos que por hipótesis si $\mathtt{X} \in \mathcal{V}_\tau(T)$, entonces $\sigma(\mathtt{X}) = \mathtt{X}$ ya que $\mathtt{X} \notin dom(\sigma)$. De esto se sigue que $\sigma(T) = T$ y por lo tanto $\mathcal{V}_\tau(T) = \mathcal{V}_\tau(\sigma(T))$.

Por lo tanto basta probar que

$$\mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma}) = \mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$$

- Mostraremos la contención $\subseteq$.

  Sea $\mathtt{X} \in \mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$. Tenemos que $\mathtt{X} \in \mathcal{V}_\tau(T)$ y $\mathtt{X} \notin \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$.

  Como $\mathtt{X} \notin ran_{\mathcal{T}_\mathcal{V}}(\sigma)$, entonces tenemos que $\mathtt{X} \notin \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$; y por lo tanto

  $$\mathtt{X} \in \mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$$

- Mostraremos la contención $\supseteq$.

  Sea $\mathtt{X} \in \mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$. Tenemos que $\mathtt{X} \in \mathcal{V}_\tau(T)$ y $\mathtt{X} \notin \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$.

  Supongamos por contradicción que $\mathtt{X} \in \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$. Entonces como $\sigma(\mathtt{X}) = \mathtt{X}$, tendríamos que $\mathtt{X} \in \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$, lo cual es una contradicción.

  Por lo tanto concluimos que

  $$\mathtt{X} \in \mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$$

Hemos mostrado que la siguiente ecuación es válida:

$$\mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma}) = \mathcal{V}_\tau(\sigma(T)) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\sigma(\bar{\Gamma}))$$

Ahora para probar el teorema desarrollaremos ambos lados de la ecuación para obtener el mismo resultado:

- Empezando por el lado izquierdo tenemos que

  $$\sigma(\mathtt{gen}(\bar{\Gamma}, T)) = \sigma(\forall \vec{\mathtt{Z}}.[\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}](T))$$

  donde $\vec{\mathtt{A}} = \mathcal{V}_\tau(T) \setminus \mathcal{V}_{\mathcal{T}_\mathcal{L}}(\bar{\Gamma})$ y $\vec{\mathtt{Z}} \subset \mathbb{N}_\mathbb{V}$. Ahora aplicando la sustitución tenemos que

  $$\begin{aligned} \sigma(\forall \vec{\mathtt{Z}}.[\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}](T)) &= \forall \vec{\mathtt{Z}}.\sigma([\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}](T)) \\ &= \forall \vec{\mathtt{Z}}.\sigma \circ [\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}](T) \end{aligned}$$

- Desarrollando el lado derecho tenemos que

  $$\mathtt{gen}(\sigma(\bar{\Gamma}), \sigma(T)) = \forall \vec{\mathtt{Z}}.[\vec{\mathtt{B}} \mapsto \vec{\mathtt{Z}}](\sigma(T))$$

donde $\vec{\mathtt{B}} = \mathcal{V}_{\mathcal{T}}(\sigma(T)) \setminus \mathcal{V}_{\mathcal{T}_{\mathcal{L}}}(\sigma(\bar{\Gamma}))$ y $\vec{\mathtt{Z}} \subset \mathbb{N}_{\mathbb{V}}$ pero por la ecuación que acabamos de probar podemos escribir esto como

$$\begin{aligned}\mathtt{gen}(\sigma(\bar{\Gamma}), \sigma(T)) &= \forall\vec{\mathtt{Z}}.[\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}](\sigma(T)) \\ &= \forall\vec{\mathtt{Z}}.[\vec{\mathtt{A}} \mapsto \mathtt{Z}] \circ \sigma(T)\end{aligned}$$

Para terminar la prueba solamente hace falta mostrar que

$$\sigma \circ [\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}] = [\vec{\mathtt{A}} \mapsto \mathtt{Z}] \circ \sigma$$

Lo cual es cierto ya que ambas sustituciones son disjuntas:

- $\varnothing = dom(\sigma) \cap dom([\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}])$ porque tenemos que $\vec{\mathtt{A}} \subseteq \mathcal{V}_{\mathcal{T}}(T)$ y por hipótesis $dom(\sigma) \cap \mathcal{V}_{\mathcal{T}}(T) = \varnothing$.
- $\varnothing = dom(\sigma) \cap ran_{\mathcal{T}_{\mathcal{V}}}([\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}])$ porque $ran_{\mathcal{T}_{\mathcal{V}}}([\vec{\mathtt{A}} \mapsto \mathtt{Z}]) \subset \mathbb{N}_{\mathbb{V}}$ y por lo tanto $\vec{\mathtt{Z}} \cap dom(\sigma) = \varnothing$.
- $\varnothing = ran_{\mathcal{T}_{\mathcal{V}}}(\sigma) \cap dom([\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}])$ porque $\vec{\mathtt{A}} \subseteq \mathcal{V}_{\mathcal{T}}(T)$ y por hipótesis $ran_{\mathcal{T}_{\mathcal{V}}}(\sigma) \cap \mathcal{V}_{\mathcal{T}}(T) = \varnothing$.

Por lo tanto se cumple que $\sigma(\mathtt{gen}(\bar{\Gamma}, T)) = \mathtt{gen}(\sigma(\bar{\Gamma}), \sigma(T))$ □

Lema 6.4.3 (Especialización)**.** *Si $\bar{\Gamma} \vdash M : T$ y σ es una sustitución de tipos, entonces $\sigma(\bar{\Gamma}) \vdash M : \sigma(T)$. Si M fuera de la forma $\mathtt{let}\ x = N\ \mathtt{in}\ M'$ donde $\bar{\Gamma} \vdash N : T'$, entonces σ también debe de cumplir que para toda $\mathtt{X} \in \mathcal{V}_{\mathcal{T}}(T')$, $\mathtt{X} \notin dom(\sigma)$ $\mathtt{X} \notin ran_{\mathcal{T}_{\mathcal{V}}}(\sigma)$.*

Demostración. Inducción sobre $\bar{\Gamma} \vdash M : T$.

Sea σ una sustitución de tipos.

- **Caso** `[tp-var]` Sea $\bar{\Gamma} \vdash x : \mathtt{finst}(\bar{\Gamma}(x))$.

  Si consideramos el contexto $\sigma(\bar{\Gamma})$, entonces por la regla `[tp-var]`

  $$\sigma(\bar{\Gamma}) \vdash x : \mathtt{finst}((\sigma(\bar{\Gamma}))(x))$$

  Ahora debemos probar que $\sigma(\mathtt{finst}(\bar{\Gamma}(x))) = \mathtt{finst}((\sigma(\bar{\Gamma}))(x))$.

  Para esto primero veamos que

  - $\mathtt{finst}(\bar{\Gamma}(x)) = \mathtt{finst}(\mathcal{E}) = \mathtt{finst}(\forall \mathtt{A}_1 \ldots \mathtt{A}_n.T) = [\vec{\mathtt{A}} \mapsto \vec{\mathtt{Z}}](T)$
  - $(\sigma(\bar{\Gamma}))(x) = \mathcal{E}' = \sigma(\forall \mathtt{A}_1 \ldots \mathtt{A}_n.T) = \forall \mathtt{A}_1 \ldots \mathtt{A}_n.\sigma_{\neg\vec{\mathtt{A}}}(T)$

Usando el primer punto podemos obtener lo siguiente

$$\begin{aligned}\sigma(\texttt{finst}(\bar{\Gamma}(x))) &= \sigma([\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}](T)) \\ &= (\sigma \circ [\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}])(T) \\ &= (\sigma_{\neg\vec{\texttt{A}}} \circ [\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}])(T)\end{aligned}$$

En el último paso cambiamos σ por $\sigma_{\neg\vec{\texttt{A}}}$ porque después de aplicar la sustitución $[\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}]$ desaparecen todas las variables $\texttt{A}_i$ de T, por lo que aplicar σ es equivalente a aplicar $\sigma_{\neg\vec{\texttt{A}}}$, la cual no tiene asociaciones de la forma $\texttt{A}_i \mapsto T_i$.

Desarrollando el otro lado de la ecuación obtenemos lo siguiente

$$\begin{aligned}\texttt{finst}((\sigma(\bar{\Gamma}))(x)) &= \texttt{finst}(\forall \texttt{A}_1 \ldots \texttt{A}_n.\sigma_{\neg\vec{\texttt{A}}}(T)) \\ &= [\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}](\sigma_{\neg\vec{\texttt{A}}}(T)) \\ &= ([\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}] \circ \sigma_{\neg\vec{\texttt{A}}})(T)\end{aligned}$$

Solamente nos hace falta probar que

$$\sigma_{\neg\vec{\texttt{A}}} \circ [\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}] = [\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}] \circ \sigma_{\neg\vec{\texttt{A}}}$$

para lograr esto mostaremos que ambas sustituciones son disjuntas.

- $dom(\sigma_{\neg\vec{\texttt{A}}}) \cap dom([\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}]) = \varnothing$ porque por definición $\sigma_{\neg\vec{\texttt{A}}}$ no contiene ninguna asociación $\texttt{A}_i \mapsto T$
- $dom(\sigma_{\neg\vec{\texttt{A}}}) \cap ran_{\mathcal{T}_\mathcal{V}}([\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}]) = \varnothing$ porque cada $\texttt{Z}_i$ es una variable nueva que se generó durante la instanciación por lo que no puede aparecer en el dominio de $\sigma_{\neg\vec{\texttt{A}}}$
- $dom([\vec{\texttt{A}} \mapsto \vec{\texttt{Z}}]) \cap ran_{\mathcal{T}_\mathcal{V}}(\sigma_{\neg\vec{\texttt{A}}}) = \varnothing$ porque las variables $\texttt{A}_i$ provienen de la generalización $\texttt{gen}(T)$ y cada una de ellas fue tomada de $\mathbb{N}_\mathbb{V}$ de manera que no pueden ocurrir en $ran_{\mathcal{T}_\mathcal{V}}(\sigma_{\neg\vec{\texttt{A}}})$

Por lo que esta composición de sustituciones conmuta y tenemos que $\sigma(\texttt{finst}(\bar{\Gamma}(x))) = \texttt{finst}((\sigma(\bar{\Gamma}))(x))$

- **Caso** `[tp-cte]` Sea $\dot{c} \in \mathcal{C}_\mathcal{P}$ y $\bar{\Gamma} \vdash \dot{c} : \mathcal{C}_\mathcal{T}(\dot{c})$.

  Como $\sigma(\mathcal{C}_\mathcal{T}(\dot{c})) = \mathcal{C}_\mathcal{T}(\dot{c})$, entonces este caso es trivial pues $\sigma(\bar{\Gamma}) \vdash \mathcal{C}_\mathcal{T}(\dot{c})$

Tomemos por hipótesis de inducción que si $\bar{\Gamma} \vdash M : T$, entonces $\sigma(\bar{\Gamma}) \vdash M : \sigma(T)$.

- **Caso** `[tp-λx]` Supongamos que $\bar{\Gamma}, x : \forall.T_1 \vdash M : T_2$.

  Por hipótesis de inducción tenemos que

$$\sigma(\bar{\Gamma}, x : \forall.T_1) \vdash M : \sigma(T_2)$$

Aplicando σ sobre el contexto de tipado tenemos que

$$\sigma(\bar{\Gamma}), x{:}\forall.\sigma(T_1) \vdash M{:}\sigma(T_2)$$

Aplicando la regla `[tp-λx]` obtenemos que

$$\sigma(\bar{\Gamma}) \vdash \lambda x.M{:}\sigma(T_1) \to \sigma(T_2)$$

Finalmente aplicamos la sustitución en sentido inverso para concluir que

$$\sigma(\bar{\Gamma}) \vdash \lambda x.M{:}\sigma(T_1 \to T_2)$$

- **Caso** `[tp-app]` Supongamos que $\bar{\Gamma} \vdash M{:}T_1 \to T_2$ y $\bar{\Gamma} \vdash N{:}T_1$

  Por hipótesis de inducción tenemos que

  - $\sigma(\bar{\Gamma}) \vdash M{:}\sigma(T_1 \to T_2)$ Yy por lo tanto $\sigma(\bar{\Gamma}) \vdash M{:}\sigma(T_1) \to \sigma(T_2)$
  - $\sigma(\bar{\Gamma}) \vdash N{:}\sigma(T_1)$

  Aplicando la regla `[tp-app]` obtenemos que

  $$\sigma(\bar{\Gamma}) \vdash M\ N{:}\sigma(T_2)$$

- **Caso** `[tp-op]` Supongamos que $\bar{\Gamma} \vdash M_i{:}T_i$ para $1 \leq i \leq n$.

  Por hipótesis de inducción $\sigma(\bar{\Gamma}) \vdash M_i : \sigma(T_i)$ y aplicando la regla `[tp-op]` tenemos que

  $$\sigma(\bar{\Gamma}) \vdash o^{(n)}[M_i] : \Delta(o^{(n)}, [\sigma(T_i)])$$

  Como cada T_i es un tipo concreto (`Nat` o `Bool`) tenemos que $[\sigma(T_i)] = [T_i]$ por lo que obtenemos que

  $$\sigma(\bar{\Gamma}) \vdash o^{(n)}[M_i] : \Delta(o^{(n)}, [T_i])$$

  De manera similar, como $\Delta(o^{(n)}, [T_i])$ es un tipo concreto, se cumple que $\sigma(\Delta(o^{(n)}, [T_i])) = \Delta(o^{(n)}, [T_i])$, por lo que entonces tenemos que

  $$\sigma(\bar{\Gamma}) \vdash o^{(n)}[M_i] : \sigma(\Delta(o^{(n)}, [T_i]))$$

- **Caso** `[tp-fix]` Supongamos que $\bar{\Gamma} \vdash M{:}T \to T$.

  Por hipótesis de inducción tenemos que

  $$\sigma(\bar{\Gamma}) \vdash M{:}\sigma(T \to T)$$

  Aplicamos la sustitución en sentido inverso

  $$\sigma(\bar{\Gamma}) \vdash M{:}\sigma(T) \to \sigma(T)$$

  Y aplicando la regla `[tp-fix]` tenemos que

  $$\sigma(\bar{\Gamma}) \vdash \texttt{Fix}\ M{:}\sigma(T)$$

- **Caso** `[tp-let]` Supongamos que $\bar{\Gamma} \vdash N : T_1 \quad \bar{\Gamma}, x : \mathtt{gen}(\bar{\Gamma}, T_1) \vdash M : T_2$ y que por hipótesis para toda $\mathtt{X} \in \mathcal{V}_\tau(T_1)$ se tiene que $\mathtt{X} \notin dom(\sigma)$ $\mathtt{X} \notin ran_{\mathcal{T}_\mathcal{V}}(\sigma)$.

  Por hipótesis de inducción tenemos que

  $$\sigma(\bar{\Gamma}) \vdash N : \sigma(T_1)$$

  $$\sigma(\bar{\Gamma}, x : \mathtt{gen}(\bar{\Gamma}, T_1)) \vdash M : \sigma(T_2)$$

  Aplicando σ sobre el contexto de tipado tenemos que

  $$\sigma(\bar{\Gamma}), x : \sigma(\mathtt{gen}(\bar{\Gamma}, T_1)) \vdash M : \sigma(T_2)$$

  Y por el lema de distribución esto último es lo mismo que

  $$\sigma(\bar{\Gamma}), x : \mathtt{gen}(\sigma(\bar{\Gamma}), \sigma(T_1)) \vdash M : \sigma(T_2)$$

  De lo anterior y $\sigma(\bar{\Gamma}) \vdash N : \sigma(T_1)$ podemos usar la regla `[tp-let]` para concluir que

  $$\sigma(\bar{\Gamma}) \vdash \mathtt{let}\ x = N\ \mathtt{in}\ M : \sigma(T_2)$$

□

Teorema 6.4.1 (Corrección)**.** *Si* $\bar{\Gamma} \models M : T|_\mathcal{R}$ *y* μ *es un unificador de* $\mathcal{R}$*, entonces se cumple que* $\mu(\bar{\Gamma}) \vdash M : \mu(T)$

Demostración. Haremos inducción sobre la relación $\bar{\Gamma} \models M : T \mid_\mathcal{R}$.

Los casos bases corresponden a las reglas `[p-var]` y `[p-cte]`, para estos el conjunto de restricciones $\mathcal{R}$ es el conjunto vacío por lo que un unificador de ellos sería una sustitución vacía $\mu_\varnothing$. Tendríamos entonces que $\mu_\varnothing(\bar{\Gamma}) = \bar{\Gamma}$ y $\mu_\varnothing(T) = T$ por lo que en ambos casos obtendríamos que $\mu_\varnothing(\bar{\Gamma}) \vdash M : \mu_\varnothing(T)$.

Tomemos como hipótesis de inducción que si $\bar{\Gamma} \models M_i : T_i|_{\mathcal{R}_i}$ y μ_i es un unificador de $\mathcal{R}_i$, entonces $\mu_i(\bar{\Gamma}) \vdash M_i : \mu_i(T_i)$.

- **Caso** `[p-λx]`. Si $\bar{\Gamma}, x : \forall.\mathtt{Z} \models M : T \mid_\mathcal{R}$ donde $\mathtt{Z}$ es una variable nueva, entonces $\bar{\Gamma} \models \lambda x.\, M : \mathtt{Z} \to T \mid_\mathcal{R}$.

  Sea μ un unificador de $\mathcal{R}$, por hipótesis de inducción tenemos que

  $$\mu(\bar{\Gamma}, x : \forall.\mathtt{Z}) \vdash M : \mu(T)$$

  Aplicando el unificador μ sobre el contexto de tipado tenemos que

  $$\mu(\bar{\Gamma}), x : \forall.\mu(\mathtt{Z}) \vdash M : \mu(T)$$

Por la regla `[tp-λx]` podemos concluir que

$$\mu(\bar{\Gamma}) \vdash \lambda x.\, M : \mu(\mathtt{Z}) \to \mu(T)$$

Y aplicando la sustitución de tipos en sentido inverso obtenemos que

$$\mu(\bar{\Gamma}) \vdash \lambda x.\, M : \mu(\mathtt{Z} \to T)$$

- **Caso** `[p-app]` Si $\bar{\Gamma} \models M_1 : T_1 \mid_{\mathcal{R}_1}$ y $\bar{\Gamma} \models M_2 : T_2 \mid_{\mathcal{R}_2}$ y $\mathtt{Z}$ es una variable nueva, entonces $\bar{\Gamma} \models M_1\ M_2 : \mathtt{Z} \mid_{\mathcal{R}}$ donde $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = T_2 \to \mathtt{Z}\}$.

  Sea μ un unificador de $\mathcal{R}$. Observemos que μ también es unificador de $\mathcal{R}_1$ y $\mathcal{R}_2$, pues estos están contenidos en $\mathcal{R}$. Por hipótesis de inducción tenemos

  $$\mu(\bar{\Gamma}) \vdash M_1 : \mu(T_1)$$

  Como μ es un unificador, entonces tenemos que $\mu(T_1) = \mu(T_2 \to \mathtt{Z})$ por lo que tenemos que

  $$\mu(\bar{\Gamma}) \vdash M_1 : \mu(T_2 \to \mathtt{Z})$$

  Y aplicando la sustitución obtenemos que

  $$\mu(\bar{\Gamma}) \vdash M_1 : \mu(T_2) \to \mu(\mathtt{Z})$$

  Ahora, por hipótesis de inducción también tenemos que

  $$\mu(\bar{\Gamma}) \vdash M_2 : \mu(T_2)$$

  Combinando estos últimos dos resultados usando la regla `[tp-app]` obtenemos que

  $$\mu(\bar{\Gamma}) \vdash M_1\ M_2 : \mu(\mathtt{Z})$$

- **Caso** `[p-fix]` Si $\bar{\Gamma} \models M : T \mid_{\mathcal{R}_1}$ y $\mathtt{Z}$ es una variable nueva, entonces tenemos que $\bar{\Gamma} \models \mathtt{Fix}\ M : \mathtt{Z} \mid_{\mathcal{R}}$ donde $\mathcal{R} = \mathcal{R}_1 \cup \{T = \mathtt{Z} \to \mathtt{Z}\}$.

  Sea μ un unificador de $\mathcal{R}$, el cual también unifica a $\mathcal{R}_1$ pues está contenido en $\mathcal{R}$.

  Por hipótesis de inducción tenemos que

  $$\mu(\bar{\Gamma}) \vdash M : \mu(T)$$

  Y como μ es un unificador de $\mathcal{R}$, entonces $\mu(T) = \mu(\mathtt{Z}) \to \mu(\mathtt{Z})$ por lo que también tenemos que

  $$\mu(\bar{\Gamma}) \vdash M : \mu(\mathtt{Z}) \to \mu(\mathtt{Z})$$

  Aplicando la regla `[tp-fix]` a eso obtenemos que

  $$\mu(\bar{\Gamma}) \vdash \mathtt{Fix}\ M : \mu(\mathtt{Z})$$

- **Caso** `[p-let]` Supongamos que $\bar{\Gamma}' \models N : T_1'|_{\mathcal{R}_1}$ y que μ_1 es un unificador de $\mathcal{R}_1$, tal que $\bar{\Gamma} = \mu_1(\bar{\Gamma}')$, $T_1 = \mu_1(T_1')$ y $\bar{\Gamma}, x : \mathtt{gen}(\bar{\Gamma}, T_1) \models M : T_2|_{\mathcal{R}_2}$.

  Sea μ_2 un unificador de $\mathcal{R}_2$. Como $\mathcal{V}_\tau(\mathcal{R}_1) \cap \mathcal{V}_\tau(\mathcal{R}_2) = \varnothing$, entonces para toda $\mathtt{X} \in \mathcal{V}_\tau(T_1)$, $\mathtt{X} \notin dom(\mu_2)$ $\mathtt{X} \notin ran_{\mathcal{T}_\mathcal{V}}(\mu_2)$.

  - Como μ_1 es un unificador de $\mathcal{R}_1$, entonces por hipótesis de inducción tenemos que
    $$\mu_1(\bar{\Gamma}') \vdash N : \mu_1(T_1')$$
    Que por hipótesis es lo mismo que
    $$\bar{\Gamma} \vdash N : T_1$$
    Como μ_2 es una sustitución de tipos tal que $\mathtt{X} \notin dom(\mu_2)$ y $\mathtt{X} \notin ran_{\mathcal{T}_\mathcal{V}}(\mu_2)$, entonces por el lema de especialización tenemos que
    $$\mu(\bar{\Gamma}) \vdash N : \mu(T_1)$$
  - Por hipótesis de inducción tenemos que
    $$\mu_2(\bar{\Gamma}, x : \mathtt{gen}(\bar{\Gamma}, T_1)) \vdash M : \mu_2(T_2)$$
    Aplicando μ sobre el contexto de tipado
    $$\mu_2(\bar{\Gamma}), x : \mu_2(\mathtt{gen}(\bar{\Gamma}, T_1))) \vdash M : \mu_2(T_2)$$
    Y por el lema de distribución tenemos que
    $$\mu_2(\bar{\Gamma}), x : \mathtt{gen}(\mu_2(\bar{\Gamma}), \mu_2(T_1)) \vdash M : \mu_2(T_2)$$

  Combinando los dos resultados anteriores podemos usar la regla `[tp-let]` para concluir que
  $$\mu_2(\bar{\Gamma}) \vdash \mathtt{let}\ \ x = N\ \ \mathtt{in}\ \ M : \mu_2(T_2)$$

□

Con el resultado anterior mostramos que el sistema de tipos es correcto y que implementa correctamente el polimorfismo paramétrico en el lenguaje. En los próximos capítulos nos dedicaremos a diseñar una máquina abstracta que evalúe de manera eficiente nuestro nuevo cálculo polimórfico, el cual puede ser consultado en el Apéndice C.

Parte III

Ejecución

Capítulo 7

Máquinas abstractas

En este capítulo nos dedicaremos a pasar de las reglas de la semántica operacional de ISWIM polimórfico a la definición de una máquina abstracta diseñada por Felleisen y Friedman [16] que nos permitirá implementar de manera eficiente un evaluador del cálculo Λ_p.

Una máquina abstracta se define mediante un conjunto de estados internos y una función de transición que convierte un estado intermedio en otro [19]. El objetivo de una máquina abstracta es hacerla funcionar a partir de un estado inicial el cual contenga el código a ejecutar y que mediante una serie de transiciones deterministas llegue a un estado final, el cual debería de corresponder con el resultado de la evaluación del programa bajo las reglas de la semántica operacional. Mostraremos que una expresión M se evalúa a un valor v si y sólo si, hay un estado inicial que contenga a M tal que llegue a un estado final que contenga a v.

7.1. Semántica operacional de Λ_p

Para empezar esta discusión primero especificaremos la semántica operacional para el cálculo de Λ_p cuya única novedad es la adición de la regla `[ev-let]` para lidiar con la construcción `let` $x = N$ `in` M. Recordemos que esta construcción la agregamos al cálculo con el único propósito de que nos permitiera introducir polimorfismo paramétrico en el tipado, y que operacionalmente no tendríamos ningún inconveniente en haberla declarado como azúcar sintáctica equivalente a $(\lambda x.M)\ N$. La regla `[ev-let]` que daremos hará precisamente esto: una expresión `let` $x = N$ `in` M se reducirá a la aplicación $(\lambda x.M)\ N$.

Antes de definir las reglas de la semántica operacional para Λ_p, deberemos de formalizar su conjunto de valores el cual se ha mantenido sin cambios.

Definición 53: $\mathcal{V}_p$ Valores de ISWIM polimórfico [15]

$$\begin{aligned} \mathcal{V}_p &= x \\ &\mid \dot{c} \\ &\mid \lambda x.M \end{aligned}$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad M \in \Lambda_p$

Las reglas de la semántica operacional para Λ_p se detallan a continuación

Definición 54: Semántica operacional de ISWIM polimórfico [15]

$$\frac{M \to_\Lambda M'}{(M\ N) \to_\Lambda (M'\ N)}\ \texttt{[ev-app-i]}$$

$$\frac{v \in \mathcal{V}_p \qquad N \to_\Lambda N'}{(v\ N) \to_\Lambda (v\ N')}\ \texttt{[ev-app-d]}$$

$$\frac{v \in \mathcal{V}_p}{(\lambda x.M)\ \ v \to_\Lambda M[x \leftarrow v]}\ \texttt{[ev-red]}$$

$$\frac{M \to_\Lambda M'}{\texttt{if}\ \ M\ \ N\ \ L \to_\Lambda \texttt{if}\ \ M'\ \ N\ \ L}\ \texttt{[ev-if-cnd]}$$

$$\frac{}{\texttt{if true}\ \ N\ \ L \to_\Lambda N}\ \texttt{[ev-if-tr]}$$

$$\frac{}{\texttt{if false}\ \ N\ \ L \to_\Lambda L}\ \texttt{[ev-if-fl]}$$

$$sel([M_i]) = \text{mín}(\{i \in \mathbb{N} | M_i \notin \mathcal{C}_{\mathcal{P}}\} \cup \{n+1\})$$

Definición 54: Semántica operacional de ISWIM polimórfico [15] (cont.)

$$\frac{j = sel(\,[M_i]\,) \quad j \leq n \quad M_j \to_\Lambda M'_j}{o^{(n)}\ [\,M_i\,]_{i=1}^n \to_\Lambda o^{(n)}\ [M_1\ \ldots\ M'_j\ \ldots\ M_n]}\ \texttt{[ev-}o^n\texttt{-v]}$$

$$\frac{sel(\,[M_i]\,) = n+1}{o^{(n)}\ [\,M_i\,]_{i=1}^n \to_\Lambda \delta(o^{(n)},\ [\,M_i\,]_{i=1}^n)}\ \texttt{[ev-}o^n\texttt{-}\delta\texttt{]}$$

$$\frac{M \to_\Lambda M'}{\texttt{Fix}\ M \to_\Lambda \texttt{Fix}\ M'}\ \texttt{[ev-fix-b]}$$

$$\frac{}{\texttt{Fix}\ \lambda f.\,M\ \ \to_\Lambda\ \ M[f \leftarrow \texttt{Fix}\ \lambda f.\,M]}\ \texttt{[ev-fix]}$$

$$\frac{}{\texttt{let}\ x = N\ \texttt{in}\ M\ \to_\Lambda (\lambda x.M)\ N}\ \texttt{[ev-let]}$$

Si tenemos una expresión M y nos fijamos en su árbol sintáctico, entonces podemos ver que $M \to_\Lambda M'$ puede ocurrir de dos maneras distintas. Una de ellas es cuando el árbol de M se convierte en M' mediante la modificación de alguno de sus subárboles; el otro caso sucede cuando M se convierte en M' al transformar el nodo raíz en M. En este segundo caso diremos que la forma de M es la de una *expresión reducible* o *redex*. En el cálculo de Λ_p existen cinco posibles redex que definimos a continuación.

Definición 55: Redex de Λ_p [15]

$$\begin{aligned} R \ =\ & (\lambda x.M)\ v \\ \mid\ & \texttt{if}\ v\ M\ N \\ \mid\ & o^{(n)}\ [v_i]_1^n \\ \mid\ & \texttt{Fix}\ \lambda f.M \\ \mid\ & \texttt{let}\ x = N\ \texttt{in}\ M \end{aligned}$$

Donde $v_i \in \mathcal{V} \qquad M, N \in \Lambda_p$

El concepto de redex nos interesa ya que una de las tareas esenciales de una máquina evaluadora consiste en identificar el próximo redex a reducir en una expresión M. En las próximas secciones discutiremos algunos mecanismos mediante los cuales se puede identificar tal redex, así como la manera en la cual se modificará un programa al rededor de él cuando éste sea reducido.

7.2. Contextos de evaluación

Durante la ejecución de la máquina evaluadora ésta tendrá que recorrer el árbol sintáctico de la expresión hasta encontrar el próximo redex a reducir, y de alguna manera debe de poder recordar la forma de la expresión al rededor de la cadena de control actual para poder reescribir la expresión original con el resultado de la reducción que llevará a cabo. Para lograr este efecto definiremos los *contextos de evaluación*.

Definición 56: Contexto de evaluación [15]

Un *contexto de evaluación* representa una expresión de Λ_p la cual contiene un único *agujero* en el que cabe otra expresión de Λ_p. Denotaremos con $\Box$ al agujero en el contexto de evaluación.

$$
\begin{aligned}
C \;=\; & \Box \\
\mid\; & (C'\ N) \\
\mid\; & (v\ C') \\
\mid\; & o^{(n)}[v_1 \ldots v_{j-1}\ C'\ M_{j+1} \ldots M_n] \\
\mid\; & \texttt{if}\ C'\ N\ L \\
\mid\; & \texttt{Fix}\ C'
\end{aligned}
$$

$$\text{Donde } M_i,\ N,\ L \in \Lambda_p \quad v \in \mathcal{V} \quad C' \in C$$

La estructura de un contexto de evaluación siempre inicia con un agujero $\Box$ y ésta va creciendo a manera de una pila según se vayan agregando *marcos* al rededor del agujero. Nos referiremos al primer marco que envuelve al agujero como el *marco base* del contexto.

Si C es un contexto de evaluación definiremos la operación $C[M]$ como el resultado de sustituir el agujero $\Box$ en C con la expresión M. En esta operación es posible que las variables libres de M queden capturadas por C pero no nos preocuparemos por eso.

También definiremos la operación $C[C']$, donde C' es otro contexto de evaluación, como el resultado de sustituir el agujero $\Box$ de C por C'. Notemos que al hacer esto el número de agujeros en C sigue siendo exactamente uno.

Observemos que la manera en la que hemos definido los contextos de evaluación respeta la estrategia de llamada por valor; es decir que $(M\ \Box)$ no es un contexto de evaluación a menos que M sea un valor. Esto nos servirá para mostrar que si respetamos la estrategia de llamada por valor, entonces para toda expresión M si ésta no es un valor, entonces existe un único redex que podemos reducir.

Teorema 7.2.1 (Partición redex [15])**.** *Si $M \in \Lambda_p$ y M puede ser tipado correctamente*[1]*, entonces sucede una de las siguientes*

- *M es un valor*
- *M puede ser particionado de manera única en un contexto de evaluación C y un redex L de manera que $M = C[L]$*

Demostración. Inducción sobre la estructura de M.

Si M es una variable, una constante o una función; entonces M cumple con el teorema pues éstos son precisamente los valores de Λ_p. Supongamos ahora que M no es de tal forma.

- **Caso** `[app]` Si $M = (M_1\ M_2)$ entonces tenemos tres opciones.
  - Si M_1 y M_2 son ambos valores, entonces como M está correctamente tipado M_1 es de la forma $\lambda x.N$ y tenemos entonces que $(\lambda x.N)\ M_2$ es un redex por lo que podemos separar a M en este redex y el contexto vacío $\Box$ y tendríamos que $M = \Box[(\lambda x.N)\ M_2]$.
  - Si M_1 es un valor pero M_2 no lo es, entonces por hipótesis de inducción podemos separar de manera única a M_2 en un contexto C_2 y un redex L_2 tales que $M_2 = C_2[L_2]$. Usando esto podemos tomar $C = (M_1\ C_2)$, el cual es un contexto de evaluación porque M_1 es un valor, y con esto tendríamos que $M = C[L_2]$.
  - Si M_1 no es un valor entonces por hipótesis de inducción tenemos que $M_1 = C_1[L_1]$ y usamos C_1 para crear un nuevo contexto $C = (C_1\ M_2)$ y con esto tenemos que $M = C[L_1]$.
- **Caso** `[op[M]]` Si $M = o^{(n)}\,[M_i]$ entonces debemos de fijarnos en cuántos de los M_i son valores. Si todos los M_i son valores, entonces M es un redex y usando el contexto vacío tenemos que $M = \Box[o^{(n)}\,[M_i]]$. Si no todos los M_i

[1] Cuando decimos que un programa M puede ser tipado correctamente, deberá entenderse que existe un tipo T tal que $\vdash M : T$

son valores entonces podemos encontrar una j mínima tal que M_j no es un valor. Por hipótesis de inducción tenemos que $M_j = C_j[L_j]$ y con esto podemos crear un nuevo contexto $C = o^{(n)}\,[M_1 \ldots M_{j-1}\ C_j\ M_{j+1} \ldots M_n]$ y con esto tendríamos que $M = C[L_j]$.

- **Caso** `[if]` Si $M = \texttt{if}\ M_1\ N\ L$ y M_1 es un valor entonces tenemos que M es un redex y por lo tanto $M = \Box[M]$. Si M_1 no es un valor entonces por hipótesis de inducción $M_1 = C_1[L_1]$ y tomando $C = \texttt{if}\ C_1\ N\ L$ obtenemos que $M = C[L_1]$.
- **Caso** `[fix]` Si $M = \texttt{Fix}\ N$, entonces consideraremos dos opciones sobre N.
  - Si N es un valor, entonces porque M está correctamente tipado N debe de ser de la forma $\lambda f.N'$. Entonces tenemos que $\texttt{Fix}\ \lambda f.N'$ es un redex, y usando el contexto vacío tenemos que $M = \Box[\lambda f.N']$
  - Si N no es un valor, entonces por hipótesis de inducción, existe un contexto C' y un redex L tal que $N = C'[L]$. Definimos entonces el contexto $C = \texttt{Fix}\ C'$ de manera que $M = C[L]$
- **Caso** `[let]` Esta expresión es un redex por sí misma, por lo que usando el contexto vacío obtenemos que $M = \Box[M]$.

□

7.3. Expresiones bloqueadas y progreso

Cuando pensamos en ejecutar un programa M de ISWIM, lo que esperamos es que podamos aplicar una secuencia determinista de reducciones basadas en la semántica operacional de manera que M se reduzca eventualmente a un valor. Desafortunadamente no podemos asegurar que cualquier programa tenga este comportamiento; existen programas que pueden estar tipados correctamente cuya secuencia de reducciones es infinita, por ejemplo consideremos la ejecución del siguiente programa cuyo tipo es `Bool` en el cual se puede observar que su evaluación se cicla infinitamente

$$M \stackrel{\text{def}}{=} \texttt{Fix}\ (\lambda f.\ \texttt{if true}\ f\ \texttt{false})$$

$$M \to_\Lambda \texttt{if true}\ M\ \texttt{false} \to_\Lambda M \to_\Lambda \texttt{if true}\ M\ \texttt{false} \to_\Lambda M \to_\Lambda \ldots$$

Parecería entonces que M tiene dos opciones: ya sea reducirse eventualmente a un valor, o que su evaluación jamás termine. En realidad hay una tercera opción y ésta es que M no sea un valor, pero que no exista un M' tal que $M \to_\Lambda M'$; por

ejemplo si consideramos que `/` es el operador primitivo que divide dos números tenemos que `/` $[2,\ 0]$ puede ser tipado correctamente y que no es un valor, sin embargo ya que la división entre cero está indefinida no existe un M' tal que `/` $[2,\ 0] \to_{\Lambda} M'$. En este caso diremos que la expresión `/` $[2,\ 0] \to_{\Lambda} M'$ es una *expresión bloqueada.*

Definición 57: Expresiones bloqueadas [15]

Si M no es un valor, entonces decimos que M es una *expresión bloqueada* si no existe un M' tal que $M \to_{\Lambda} M'$

Si M puede ser tipado correctamente, entonces solamente existe una posible forma en la que la evaluación de M llegue a una expresión bloqueada, y ésta involucra a la función parcial δ (Def. 18). Si δ no está definida para alguna combinación particular de operador y operandos, entonces no existe una transición que permita que la evaluación progrese.

Teorema 7.3.1 (Expresión bloqueada [15])**.** *Si M es un programa correctamente tipado, M es una expresión bloqueada si y sólo si $M = C[L]$, donde C es un contexto de evaluación y L es un redex de la forma $(o^{(n)}[\dot{c}_i]_1^n)$, para el cual la función $\delta(o^{(n)}, [\dot{c}_i])$ no está definida.*

Demostración.

- [$\Rightarrow$] Supongamos que M es una expresión bloqueada correctamente tipada.

  Como M está bloqueada entonces M no es un valor, por lo que por el Teorema 7.2.1 existe un único contexto de evaluación C y un redex L tal que $M = C[L]$. Si hacemos un análisis sobre las posibles formas de L veremos que la única posibilidad en la que M esté bloqueada es cuando L es de la forma $(o^{(n)}[\dot{c}_i]_1^n)$.

  - Si $L = (\lambda x.M)\ \ v$, donde v es un valor, entonces $L \to_{\Lambda} M[x \leftarrow v]$; por lo que M no está bloqueada si L es de esta forma.
  - Si $L =$ `if` $v\ N_1\ N_2$, donde v es un valor, y por estar correctamente tipado v : `Bool`, entonces $L \to_{\Lambda} L'$ donde L' es N_1 o N_2 dependiendo de si v es `true` o `false` respectivamente; por lo que M no está bloqueada si L es de esta forma.
  - Si $L = (o^{(n)}[\dot{c}_i]_1^n)$, entonces $L \to_{\Lambda} \delta(o^{(n)}, [\dot{c}_i])$ a menos que $\delta(o^{(n)}, [\dot{c}_i])$ no esté definida.
  - Si $L =$ `Fix` $\lambda f.N$, entonces $L \to_{\Lambda} N[f \leftarrow L]$; por lo que M no está bloqueada si L es de esta forma.
  - Si $L =$ `let` $x = N$ `in` M', entonces $L \to_{\Lambda} (\lambda x.M')\ N$; por lo que M no está bloqueada si L es de esta forma.

- [⇐] Si M puede ser particionado en un contexto de evaluación C y un redex L de la forma $(o^{(n)}[\dot{c}_i]_1^n)$ tal que $M = C[L]$, entonces tenemos que M no es un valor por el Teorema 7.2.1; y como por hipótesis $\delta(o^{(n)}, [\dot{c}_i])$ no está definida, tenemos entonces que no existe un M' tal que $M \to_\Lambda M'$ por lo que M es una expresión bloqueada.

□

Ahora que hemos caracterizado la única forma en la cual una expresión M puede quedar bloqueada, podemos demostrar que si M no es una expresión bloqueada ni un valor, entonces necesariamente existe un M' tal que $M \to_\Lambda M'$. A esta propiedad se le conoce como *progreso*.

Teorema 7.3.2 (Progreso [15])**.** *Si $M \in \Lambda_p$ donde $\vdash M : T$, entonces ocurre una de las siguientes*

1. *M es un valor*
2. *$M \to_\Lambda M'$ para alguna $M' \in \Lambda_p$*
3. *M es una expresión bloqueada*

Demostración. Inducción sobre M

Los casos base corresponden a las construcciones `[cte]` y `[λx.M]`; para ambos casos tenemos que M es un valor, por lo que cumplen con el teorema.

Obsérvese que la construcción `[var]` no se considera ya que no cumple con las premisas del teorema al ser imposible tipar a $x \in \mathbb{X}$ bajo el contexto vacío.

Supongamos ahora que para $M_1, M_2 \in \Lambda_p$ si $\vdash M_i : T_i$, entonces M_i cumple con la propiedad de *progreso*.

- **Caso** `[app]` Supongamos que $\vdash M_1\ M_2 : T$, entonces por inversión de la regla de tipado `[tp-app]` tenemos que $\vdash M_1 : T_1 \to T$ y $\vdash M_2 : T_1$. Por hipótesis de inducción tenemos que M_1 y M_2 cumplen con la propiedad de progreso.

  Ahora debemos mostrar que $M_1\ M_2$ cumple con la propiedad de progreso. Para esto analizaremos las posibles formas en las que M_1 y M_2 cumplen con esta propiedad:

  - Si M_1 es un valor, entonces como $M_1\ M_2$ está bien tipado tenemos que M_1 es de la forma $\lambda x.N$, por lo que $M_1\ M_2 = (\lambda x.N)\ M_2$. Ahora debemos analizar el progreso de M_2:

- Si $M_2 = v$ es un valor, entonces tenemos que $M_1\ M_2 = (\lambda x.N)\ v \rightarrow_\Lambda N[x \leftarrow v]$ por la regla `[ev-red]` de la semántica operacional. Por lo tanto tenemos que en este caso se cumple con el segundo inciso del progreso.
- Si $M_2 \rightarrow_\Lambda M_2'$, entonces tenemos que $M_1\ M_2 \rightarrow_\Lambda M_1\ M_2'$ por la regla `[ev-app-d]` cumpliendo así con el segundo inciso de la propiedad.
- Si M_2 es una expresión bloqueada, entonces no podemos aplicar la regla `[ev-app-d]`, por lo que $M_1\ M_2$ es una expresión bloqueada.

- Si $M_1 \rightarrow_\Lambda M_1'$, entonces tenemos que $M_1\ M_2 \rightarrow_\Lambda M_1'\ M_2$ por la regla `[ev-app-i]` de la semántica operacional. Por lo que en este caso también se cumple el segundo inciso del progreso.
- Si M_1 es una expresión bloqueada, entonces no podemos aplicar la regla `[ev-app-i]`, por lo que $M_1\ M_2$ es una expresión bloqueada.

- La pruebas para los casos faltantes son análogas a la anterior y se omiten.

□

La propiedad de progreso nos permite tener la confianza de que evaluar una expresión de Λ_p no es un esfuerzo inútil por lo que en las siguiente secciones nos dedicaremos a desarrollar máquinas abstractas que realicen tal tarea de manera eficiente.

7.4. La máquina CC

En esta sección desarrollaremos una máquina evaluadora para el cálculo Λ_p aprovechando el resultado que dice que si una expresión no es un valor, entonces es posible identificar de manera determinista el próximo redex a reducir. Esta máquina tendrá un estado interno de la forma $\langle M, C\rangle$ donde la primera componente es una expresión de Λ_p la cual representa a la subexpresión que estamos evaluando, y siguiendo la tradición le llamaremos la *cadena de control*. La segunda componente es un contexto de evaluación que sirve para recordar la parte del cómputo al rededor de la cadena de control que ha quedado pendiente. Las transiciones funcionan a grandes rasgos de la siguiente manera:

- $\langle M, \Box\rangle$ es el estado inicial de la máquina
- $\langle v,\ C\rangle \longmapsto \langle C[v],\ \Box\rangle$ cuando v es un valor
- $\langle M,\ C\rangle \longmapsto \langle L',\ C[C']\rangle$ cuando $M = C'[L]$ y el redex L se reduce a L'

- $\langle v, \Box\rangle$ es el estado final de la máquina cuando v es un valor.

La definición de esta máquina, a pesar de que es correcta [15], tiene algunos incovenientes. El primero de ellos es que en la transición $\langle v, C\rangle \longmapsto \langle C[v], \Box\rangle$ le estamos pidiendo a la máquina que vuelva a buscar desde la raíz de la expresión el próximo redex a reducir, cuando en realidad podríamos ahorrarnos esta búsqueda si nos apoyamos de la información que nos provee el contexto de evaluación. El segundo inconveniente es que no queda del todo claro cómo particionar a M en un contexto de evaluación y un redex. Solucionaremos ambos problemas con la definición de la máquina CC. En el Apéndice D se encuentra una versión más grande de esta definición.

Definición 58: Máquina CC [15]

Definimos el comportamiento de la máquina CC mediante la siguiente tabla de transiciones donde para cada estado $\langle M, C\rangle$ inspeccionamos la forma de M y C para especificar el siguiente estado de la máquina. Para el caso de los contextos de evaluación nos fijaremos en la forma del *marco base* mediante la notación $C[(\Box\ N)]$ a veces sustituiremos el marco base por uno distinto pero mantendremos el resto del contexto intacto como en la regla `[cc-fn]`, en otros casos eliminaremos el marco base del contexto y este lo sustituiremos por un agujero como en el caso de la regla `[cc-ar]`. Usamos la letra v para especificar que M debe ser un valor.

M	C	$\overset{}{\underset{\text{CC}}{\mapsto}}$	M	C	
v	$C\,[(\Box\ N)]$		N	$C[(v\ \Box)]$	`[cc-fn]`
v	$C\,[((\lambda x.N)\ \Box)]$		$N[x \leftarrow v]$	C	`[cc-ar]`
v	$C\,[o^{(n)}\ [v_1 \dots \Box\ M_{j+1} \dots M_n]]$		M_{j+1}	$C[o^{(n)}[v_1 \dots v\ \Box\ M_{j+2} \dots M_n]]$	`[cc-opv]`
v	$C\,[o^{(n)}\ [v_1 \dots v_{n-1}\ \Box]]$		$\delta(o^{(n)}, [v_i])$	C	`[cc-δ]`
v	$C\,[\texttt{if}\ \Box\ N\ L]$		$\begin{cases} N & \text{si } v = \texttt{true} \\ L & \text{si } v = \texttt{false} \end{cases}$	C	`[cc-frk]`
$\lambda f.M$	$C\,[\texttt{Fix}\ \Box]$		$M[f \leftarrow \texttt{Fix}\ \lambda f.M]$	C	`[cc-fix]`
$(M\ N)$	C		M	$C[(\Box\ N)]$	`[cc-ap]`
$o^{(n)}[M_i]$	C		M_1	$C[o^{(n)}[\Box\ M_2 \dots M_n]]$	`[cc-op]`
$\texttt{if}\ M\ N\ L$	C		M	$C[\texttt{if}\ \Box\ N\ L]$	`[cc-if]`
$\texttt{Fix}\ M$	C		M	$C[\texttt{Fix}\ \Box]$	`[cc-fix-b]`
$\texttt{let}\ x = N\ \texttt{in}\ M$	C		$(\lambda x.M)\ N$	C	`[cc-let]`
v	$\Box$		*estado final*		`[cc-fin]`

La máquina CC es correcta en el sentido de que modela las reglas de la semántica operacional tal y como muestra el Teorema 7.4.1. Con esta máquina hemos solucionado los dos problemas que teníamos con la definición anterior, siendo esto que el comportamiento de la máquina ha quedado especificado de manera precisa para todas las posibles construcciones de Λ_p, y también hemos solucionado el problema de tener que volver a recorrer toda la expresión desde la raíz para encontrar el próximo redex a reducir. No obstante, la definición de la máquina CC aún conlleva algunos inconvenientes que afectarían la eficiencia de una implementación real. En las siguientes secciones nos dedicaremos a solucionar estos problemas, pero primero mostraremos que la máquina CC modela de manera precisa a las reglas de la semántica operacional.

Teorema 7.4.1 ($\to_\Lambda^* \Leftrightarrow \underset{\text{CC}}{\mapsto^*}$ [15])**.** *Si C es un contexto de evaluación, $M \in \Lambda_p$ y v es un valor, entonces se tiene que*

$$C[M] \to_\Lambda^* v \iff \langle M, C\rangle \underset{CC}{\mapsto^*} \langle v, \Box\rangle$$

Demostración. Probaremos cada sentido de la equivalencia mediante inducción sobre la longitud de la derivación.

- [⇒] Supongamos que $C[M] \to_\Lambda^* v$.

  - Si $C[M] \to_\Lambda^0 v$, entonces tenemos que $C[M] = v$, $C = \Box$ y $M = v$. Por lo tanto $\langle M, C\rangle = \langle v, \Box\rangle$ y así $\langle M, C\rangle \underset{\text{CC}}{\mapsto^*} \langle v, \Box\rangle$

  Tomemos por hipótesis de inducción que si $C[M] \to_\Lambda^n v$, entonces $\langle M, C\rangle \underset{\text{CC}}{\mapsto^*} \langle v, \Box\rangle$

  - Supongamos que $N \to_\Lambda^{n+1} v$. Por el Teorema 7.2.1 $N = C[L]$, donde L es un redex tal que $L \to_\Lambda L'$ y por lo tanto $C[L] \to_\Lambda C[L'] \to_\Lambda^n v$. Por hipótesis de inducción $\langle L', C\rangle \underset{\text{CC}}{\mapsto^*} \langle v, \Box\rangle$.

    Ahora sólo hace falta mostrar que $\langle L, C\rangle \underset{\text{CC}}{\mapsto^*} \langle L', C\rangle$. Para esto analizaremos las posibles formas que puede tener la transición $L \to_\Lambda L'$.

    - Si $L = ((\lambda x.N')\ v)$ y $L' = N'[x \leftarrow v]$ entonces tenemos que

      $$\begin{aligned}\langle L, C\rangle &= \langle (\lambda x.N')\ v, C\rangle \\ &\underset{\text{CC}}{\mapsto} \langle \lambda x.N', C[\Box\ v]\rangle && \texttt{[cc-ap]} \\ &\underset{\text{CC}}{\mapsto} \langle v, C[(\lambda x.N')\ \Box]\rangle && \texttt{[cc-fn]} \\ &\underset{\text{CC}}{\mapsto} \langle N'[x \leftarrow v], C\rangle && \texttt{[cc-ar]} \\ &= \langle L', C\rangle\end{aligned}$$

    - Si $L = \texttt{if true}\ N_1\ N_2$ y $L' = N_1$

      $$\begin{aligned}\langle L, C\rangle &= \langle \texttt{if true}\ N_1\ N_2, C\rangle \\ &\underset{\text{CC}}{\mapsto} \langle N_1, C\rangle && \texttt{[cc-frk]} \\ &= \langle L', C\rangle\end{aligned}$$

    - Si $L = \texttt{if false}\ N_1\ N_2$ y $L' = N_2$

      $$\begin{aligned}\langle L, C\rangle &= \langle \texttt{if false}\ N_1\ N_2, C\rangle \\ &\underset{\text{CC}}{\mapsto} \langle N_2, C\rangle && \texttt{[cc-frk]} \\ &= \langle L', C\rangle\end{aligned}$$

- Si $L = o^{(n)}[v_i]$ y $L' = \delta(o^{(n)}, [v_i])$

$$\begin{aligned}
\langle L, C\rangle &= \langle o^{(n)}[v_i], C\rangle \\
&\underset{\text{CC}}{\mapsto} \langle v_1, C[o^{(n)}\ [\Box\ v_2 \dots v_n]]\rangle && \texttt{[cc-op]} \\
&\underset{\text{CC}}{\mapsto} \langle v_2, C[o^{(n)}\ [v_1\ \Box\ v_3 \dots v_n]\rangle && \texttt{[cc-opv]} \\
&\qquad\vdots \\
&\underset{\text{CC}}{\mapsto} \langle v_n, C[o^{(n)}\ [v_1 \dots v_{n-1}\ \Box]\rangle && \texttt{[cc-opv]} \\
&\underset{\text{CC}}{\mapsto} \langle \delta(o^{(n)}, [v_i], C\rangle && \texttt{[cc-}\delta\texttt{]} \\
&= \langle L', C\rangle
\end{aligned}$$

- Si $L = \texttt{Fix}\ \lambda f.N'$ y $L' = N'[f \leftarrow \texttt{Fix}\ \lambda f.N']$

$$\begin{aligned}
\langle L, C\rangle &= \langle \texttt{Fix}\ \lambda f.N', C\rangle \\
&\underset{\text{CC}}{\mapsto} \langle N'[f \leftarrow \texttt{Fix}\ \lambda f.N'], C\rangle && \texttt{[cc-fix]} \\
&= \langle L', C\rangle
\end{aligned}$$

- Si $L = \texttt{let}\ \ x = N_1\ \ \texttt{in}\ \ N_2$ y $L' = (\lambda x.N_2)\ N_1$

$$\begin{aligned}
\langle L, C\rangle &= \langle \texttt{let}\ \ x = N_1\ \ \texttt{in}\ \ N_2, C\rangle \\
&\underset{\text{CC}}{\mapsto} \langle (\lambda x.N_2), C\rangle && \texttt{[cc-let]} \\
&= \langle L', C\rangle
\end{aligned}$$

Por lo tanto se cumple que si $C[M] \overset{*}{\to}_{\Lambda} v$, entonces $\langle M, C\rangle \overset{*}{\underset{\text{CC}}{\mapsto}} \langle v, \Box\rangle$.

- [$\Leftarrow$] Supongamos que $\langle M, C\rangle \overset{*}{\underset{\text{CC}}{\mapsto}} \langle v, \Box\rangle$.

  - Si $\langle M, C\rangle \overset{0}{\underset{\text{CC}}{\mapsto}} \langle v, \Box\rangle$, entonces $M = v$ y $C = \Box$, por lo que entonces $v = \Box[v] = C[M] \overset{0}{\to}_{\Lambda} v$

  Tomemos por hipótesis de inducción que si $\langle M, C\rangle \underset{\text{CC}}{\mapsto}^{n} \langle v, \Box\rangle$, entonces $C[M] \overset{*}{\to}_{\Lambda} v$

  - Supongamos que $\langle M', C'\rangle \overset{n+1}{\underset{\text{CC}}{\mapsto}} \langle v, \Box\rangle$, entonces tenemos que

$$\langle M', C'\rangle \underset{\text{CC}}{\mapsto} \langle M, C\rangle \overset{n}{\underset{\text{CC}}{\mapsto}} \langle v, \Box\rangle$$

    Por hipótesis de inducción $C[M] \overset{*}{\to}_{\Lambda} v$, por lo que solamente hace falta probar que $C'[M'] \overset{*}{\to}_{\Lambda} C[M]$. Para mostrar esto consideraremos las posibles reglas que usa la transición $\langle M', C'\rangle \underset{\text{CC}}{\mapsto} \langle M, C\rangle$.

- Para las reglas `[cc-fn]`, `[cc-opv]`, `[cc-ap]`, `[cc-op]`, `[cc-frk]` y `[cc-fix-b]` tenemos que $C'[M'] = C[M]$, por lo tanto $C'[M'] \overset{0}{\to}_{\Lambda} C[M]$
- Para las reglas `[cc-ar]`, `[cc-`δ`]`, `[cc-frk]` `[cc-fix]` y `[cc-let]` tenemos que M' es un redex, por lo que usando respectivamente las reglas `[ev-red]`, `[ev-if-tr]` o `[ev-if-fl]`, `[ev-`o^n`-`δ`]`, `[ev-fix]` y `[ev-let]` obtenemos que $M' \to_{\Lambda} M$ y $C' = C$; por lo tanto $C'[M'] \to_{\Lambda} C[M]$.

Por lo tanto se cumple que si $\langle M, C\rangle \underset{\text{CC}}{\overset{*}{\mapsto}} \langle v, \Box\rangle$, entonces $C[M] \overset{*}{\to}_{\Lambda} v$.

□

Corolario 7.4.1. *La máquina CC modela las reglas de la semántica operacional, es decir que* $M \overset{*}{\to}_{\Lambda} v$ *si y sólo si* $\langle M, \Box\rangle \underset{CC}{\overset{*}{\mapsto}} \langle v, \Box\rangle$

7.5. Continuaciones y la máquina CK

El primero de los problemas de la máquina CC tiene que ver con una mala elección para la estructura de datos con la que representamos los contextos de evaluación. El problema viene de que en cada transición estamos interesados en leer y modificar el contexto de evaluación al rededor del marco base, y éste se encuentra en el fondo de la estructura, por lo que en cada transición debemos de recorrerla por completo para llegar a él.

Este comportamiento es altamente ineficiente y podríamos solucionarlo si junto con la estructura incluyéramos un apuntador al fondo de ésta, de manera que siempre podamos acceder al marco base en tiempo constante. El problema de esto es que para quitar el marco base y sustituirlo por un *agujero* necesitaríamos un apuntador del marco base al inmediato superior, lo que finalmente nos obligaría a hacer que toda la estructura estuviera doblemente ligada de arriba a abajo y de abajo arriba. En realidad no nos interesa seguir los apuntadores originales que van de arriba a abajo, por lo que podríamos eliminarlos.

La solución que daremos será dar una estructura de datos equivalente a los contextos de evaluación en el sentido de que ambas estructuras contendrían exactamente la misma información. La diferencia ocurre en la manera en la que se anidan los marcos. Esta nueva estructura, a la que llamaremos *continuación*, se parece estructuralmente a una lista ligada donde el marco base siempre está a la cabeza de la lista, permitiendo con esto leerlo y modificarlo en tiempo constante.

Definición 59: Continuaciones [15]

$$
\begin{aligned}
K \;=\;& \texttt{Mt} \\
\mid\;& \texttt{Ar}\ N\ \kappa \\
\mid\;& \texttt{Fn}\ v\ \kappa \\
\mid\;& \texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \kappa \\
\mid\;& \texttt{If}\ N\ L\ \kappa \\
\mid\;& \texttt{Fix}\ \ \kappa
\end{aligned}
$$

Donde $M_i,\ N,\ L \in \Lambda_p \quad v_i \in \mathcal{V} \quad \kappa \in K$

Hemos definido la continuaciones con el propósito de que su implementación sea eficiente y directa de la definición, para esto último hemos incluido etiquetas que fungen como constructores a la vez que sugieren su significado [15].

- `Mt` es el equivalente al agujero $\square$ de los contextos de evaluación. La etiqueta viene de la palabra `EMPTY` que denota a la continuación vacía.
- `Ar` $N\ \kappa$ representa al cómputo pendiente que sabe que N es un argumento para alguna función que está esperando recibir, y que cuando lo haga se deberá evaluar el argumento N hasta convertirlo en un valor para poder aplicar la función sobre él; y así obtener una nueva expresión la cual está siendo esperada por la continuación κ. Esta continuación es el equivalente al contexto $C[(\square\ N)]$ donde C y κ son equivalentes.
- `Fn` $v\ \kappa$ representa al cómputo pendiente que sabe que v es una función y que espera recibir un valor para poder aplicar la función sobre él, obteniendo así una nueva expresión la cual está siendo esperada por κ. Esta continuación es el equivalente al contexto $C[(v\ \ \square)]$ donde C y κ son equivalentes.
- `Op` $[v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \kappa$ representa al cómputo cuyo objetivo final es reducir n expresiones a constantes primitivas para poder aplicar sobre ellas un operador primitivo de aridad n. Esta continuación tiene dos listas donde acumula los valores obtenidos de evaluar las primeras $j-1$ expresiones, y otra en la que guarda las $n-j$ expresiones que aún deben ser reducidas a un valor. La lista que acumula los valores obtenidos está invertida en cuanto al orden de los índices, esto es que el valor v_1 está al final de la lista y el último valor obtenido está a la cabeza. Esto lo hacemos para que cuando la continuación reciba el j-ésimo valor, pueda agregarlo a la lista

en tiempo constante y al mismo tiempo pueda quitar la expresión M_{j+1} de la segunda lista también en tiempo constante. La lista de valores y la de expresiones está separada por el operador $o^{(n)}$ de manera que cuando la continuación reciba el valor v_n, dejando a la segunda lista vacía, entonces podamos agregar a v_n a la cabeza de la primera lista para luego aplicar la función δ sobre su reversa y devolverle ese valor a la continuación κ. Esta continuación es equivalente al contexto $C[o^{(n)}\ [v_1 \dots v_{j-1} \Box\ M_{j+1} \dots M_n]]$ donde C y κ también son equivalentes.

- `If` $N\ L\ \kappa$ representa al cómputo que está esperando recibir un valor booleano, y dependiendo de si éste es `true` o `false`, entonces devuelve N o L respectivamente a la continuación κ. Esta continuación es el equivalente al contexto $C[\texttt{if}\ \Box\ N\ L]$ cuando C y κ son equivalentes.

- `Fix` κ representa al cómputo pendiente que está esperando recibir una función $\lambda f.M$, de manera que cuando la reciba pueda devolver la expresión $M[f \leftarrow Fix\ \lambda f.M]$ a la continuación κ. Esta continuación es equivalente al contexto $C[\texttt{Fix}\ \Box]$ cuando C y κ son equivalentes.

Con lo anterior podemos dar una función que transforme contextos de evaluación en continuaciones equivalentes.

Definición 60: Φ Equivalencia entre continuaciones y contextos [15]

Definimos la función $\Phi : C \to K$ que toma un contexto de evaluación y lo transforma en una continuación equivalente de la siguiente manera

$$\begin{aligned}
\Phi(\Box) &= \texttt{Mt} \\
\Phi(C[(\Box\ N)]) &= \texttt{Ar}\ N\ \Phi(C) \\
\Phi(C[(v\ \Box)]) &= \texttt{Fn}\ v\ \Phi(C) \\
\Phi(C[o^{(n)}\ [v_1 \dots v_{j-1} \Box\ M_{j+1} \dots M_n]]) &= \texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \Phi(C) \\
\Phi(C[\texttt{if}\ \Box\ N\ L]) &= \texttt{If}\ N\ L\ \Phi(C) \\
\Phi(C[\texttt{Fix}\ \Box]) &= \texttt{Fix}\ \Phi(C)
\end{aligned}$$

Por supuesto también podemos definir la función inversa $\Phi^{-1} : K \to C$ que toma una continuación y la transforma en un contexto de evaluación

Definición 60: Φ Equivalencia entre continuaciones y contextos [15] (cont.)

equivalente

$$\begin{aligned}
\Phi^{-1}(\texttt{Mt}) &= \Box \\
\Phi^{-1}(\texttt{Ar}\ N\ \kappa) &= (\Phi^{-1}(\kappa))[(\Box\ N)] \\
\Phi^{-1}(\texttt{Fn}\ v\ \kappa) &= (\Phi^{-1}(\kappa))[(v\ \Box)] \\
\Phi^{-1}(\texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \kappa) &= (\Phi^{-1}(\kappa))[o^{(n)}\ [v_1 \dots v_{j-1}\ \Box\ M_{j+1} \dots M_n]] \\
\Phi^{-1}(\texttt{If}\ N\ L\ \kappa) &= (\Phi^{-1}(\kappa))[\texttt{if}\ \Box\ N\ L] \\
\Phi^{-1}(\texttt{Fix}\ \kappa) &= (\Phi^{-1}(\kappa))[\texttt{Fix}\ \Box]
\end{aligned}$$

Escribiremos $C \approx \kappa$ ó $\kappa \approx C$ cuando $\Phi(C) = \kappa$

Por ejemplo si tenemos el contexto de evaluación $((\lambda x.M\ (\texttt{Fix}\ \Box))\ N)$ también lo podemos expresar como $((\Box\,[(\Box\ N)])\,[((\lambda x.M)\ \Box)])\,[\texttt{Fix}\ \Box]$ y podemos aplicar la función Φ sobre él para obtener la continuación correspondiente.

$$\begin{aligned}
\Phi(((\Box\,[(\Box\ N)])\,[((\lambda x.M)\ \Box)])\,[\texttt{Fix}\ \Box]) &= \texttt{Fix}\ \Phi((\Box\,[(\Box\ N)])\,[((\lambda x.M)\ \Box)]) \\
&= \texttt{Fix}\ (\texttt{Fn}\ (\lambda x.M)\ \Phi(\Box\,[(\Box\ N)])) \\
&= \texttt{Fix}\ (\texttt{Fn}\ (\lambda x.M)\ (\texttt{Ar}\ N\ \Phi(\Box)) \\
&= \texttt{Fix}\ (\texttt{Fn}\ (\lambda x.M)\ (\texttt{Ar}\ N\ \texttt{Mt})
\end{aligned}$$

Damos ahora la definición de la máquina CK. En el Apéndice D se encuentra una versión más grande de esta definición.

Definición 61: La máquina CK [15]

Definimos la máquina CK como se muestra en la tabla de transiciones, para estados de la forma $\langle M, \kappa \rangle$ donde $M \in \Lambda_p$ y $\kappa \in K$. El estado inicial de la máquina es $\langle M, \texttt{Mt} \rangle$.

Definición 61: La máquina CK [15] (cont.)

M	κ	$\overset{}{\underset{\text{CK}}{\mapsto}}$	M	κ	
v	$\texttt{Ar}\ N\ \kappa$		N	$\texttt{Fn}\ v\ \kappa$	[ck-fn]
v	$\texttt{Fn}\ (\lambda x.M)\ \kappa$		$M[x \leftarrow v]$	κ	[ck-ar]
v	$\texttt{Op}\ [v_i]_{i-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \kappa$		M_{j+1}	$\texttt{Op}\ [v_i]_{j}^{1}\ o^{(n)}\ [M_i]_{j+2}^{n}\ \kappa$	[ck-opv]
v	$\texttt{Op}\ [v_i]_{n-1}^{1}\ o^{(n)}\ []\ \kappa$		$\delta(o^{(n)}, [v_i]_1^n)$	κ	[ck-δ]
v	$\texttt{If}\ N\ L\ \kappa$		$\begin{cases} N & \text{si } v = \texttt{true} \\ L & \text{si } v = \texttt{false} \end{cases}$	κ	[ck-frk]
$\lambda f.M$	$\texttt{Fix}\ \kappa$		$M[f \leftarrow \texttt{Fix}\ \lambda f.M]$	κ	[ck-fix]
$(M\ N)$	κ		M	$\texttt{Ar}\ N\ \kappa$	[ck-ap]
$o^{(n)}[M_i]$	κ		M_1	$\texttt{Op}\ []\ o^{(n)}\ [M_i]_2^n\ \kappa$	[ck-op]
$\texttt{if}\ M\ N\ L$	κ		M	$\texttt{If}\ \ N\ L\ \kappa$	[ck-if]
$\texttt{Fix}\ M$	κ		M	$\texttt{Fix}\ \kappa$	[ck-fix-b]
$\texttt{let}\ x = N\ \texttt{in}\ M$	κ		$(\lambda x.M)\ N$	κ	[ck-let]
v	$\texttt{Mt}$		*estado final*		[ck-fin]

A continuación mostramos la equivalencia entre los estados de la máquina CC y la máquina CK usando la función Φ y su inversa.

Teorema 7.5.1 ($\overset{*}{\underset{\text{CC}}{\mapsto}} \Leftrightarrow \overset{*}{\underset{\text{CK}}{\mapsto}}$ [15]). *Si $\kappa \approx C$ y $\kappa' \approx C'$, entonces se tiene que*

$$\langle M, C\rangle \overset{*}{\underset{CC}{\mapsto}} \langle N, C'\rangle \iff \langle M, \kappa\rangle \overset{*}{\underset{CK}{\mapsto}} \langle N, \kappa'\rangle$$

Demostración. Probaremos cada sentido de la equivalencia mediante inducción sobre la longitud de la derivación

- [$\Rightarrow$] Supongamos que $\langle M, C\rangle \overset{*}{\underset{\text{CC}}{\mapsto}} \langle N, C'\rangle$

  - Si $\langle M, C\rangle \overset{0}{\underset{\text{CC}}{\mapsto}} \langle N, C'\rangle$, entonces $M = N$ y $C = C'$; por lo tanto si $\kappa \approx C$ y $C' \approx \kappa'$ Concluimos que $\langle M, \kappa\rangle \overset{0}{\underset{\text{CK}}{\mapsto}} \langle M, \kappa'\rangle$
  - Tomemos por hipótesis de inducción que si $\langle M, C\rangle \overset{n}{\underset{\text{CC}}{\mapsto}} \langle N, C'\rangle$, entonces $\langle M, \kappa\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle N, \kappa'\rangle$, donde $\kappa \approx C$ y $\kappa' \approx C'$.

    Supongamos que $\langle M, C\rangle \overset{n+1}{\underset{\text{CC}}{\mapsto}} \langle N, C'\rangle$; por lo tanto existen M_1, C_1 tales que

    $$\langle M, C\rangle \underset{\text{CC}}{\mapsto} \langle M_1, C_1\rangle \overset{n}{\underset{\text{CC}}{\mapsto}} \langle N, C'\rangle$$

    Si tomamos $C \approx \kappa$ y $C_1 \approx \kappa_1$ entonces deberemos mostrar que

    $$\langle M, \kappa\rangle \underset{\text{CK}}{\mapsto} \langle M_1, \kappa_1\rangle$$

    para lo cual basta observar la definición $\underset{\text{CC}}{\mapsto}$ y $\underset{\text{CK}}{\mapsto}$ para notar que cada que $\langle M, C\rangle \underset{\text{CC}}{\mapsto} \langle N, C'\rangle$ se tiene que $\langle M, \Phi(C)\rangle \underset{\text{CK}}{\mapsto} \langle N, \Phi(C')\rangle$

Por hipótesis de inducción $\langle M_1, \kappa_1\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle N, \kappa'\rangle$. Por lo tanto tenemos que

$$\langle M, \kappa\rangle \underset{\text{CK}}{\mapsto} \langle M_1, \kappa_1\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle N, \kappa'\rangle$$

- [$\Leftarrow$] Análogo al caso anterior.

□

Ya que los estados de la máquina CC son equivalentes a los de la máquina CK, entonces podemos asegurar que la máquina CK modela de manera fiel las reglas de la semántica operacional por lo que podríamos usar esta máquina para implementar un evaluador de Λ_p. Sin embargo como veremos en la siguiente sección, aun hay ciertas ineficiencias en esta máquina.

Corolario 7.5.1. *La máquina CK y la máquina CC son equivalentes, es decir que* $\langle M, \texttt{Mt}\rangle \overset{*}{\underset{CK}{\mapsto}} \langle v, \texttt{Mt}\rangle$ *si y sólo si* $\langle M, \square\rangle \overset{*}{\underset{CC}{\mapsto}} \langle v, \square\rangle$

Corolario 7.5.2. *La máquina CK modela las reglas de la semántica operacional, es decir que* $M \overset{*}{\rightarrow}_{\Lambda} v$ *si y sólo si* $\langle M, \texttt{Mt}\rangle \overset{*}{\underset{CK}{\mapsto}} \langle v, \texttt{Mt}\rangle$

7.6. Cerraduras y la máquina CEK

Con la máquina CK evitamos tener que recorrer la estructura de los contextos de evaluación innecesariamente, sin embargo aún tenemos un problema similar en las reglas `[ck-ar]` y `[ck-fix]` donde cada vez que aplicamos una sustitución debemos de recorrer todo el árbol sintáctico de la subexpresión intentando aplicar la sustitución correspondiente. Éste es, en la mayoría de los casos, un esfuerzo inútil ya que solamente una pequeña porción de los nodos requieren de la sustitución; además de que debemos de realizar este recorrido cada vez que la máquina haga una transición `[ck-ar]` o `[ck-fix]`.

Una manera de evitar estos recorridos es postergando la aplicación de las sustituciones de manera perezosa. Esto es que en lugar de aplicar inmediatamente una sustitución sobre toda la subexpresión recordemos que la sustitución ha quedado pendiente y únicamente la aplicaremos cuando sea indispensable, es decir cuando la cadena de control sea únicamente una variable. En tal caso buscaremos cuál era la expresión que debió haber sido sustituida por tal variable y sólo entonces aplicaremos la sustitución correspondiente.

Para lograr recordar las sustituciones que han quedado pendientes consideraremos una nueva estructura conocida como *entorno de evaluación*, a la cual denotaremos mediante ε. Intuitivamente un entorno de evaluación responde a la

consulta «¿con qué expresión debo sustituir esta variable?», aunque en realidad esto es un poco más complicado ya que la respuesta a esta consulta no puede ser únicamente una expresión M pues esta expresión puede contener variables libres que aún esperan ser sustituidas. Lo que necesitamos es que la respuesta a tal consulta sea de la forma $\langle M, \varepsilon' \rangle$, donde M es la expresión que debió haber sido sustituida por la variable que nos interesa, y ε' es el entorno de evaluación correspondiente a M que contiene la información necesaria para realizar las sustituciones pendientes en M. Por lo tanto nuestra nueva máquina no opera sobre una expresión M sino sobre una estructura de la forma $\langle M, \varepsilon \rangle$ a la cual llamaremos *cerradura*. Podemos ver que las definiciones de estas dos estructuras son mutuamente recursivas: una cerradura es una estructura formada por una expresión y un entorno de evaluación, que a su vez está formada por cerraduras; definimos ambas estructuras a continuación.

Definición 62: $\langle M, \varepsilon \rangle$ Cerraduras [15]

Nos referiremos al par $\langle M, \varepsilon \rangle$, donde $M \in \Lambda_p$ y ε es un entorno de evaluación, como una *cerradura*. El entorno ε deberá de contener asociaciones para al menos todas las variables libres de M, las cuales deberán ser mapeadas a una cerradura.

Definición 63: $\varepsilon \mapsto \langle M, \varepsilon \rangle$ Entornos de evaluación [15]

Un entorno de evaluación, denotado ε, es un mapeo de variables a cerraduras que cuenta con las siguientes operaciones:

- $dom(\varepsilon)$ devuelve el conjunto de variables que tienen asociadas una cerradura en ε
- $\varepsilon(x)$ devuelve la cerradura asociada a x, o un error si $x \notin dom(\varepsilon)$
- $\varepsilon_\Lambda(x)$ devuelve solamente la expresión M correspondiente a la cerradura asociada a x
- $\varepsilon_\varepsilon(x)$ devuelve solamente el entorno de ejecución correspondiente a la cerradura asociada a x
- $\varepsilon[x \mapsto \langle M, \varepsilon' \rangle]$ extiende a ε con la nueva asociación $x \mapsto \langle M, \varepsilon' \rangle$
- $\varnothing$ denota al entorno vacío

La máquina que definiremos tendrá estados internos de la forma $\langle \langle M, \varepsilon \rangle, \kappa \rangle$, donde $\langle M, \varepsilon \rangle$ es una cerradura, y κ es una continuación. Sin embargo no podemos usar las continuaciones que definimos anteriormente ya que éstas usan

expresiones en lugar de cerraduras, por lo que tendremos que redefinir nuestras continuaciones de manera que usen cerraduras en lugar de expresiones. Formalmente deberíamos de utilizar las siguientes continuaciones [15]:

$$
\begin{aligned}
K_C \;=\;& \texttt{Mt} \\
\mid\;& \texttt{Ar}\ \langle N,\, \varepsilon\rangle\ \kappa \\
\mid\;& \texttt{Fn}\ \langle v,\, \varepsilon\rangle\ \kappa \\
\mid\;& \texttt{Op}\ [\langle v_i,\, \varepsilon_i\rangle]_{j-1}^{1}\ o^{(n)}\ [\langle M_i,\, \varepsilon_i\rangle]_{j+1}^{n}\ \kappa \\
\mid\;& \texttt{If}\ \langle N,\, \varepsilon\rangle\ \langle L,\, \varepsilon'\rangle\ \kappa \\
\mid\;& \texttt{Fix}\ \kappa
\end{aligned}
$$

Sin embargo podemos observar que para los penúltimos dos casos todas las cerraduras deberían de tener asociadas el mismo entorno, por lo que podemos simplificar la notación de la siguiente manera.

Definición 64: Continuaciones con cerraduras [15]

$$
\begin{aligned}
K_C \;=\;& \texttt{Mt} \\
\mid\;& \texttt{Ar}\ N\ \varepsilon\ \kappa \\
\mid\;& \texttt{Fn}\ v\ \varepsilon\ \kappa \\
\mid\;& \texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \varepsilon\ \kappa \\
\mid\;& \texttt{If}\ N\ L\ \varepsilon\ \kappa \\
\mid\;& \texttt{Fix}\ \kappa
\end{aligned}
$$

Donde $M_i,\ N,\ L \in \Lambda_p \quad v_i \in \mathcal{V} \quad \kappa \in K_C \quad \varepsilon$ es un entorno de evaluación

Observemos también que hasta ahora ninguna de las máquinas que hemos dado era capaz de encontrarse con una variable. Esto es porque todas las expresiones que recibe la máquina como entrada son expresiones cerradas, y al reducir un redex que liga variables aplicábamos inmediatamente la sustitución eliminando así toda ocurrencia de la variable. Al usar entornos de evaluación estamos postergando la aplicación de la sustitución hasta que sea necesaria por lo que en este caso la máquina sí se encontraría con variables libres. Es por esto que debemos agregar una regla adicional a la tabla de transiciones de nuesta máquina basada en cerraduras para que cuando encuentre una variable consulte en el entorno la cerradura correspondiente.

A pesar de que formalmente los estados internos de nuestra máquina deberían ser de la forma $\langle\langle M, \varepsilon\rangle, \kappa\rangle$, simplificaremos la notación y diremos que son de la forma $\langle M, \varepsilon, \kappa\rangle$, pero es importante recordar que en realidad se trata de un par formado por una cerradura y una continuación. En el Apéndice D se encuentra una versión más grande de esta definición.

Definición 65: La máquina CEK [15]

Definimos el comportamiento de la máquina CEK que tiene estados de la forma $\langle M, \varepsilon, \kappa\rangle$ donde $\langle M, \varepsilon\rangle$ es una cerradura y $\kappa \in K_C$. El estado inicial de la máquina es $\langle M, \varnothing, \texttt{Mt}\rangle$.

M	ε	κ	$\overset{}{\underset{\text{CEK}}{\mapsto}}$	M	ε	κ	
x	ε	κ		$\varepsilon_\Lambda(x)$	$\varepsilon_\varepsilon(x)$	κ	[cek-var]
v	ε	$\texttt{Ar}\ N\ \varepsilon'\ \kappa$		N	ε'	$\texttt{Fn}\ v\ \varepsilon\ \kappa$	[cek-fn]
v	ε	$\texttt{Fn}\ (\lambda x.M)\ \varepsilon'\ \kappa$		M	$\varepsilon'[x \mapsto \langle v, \varepsilon\rangle]$	κ	[cek-ar]
v	ε	$\texttt{Op}\ [v_i]_{x-1}^{1}\ o^{(n)}\ []\ \varepsilon'\ \kappa$		$\delta(o^{(n)}, [v_i]_1^n)$	$\varnothing$	κ	[cek-opv]
v	ε	$\texttt{Op}\ [v_i]_{\lambda-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \varepsilon'\ \kappa$		M_{j+1}	ε'	$\texttt{Op}\ [v_i]_j^1\ o^{(n)}\ [M_i]_{j+2}^n\ \varepsilon'\ \kappa$	[cek-δ]
v	ε	$\texttt{If}\ N\ L\ \varepsilon'\ \kappa$		$\begin{cases} N & \text{si } v = \texttt{true} \\ L & \text{si } v = \texttt{false} \end{cases}$	ε'	κ	[cek-frk]
$\lambda f.M$	ε	$\texttt{Fix}\ \kappa$		M	$\varepsilon[f \mapsto \langle \texttt{Fix}\ \lambda f.M, \varepsilon\rangle]$	κ	[cek-fix]
$(M\ N)$	ε	κ		M	ε	$\texttt{Ar}\ N\ \varepsilon\ \kappa$	[cek-ap]
$o^{(n)}[M_i]$	ε	κ		M_1	ε	$\texttt{Op}\ []\ o^{(n)}\ [M_i]_2^n\ \varepsilon\ \kappa$	[cek-op]
$\texttt{if}\ M\ N\ L$	ε	κ		M	ε	$\texttt{If}\ N\ L\ \varepsilon\ \kappa$	[cek-if]
$\texttt{Fix}\ M$	ε	κ		M	ε	$\texttt{Fix}\ \kappa$	[cek-fix-b]
$\texttt{let}\ x = N\ \texttt{in}\ M$	ε	κ		$(\lambda x.M)\ N$	ε	κ	[cek-let]
v	ε	$\texttt{Mt}$		*estado final*			[cek-fin]

Veamos mediante un ejemplo sencillo cómo funciona la evaluación de una expresión de Λ_p usando la máquina CEK. Supongamos que queremos evaluar la expresión $(\lambda y.\lambda x.y)\ 7$. Empezaremos transformando la expresión a un estado inicial, es decir $\langle (\lambda y.\lambda x.y)\ 7, \varnothing, \texttt{Mt}\ \rangle$ y apliquemos las transiciones correspondientes.

$$\begin{aligned}
\langle (\lambda y.\lambda x.y)\ 7, \varnothing, \texttt{Mt}\rangle &\underset{\text{CEK}}{\mapsto} \langle \lambda y.\lambda x.y, \varnothing, \texttt{Ar}\ 7\ \varnothing\ \texttt{Mt}\rangle && \texttt{[cek-ap]} \\
&\underset{\text{CEK}}{\mapsto} \langle 7, \varnothing, \texttt{Fn}\ (\lambda y.\lambda x.y)\ \varnothing\ \texttt{Mt}\rangle && \texttt{[cek-fn]} \\
&\underset{\text{CEK}}{\mapsto} \langle \lambda x.y, \{y \mapsto \langle 7, \varnothing\rangle\}, \texttt{Mt}\rangle && \texttt{[cek-ar]}
\end{aligned}$$

Ahora tenemos que hemos llegado al estado final $\langle \lambda x.y, \{y \mapsto \langle 7, \varnothing\rangle\}, \texttt{Mt}\ \rangle$, sin embargo el resultado que esperaríamos de evaluar tal expresión es $\lambda y.7$. Para obtener el resultado real a partir del estado final al que hemos llegado es necesario *descargar* el estado final de la máquina lo cual involucra realizar todas las sustituciones que han quedado pendientes. Para lograr esto definimos una función que toma una cerradura y realiza las sustituciones pendientes para devolver una expresión.

Definición 66: Ψ Conversión de cerraduras a expresiones [15]

Definimos la función Ψ que toma una cerradura $\langle M, \varepsilon\rangle$ y devuelve la expresión que resulta de aplicar sobre cada variable libre de M la sustitución indicada en ε.

$$\begin{aligned}\Psi(\langle M, \varnothing\rangle) &= M\\ \Psi(\langle M, \varepsilon\rangle) &= M[x_i \leftarrow \Psi(\varepsilon(x_i))]\end{aligned}$$

$$\text{para } 1 \leq i \leq n \text{ donde } \{x_1, \ldots, x_n\} = \mathcal{V}_{\mathcal{L}}(M)$$

Si usamos esta función sobre la cerradura contenida en nuestro estado final entonces llegamos al resultado que esperábamos.

$$\begin{aligned}\Psi(\langle \lambda x.y, \{y \mapsto \langle 7, \varnothing\rangle\}\rangle) &= \lambda x.y[y \leftarrow \Psi(\langle 7, \varnothing\rangle)]\\ &= \lambda x.y[y \leftarrow 7]\\ &= \lambda x.7\end{aligned}$$

La función Ψ será indispensable para realizar una implementación de la máquina CEK como veremos en el próximo capítulo, pero para propósitos de la demostración del teorema que relaciona a la máquina CEK con la máquina CK también definiremos una función que transforma continuaciones con cerraduras a continuaciones sencillas.

Definición 67: Υ Conversión de K_C a K [15]

$$\begin{aligned}\Upsilon(\texttt{Mt}) &= \texttt{Mt}\\ \Upsilon(\texttt{Ar}\ N\ \varepsilon\ \kappa) &= \texttt{Ar}\ \Psi(\langle N, \varepsilon\rangle)\ \ \Upsilon(\kappa)\\ \Upsilon(\texttt{Fn}\ v\ \varepsilon\ \kappa) &= \texttt{Fn}\ \Psi(\langle v, \varepsilon\rangle)\ \ \Upsilon(\kappa)\\ \Upsilon(\texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \varepsilon\ \kappa) &= \texttt{Op}\ [\Psi(\langle v_i, \varepsilon\rangle)]_{j-1}^{1}\ o^{(n)}\ [\Psi(\langle M_i, \varepsilon\rangle)]_{j+1}^{n}\ \Upsilon(\kappa)\\ \Upsilon(\texttt{If}\ N\ L\ \varepsilon\ \kappa) &= \texttt{If}\ \Psi(\langle N, \varepsilon\rangle)\ \ \Psi(\langle L, \varepsilon\rangle)\ \ \Upsilon(\kappa)\\ \Upsilon(\texttt{Fix}\ \kappa) &= \texttt{Fix}\ \ \Upsilon(\kappa)\end{aligned}$$

Observemos que la función Ψ no es inyectiva pues podemos dar entornos ε y ε' distintos tales que $\Psi(\langle M, \varepsilon\rangle) = \Psi(\langle M, \varepsilon'\rangle)$ por lo que no podemos invertir Ψ ni Υ. Para probar que la máquina CEK y CK son equivalentes habrá que tomar en cuenta que la máquina CEK tiene transiciones adicionales cuando usa la regla `[cek-var]`.

Teorema 7.6.1 ($\overset{*}{\underset{\text{CEK}}{\mapsto}} \Rightarrow \overset{*}{\underset{\text{CK}}{\mapsto}}$ [15])**.** *Si* $\langle M, \varepsilon, \bar{\kappa}\rangle \overset{*}{\underset{CEK}{\mapsto}} \langle v', \varepsilon', \mathtt{Mt}\rangle$*, entonces* $\langle N, \kappa\rangle \overset{*}{\underset{CK}{\mapsto}} \langle v, \mathtt{Mt}\rangle$ *donde* $N = \Psi(\langle M, \varepsilon\rangle)$*,* $\kappa = \Upsilon(\bar{\kappa})$ *y* $v = \Psi(\langle v', \varepsilon'\rangle)$

Demostración. Inducción sobre $\overset{n}{\underset{\text{CEK}}{\mapsto}}$

- Si $\langle M, \varepsilon, \bar{\kappa}\rangle \overset{0}{\underset{\text{CEK}}{\mapsto}} \langle v', \varepsilon', \mathtt{Mt}\rangle$ con $N = \Psi(\langle M, \varepsilon\rangle)$, entonces $M = v'$, $\varepsilon = \varepsilon'$ y $\bar{\kappa} = \mathtt{Mt}$; y por lo tanto $N = v = \Psi(\langle v', \varepsilon'\rangle)$, por lo que $\langle N, \mathtt{Mt}\rangle \overset{0}{\underset{\text{CK}}{\mapsto}} \langle v, \mathtt{Mt}\rangle$

Supongamos que si $\langle M, \varepsilon, \bar{\kappa}\rangle \overset{n}{\underset{\text{CEK}}{\mapsto}} \langle v', \varepsilon', \mathtt{Mt}\rangle$ entonces $\langle N, \kappa\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle v, \mathtt{Mt}\rangle$ donde $N = \Psi(\langle M, \varepsilon\rangle)$, $\kappa = \Upsilon(\bar{\kappa})$ y $v = \Psi(\langle v', \varepsilon'\rangle)$.

Supongamos que $\langle M_0, \varepsilon_0, \bar{\kappa}_0\rangle$ es un estado tal que

$$\langle M_0, \varepsilon_0, \bar{\kappa}_0\rangle \underset{\text{CEK}}{\mapsto} \langle M, \varepsilon, \bar{\kappa}\rangle \overset{n}{\underset{\text{CEK}}{\mapsto}} \langle v', \varepsilon', \mathtt{Mt}\rangle$$

Por hipótesis de inducción sobre $\langle M, \varepsilon, \bar{\kappa}\rangle \overset{n}{\underset{\text{CEK}}{\mapsto}} \langle v', \varepsilon', \mathtt{Mt}\rangle$, existe un estado $\langle N, \kappa\rangle$ donde $N = \Psi(\langle M, \varepsilon\rangle)$ y $\kappa = \Upsilon(\bar{\kappa})$ tal que $\langle N, \kappa\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle v, \mathtt{Mt}\rangle$ donde $v = \Psi(\langle v', \varepsilon'\rangle)$.

Sea $\langle N_0, \kappa_0\rangle$ un estado tal que $N_0 = \Psi(\langle M_0, \varepsilon_0\rangle)$ y $\kappa_0 = \Upsilon(\bar{\kappa}_0)$.

Para terminar la prueba debemos mostrar que $\langle N_0, \kappa_0\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle N, \kappa\rangle$.

Verificando las definiciones de $\underset{\text{CEK}}{\mapsto}$ y $\underset{\text{CK}}{\mapsto}$ podemos ver que siempre que $\langle M, \varepsilon, \bar{\kappa}\rangle \underset{\text{CEK}}{\mapsto} \langle M', \varepsilon', \bar{\kappa}'\rangle$ también ocurre que $\langle \Psi(\langle M, \varepsilon\rangle), \Upsilon(\bar{\kappa})\rangle \underset{\text{CK}}{\mapsto} \langle \Psi(\langle M', \varepsilon'\rangle), \Upsilon(\bar{\kappa}')\rangle$ a excepción de cuando la máquina CEK hace una transición `[cek-var]`, en cuyo caso tenemos entonces que $\langle \Psi(\langle M, \varepsilon\rangle), \Upsilon(\bar{\kappa})\rangle \overset{0}{\underset{\text{CK}}{\mapsto}} \langle \Psi(\langle M', \varepsilon'\rangle), \Upsilon(\bar{\kappa}')\rangle$.

Con esta observación podemos concluir que

$$\langle N_0, \kappa_0\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle N, \kappa\rangle \overset{*}{\underset{\text{CK}}{\mapsto}} \langle v, \mathtt{Mt}\rangle$$

□

Teorema 7.6.2 ($\overset{*}{\underset{\text{CK}}{\mapsto}} \Rightarrow \overset{*}{\underset{\text{CEK}}{\mapsto}}$ [15])**.** *Sean* $\langle M, \kappa\rangle$ *y* $\langle N, \varepsilon, \bar{\kappa}\rangle$ *estados de la máquina CK y CEK respectivamente donde* $M = \Psi(\langle N, \varepsilon\rangle)$ *y* $\kappa = \Upsilon(\bar{\kappa})$*.*

Si $\langle M, \kappa\rangle \overset{*}{\underset{CK}{\mapsto}} \langle v, \texttt{Mt}\rangle$*, entonces existe alguna cerradura* $\langle N', \varepsilon'\rangle$ *para la cual* $v = \Psi(\langle N', \varepsilon'\rangle)$ *tal que*

$$\langle N, \varepsilon, \bar{\kappa}\rangle \overset{*}{\underset{CEK}{\mapsto}} \langle N', \varepsilon', \texttt{Mt}\ \rangle$$

Demostración. Inducción sobre $\overset{n}{\underset{\text{CK}}{\mapsto}}$

- Si $\langle M, \kappa\rangle \overset{0}{\underset{\text{CK}}{\mapsto}} \langle v, \texttt{Mt}\ \rangle$, entonces $\kappa = \texttt{Mt} = \bar{\kappa}$ y $v = M = \Psi(\langle N, \varepsilon\rangle)$; y sucede una de las siguientes
  - Si $N = x$ y $[x \mapsto \langle v, \varepsilon_x\rangle] \in \varepsilon$, entonces $\langle x, \varepsilon, \texttt{Mt}\rangle \underset{\text{CEK}}{\mapsto} \langle v, \varepsilon_x, \texttt{Mt}\rangle$
  - Si $N = v$, entonces $\langle N, \varepsilon, \texttt{Mt}\rangle \overset{0}{\underset{\text{CEK}}{\mapsto}} \langle v, \varepsilon, \texttt{Mt}\rangle$

Supongamos que si $\langle M, \kappa\rangle \overset{n}{\underset{\text{CK}}{\mapsto}} \langle v, \texttt{Mt}\rangle$, entonces $\langle N, \varepsilon, \bar{\kappa}\rangle \overset{*}{\underset{\text{CEK}}{\mapsto}} \langle N', \varepsilon', \texttt{Mt}\rangle$, donde $M = \Psi(\langle N, \varepsilon\rangle)$, $\kappa = \Upsilon(\bar{\kappa})$ y $v = \Psi(\langle N', \varepsilon'\rangle)$.

Supongamos también que $\langle M_0, \kappa_0\rangle \underset{\text{CK}}{\mapsto} \langle M, \kappa\rangle \overset{n}{\underset{\text{CK}}{\mapsto}} \langle v, \texttt{Mt}\rangle$, donde $M_0 = \Psi(\langle N_0, \varepsilon_0\rangle)$, $\kappa_0 = \Upsilon(\bar{\kappa}_0)$ para algún estado $\langle N_0, \varepsilon_0, \bar{\kappa}_0\rangle$ de la máquina CEK.

Primero debemos mostrar que $\langle N_0, \varepsilon_0, \bar{\kappa}_0\rangle \overset{*}{\underset{\text{CEK}}{\mapsto}} \langle N, \varepsilon, \kappa\rangle$ tal que $M = \Psi(\langle N, \varepsilon\rangle)$ y $\kappa = \Upsilon(\bar{\kappa})$

Analicemos las posibles formas de la transición $\langle M_0, \kappa_0\rangle \underset{\text{CK}}{\mapsto} \langle M, \kappa\rangle$

- `[ck-fn]` $\langle M_0, \kappa_0\rangle \underset{\text{CK}}{\mapsto} \langle M, \kappa\rangle = \langle v, \texttt{Ar}\ L\ \kappa_0'\rangle \underset{\text{CK}}{\mapsto} \langle L, \texttt{Fn}\ v\ \kappa_0'\rangle$

  Considerando que $v = M_0 = \Psi(\langle N_0, \varepsilon_0\rangle)$, entonces tenemos dos posibles casos:

  - Si $N_0 = v' \notin \mathbb{X}$, entonces $\langle N_0, \varepsilon_0, \bar{\kappa}_0\rangle = \langle v', \varepsilon_0, \texttt{Ar}\ L'\ \varepsilon_0'\ \bar{\kappa}_0'\rangle$

    $$\langle v', \varepsilon_0, \texttt{Ar}\ L'\ \varepsilon_0'\ \bar{\kappa}_0'\rangle \underset{\text{CEK}}{\mapsto} \langle L', \varepsilon_0', \texttt{Fn}\ v'\ \varepsilon_0\ \bar{\kappa}_0'\rangle$$

    De la suposición de que

    $$\texttt{Ar}\ L\ \kappa_0' = \Upsilon(\texttt{Ar}\ L'\ \varepsilon_0'\ \bar{\kappa}_0')$$

    Tenemos que $L = \Psi(\langle L', \varepsilon_0'\rangle)$ por lo tanto tenemos que

- $M = L = \Psi(\langle L', \varepsilon_0' \rangle)$

También sabemos que $\kappa' = \Upsilon(\bar{\kappa}')$, y como $v = \Psi(\langle v', \varepsilon_0 \rangle)$ entonces tenemos que

- $\kappa = \texttt{Fn}\ v\ \kappa_0' = \Upsilon(\texttt{Fn}\ v'\ \varepsilon_0\ \bar{\kappa}_0')$

- Si $N_0 = y \in \mathbb{X}$, entonces $v = \Psi(\langle y, \varepsilon_0 \rangle)$.

$$\begin{aligned}\langle N_0, \varepsilon_0, \bar{\kappa}_0 \rangle &= \langle y, \varepsilon_0, \texttt{Ar}\ L'\ \varepsilon_0'\ \bar{\kappa}_0' \rangle \\ &\underset{\text{CEK}}{\mapsto} \langle v', \varepsilon_y, \texttt{Ar}\ L'\ \varepsilon_0'\ \bar{\kappa}_0' \rangle \\ &\underset{\text{CEK}}{\mapsto} \langle L', \varepsilon_0', \texttt{Fn}\ v'\ \varepsilon_y\ \bar{\kappa}_0' \rangle\end{aligned}$$

Analizando las reglas `[cek-ar]` y `[cek-fix]` podemos ver que $\varepsilon_0 = \varepsilon_y$, por lo que $v = \Psi(\langle v', \varepsilon_y \rangle)$ y podemos usar el caso anterior para concluir lo mismo.

- `[ck-ar]` $\langle M_0, \kappa_0 \rangle \underset{\text{CK}}{\mapsto} \langle M, \kappa \rangle = \langle v, \texttt{Fn}\ \lambda x.L\ \kappa_0' \rangle \underset{\text{CK}}{\mapsto} \langle L[x \leftarrow v], \kappa_0' \rangle$

Considerando que $v = M_0 = \Psi(\langle N_0, \varepsilon_0 \rangle)$, entonces tenemos dos posibles casos:

- Si $N_0 = v' \notin \mathbb{X}$, entonces $\langle N_0, \varepsilon_0, \bar{\kappa}_0 \rangle = \langle v', \varepsilon_0, \texttt{Fn}\ \lambda x.L'\ \varepsilon_0'\ \bar{\kappa}_0' \rangle$

$$\langle v', \varepsilon_0, \texttt{Fn}\ \lambda x.L'\ \varepsilon_0'\ \bar{\kappa}_0' \rangle \underset{\text{CEK}}{\mapsto} \langle L', \varepsilon_0'[x \mapsto \langle v', \varepsilon_0 \rangle], \bar{\kappa}_0' \rangle$$

  - Veamos que $L[x \leftarrow v] = \Psi(\langle L', \varepsilon_0'[x \mapsto \langle v', \varepsilon_0 \rangle] \rangle)$.
  Esto es cierto porque tenemos que

$$\texttt{Fn}\ \lambda x.L\ \kappa_0' = \Upsilon(\texttt{Fn}\ \lambda x.L'\ \varepsilon_0'\ \bar{\kappa}_0')$$

  por lo que $\lambda x.L = \Psi(\langle \lambda x.L', \varepsilon_0' \rangle)$ y por lo tanto $L = \Psi(\langle L', \varepsilon_0' \rangle)$. Ahora considerando que $\{x_1, \ldots, x_n\} = \mathcal{V}_{\mathcal{L}}(L')$ obtenemos lo siguiente

$$\begin{aligned}\Psi(\langle L', \varepsilon_0'[x \mapsto \langle v', \varepsilon_0 \rangle] \rangle) &= L'[x_i \leftarrow \Psi(\varepsilon_0'(x_i))]\ [x \leftarrow \Psi(\langle v', \varepsilon_0 \rangle)] \\ &= \Psi(\langle L', \varepsilon_0' \rangle)\ [x \leftarrow \Psi(\langle v', \varepsilon_0 \rangle)] \\ &= L\ [x \leftarrow \Psi(\langle v', \varepsilon_0 \rangle)] \\ &= L\ [x \leftarrow v]\end{aligned}$$

  - También tenemos que $\kappa_0' = \Upsilon(\bar{\kappa}_0')$ porque $\texttt{Fn}\ \lambda x.L\ \kappa_0' = \Upsilon(\texttt{Fn}\ \lambda x.L'\ \varepsilon_0'\ \bar{\kappa}_0')$

- Si $N_0 = y \in \mathbb{X}$, entonces $v = \Psi(\langle y, \varepsilon_0 \rangle)$

$$\begin{aligned}\langle N_0, \varepsilon_0, \bar{\kappa}_0 \rangle &= \langle y, \varepsilon_0, \texttt{Fn}\ \lambda x.L'\ \varepsilon_0'\ \bar{\kappa}_0' \rangle \\ &\underset{\text{CEK}}{\mapsto} \langle v', \varepsilon_y, \texttt{Fn}\ \lambda x.L'\ \varepsilon_0'\ \bar{\kappa}_0' \rangle \\ &\underset{\text{CEK}}{\mapsto} \langle L', \varepsilon_0'[x \mapsto \langle v', \varepsilon_y \rangle], \bar{\kappa}_0' \rangle\end{aligned}$$

Analizando las reglas `[cek-ar]` y `[cek-fix]` podemos ver que $\varepsilon_0 = \varepsilon_y$, por lo que $v = \Psi(\langle v', \varepsilon_y \rangle)$ y podemos usar el caso anterior para concluir lo mismo.

- Los casos para las reglas faltantes son análogos a los dos casos anteriores.

Ya que hemos mostrado que siempre que $\langle M_0, \kappa_0\rangle \underset{\text{CK}}{\mapsto} \langle M, \kappa\rangle$ existen estados de la máquina CEK tales que $\langle N_0, \varepsilon_0, \bar{\kappa}_0\rangle \underset{\text{CEK}}{\overset{*}{\mapsto}} \langle N, \varepsilon, \kappa\rangle$ donde

- $M_0 = \Psi(\langle N_0, \varepsilon_0\rangle)$
- $\kappa_0 = \Upsilon(\bar{\kappa}_0)$
- $M = \Psi(\langle N, \varepsilon\rangle)$
- $\kappa = \Upsilon(\bar{\kappa})$

podemos usar la hipótesis de inducción sobre $\langle M, \kappa\rangle \underset{\text{CK}}{\overset{n_1}{\mapsto}} \langle v, \texttt{Mt}\rangle$ para concluir que existe un estado CEK $\langle N', \varepsilon', \texttt{Mt}\rangle$ tal que $\langle N, \varepsilon, \kappa\rangle \underset{\text{CEK}}{\overset{*}{\mapsto}} \langle N', \varepsilon', \texttt{Mt}\rangle$ donde $v = \Psi(\langle N', \varepsilon'\rangle)$. Combinado con el resultado anterior conseguimos que

$$\langle N_0, \varepsilon_0, \bar{\kappa}_0\rangle \underset{\text{CEK}}{\overset{*}{\mapsto}} \langle N', \varepsilon', \texttt{Mt}\rangle$$

□

Corolario 7.6.1. *Si* $\langle M, \varnothing, \texttt{Mt}\,\rangle \underset{CEK}{\overset{*}{\mapsto}} \langle v, \varepsilon, \texttt{Mt}\rangle$*, entonces* $\langle M, \texttt{Mt}\rangle \underset{CK}{\overset{*}{\mapsto}} \langle \Psi(\langle v, \varepsilon\rangle), \texttt{Mt}\rangle$

Corolario 7.6.2. *Si* $\langle M, \texttt{Mt}\rangle \underset{CK}{\overset{*}{\mapsto}} \langle v, \texttt{Mt}\rangle$*, entonces* $\langle M, \varnothing, \texttt{Mt}\rangle \underset{CEK}{\overset{*}{\mapsto}} \langle v, \varepsilon, \texttt{Mt}\rangle$*.*

Corolario 7.6.3 ($\underset{\text{CK}}{\overset{*}{\mapsto}} \Leftrightarrow \underset{\text{CEK}}{\overset{*}{\mapsto}}$)**.** *La máquina CK y la máquina CEK son equivalentes, es decir que* $\langle M, \texttt{Mt}\rangle \underset{CK}{\overset{*}{\mapsto}} \langle v, \texttt{Mt}\rangle$ *si y sólo si* $\langle M, \varnothing, \texttt{Mt}\rangle \underset{CEK}{\overset{*}{\mapsto}} \langle v', \varepsilon', \texttt{Mt}\rangle$ *donde* $v = \Psi(\langle v', \varepsilon'\rangle)$*.*

Concluimos este capítulo mostrando que la máquina CEK modela las reglas de la semántica operacional, y por lo tanto la podemos usar para implementar de manera eficiente un evaluador de Λ_p como se muestra en el capítulo siguiente.

Corolario 7.6.4 ($\overset{*}{\to}_{\Lambda} \Leftrightarrow \underset{\text{CEK}}{\overset{*}{\mapsto}}$)**.** *La máquina CEK modela las reglas de la semántica operacional, es decir que* $M \overset{*}{\to}_{\Lambda} v$ *si y sólo si* $\langle M, \varnothing, \texttt{Mt}\rangle \underset{CEK}{\overset{*}{\mapsto}} \langle v', \varepsilon', \texttt{Mt}\rangle$ *donde* $v = \Psi(\langle v', \varepsilon'\rangle)$*.*

Capítulo 8

Implementación en Haskell

En este capítulo codificaremos las definiciones de Λ_p, su sistema de tipos polimórfico, y la máquina CEK usando el lenguaje de programación Haskell [24]. Una de las razones por la cual se eligió este lenguaje para realizar la implementación es que dado que Haskell es un lenguaje de programación funcional puro, las definiciones que hemos dado se traducen de manera casi directa a código; por lo que debería ser sencillo mostrar que la implementación es correcta.

Dado que una introducción a Haskell queda fuera del alcance de este trabajo, se presupone que el lector ya está familiarizado con el lenguaje; y en caso de no ser así se le refiere a siguiente bibliografía introductoria [23, 31].

El código de Haskell está divido en tres módulos que comprenden respectivamente al cálculo de Λ_p, su sistema de tipos polimórfico y a la máquina CEK para su evaluación.

8.1. El cálculo de Λ_p

```
module Iswim where
```

8.1.1. Las constantes primitivas $\mathcal{C}_\mathcal{P}$ (Def. 16)

```
data Cte = Nat Int      -- Naturales
         | TF Bool      -- Booleanos
           deriving (Eq, Ord)
```

8.1.2. Los operadores primitivos $\mathcal{O}_\mathcal{P}$ (Def. 17)

```
data Op = IsZero  -- dice si n es cero
        | Sum     -- n + m
        | Prod    -- n * m
        | Pred    -- predecesor de n
        | Not     -- negación booleana
          deriving (Eq, Show, Ord)
```

Será necesario expandir estas dos últimas definiciones si se deben agregar más constantes y operadores primitivos al lenguaje.

8.1.3. El cálculo de Λ_p (Def. 77)

```
data Iswim = IVar Ident                  -- Variables
           | C    Cte                   -- Constantes
           | Lam  Ident Iswim           -- λx.M
           | App  Iswim Iswim           -- (M N) aplicación
           | Op   Op [Iswim]            -- Operadores primitivos con [operandos]
           | If   Iswim Iswim Iswim     -- If M then N else L
           | Fix  Iswim                 -- Punto fijo para recursión
           | Let  Ident Iswim Iswim     -- let x = N in M
             deriving (Eq, Ord)
```

Donde `Ident` es el tipo de las variables $\mathbb{X}$ (Def. 1)

```
type Ident = String
```

8.2. El sistema de tipos polimórfico

```
module HM where

import Iswim

import Data.List
import Data.Maybe
import Control.Arrow ( (***) )
import Control.Monad
import Control.Monad.State
import Control.Monad.Except
import qualified Data.Set as S
import qualified Data.Map.Strict as M
```

8.2.1. Algunas definiciones básicas

Las variables de tipo $\mathcal{T}_\mathcal{V}$ (Def. 28)

```
newtype TVar = TVar Ident deriving(Eq, Ord)
```

Los tipos $\mathcal{T}$ (Def. 29)

```
data Type = CType Ident      -- tipos concretos e.g. Nat, Bool
          | Var   TVar       -- variables de tipo
          | Func  Type  Type -- tipo función
          deriving (Eq, Ord)
```

Los esquemas de tipos $\mathcal{E}$ (Def. 45)

```
data TScheme = Forall [TVar] Type deriving(Eq,Show,Ord)
```

La codificación para esquemas de tipo $\mathcal{E}$ usa un constructor `Forall`, representando al cuantificador $\forall$, seguido de una lista de variables de tipo con la cual representamos a las variables ligadas de $\mathcal{E}$. Por ejemplo un esquema de tipos $\forall$`AB.A` $\rightarrow$ `B` se representaría mediante la expresión

```
Forall [a, b] (Func (Var a) (Var b))
  where a = TVar "a"
        b = TVar "b"
```

Los contextos de tipado generalizados $\bar{\Gamma}$ (Def. 46)

```
newtype Gamma = G (M.Map Ident TScheme) deriving (Eq, Show, Ord)
```

En Haskell `Data.Map` es una estructura de datos que implementa eficientemente un *mapeo* de llaves a valores, y que permite realizar consultas e inserciones en tiempo logarítmico. Usamos esta estructura para crear un mapeo de variables a esquemas de tipo, de manera que dada una variable podamos obtener rápidamente su esquema de tipos asociado. Los detalles de esta implementación pueden ser consultados en [28, 2] y el API de esta estructura puede ser consultada en [22].

Las restricciones de tipo $\mathcal{R}$ (Def. 33)

```
newtype Constraint = R [(Type, Type)] deriving (Eq, Ord)
```

Una restricción $T = S$ la representamos mediante una tupla (T, S), por lo que el conjunto de restricciones $\mathcal{R}$ lo representamos mediante una lista de estas tuplas envuelta con el constructor `R`.

La sustitución de tipos σ (Def. 31)

```
newtype Substitution = S (M.Map TVar Type)  deriving (Eq, Ord)
```

Representamos una sustitución σ mediante un mapeo de variables de tipo a tipos.

8.2.2. Aplicando sustituciones de tipo

Formalmente una sustitución de tipo σ solamente se puede aplicar sobre tipos $T \in \mathcal{T}$, sin embargo por practicidad quisiéramos poder aplicarla también sobre restricciones $\mathcal{R}$, esquemas de tipo $\mathcal{E}$ y sobre contextos de tipado generalizados $\bar{\Gamma}$. Para esto definiremos una nueva clase de Haskell que permita aplicar una sustitución σ sobre los tipos de datos pertinentes.

```
class CanSubstitute a where
    subst :: Substitution -> a -> a
```

Ahora crearemos instancias de esta clase para cada una de las estructuras antes mencionadas.

Sobre tipos $\sigma(T)$ (Def. 31)

```
instance CanSubstitute Type  where
  subst _ ct@CType{} = ct
  subst s (Func x y) = Func (subst s x) (subst s y)
  subst s (Var tv)   = query tv s
    where
    query :: TVar -> Substitution -> Type
    query tv' (S s')
        | M.member tv' s' = fromJust $ M.lookup tv' s'
        | otherwise = Var tv'
```

Sobre restricciones $\sigma(\mathcal{R})$ (Def. 36)

```
instance CanSubstitute Constraint where
  subst s (R c) =  R $ map  (subst s *** subst s) c
```

Sobre esquemas de tipo $\sigma(\mathcal{E})$ (Def. 49)

Para definir $\sigma(\mathcal{E})$ como en la Definición 49 necesitaremos definir una nueva sustitución $\sigma_{\neg\vec{\mathtt{A}}}$ la cual no contenga asociaciones $\mathtt{A}_i \mapsto T_i$ para las variables ligadas en el esquema de tipos $\forall \mathtt{A}_1 \ldots \mathtt{A}_n.T$. Para lograr esto definiremos un mapeo `d` que contenga asociaciones artificiales $\{\mathtt{A}_i \mapsto \mathtt{Bool}\}$ para las variables cuantificadas del esquema $\mathcal{E}$ y así aplicar la sustitución cuyo mapeo es la diferencia entre el mapeo de σ y `d`.

```
instance CanSubstitute TScheme where
  subst (S s) (Forall tvs t) = Forall tvs (subst s' t)
    where s' = S $ M.difference d s
          d  = M.fromList $ map (\tv -> (tv, tBool)) tvs
```

Sobre contextos de tipado $\sigma(\bar{\Gamma})$ (Def. 50)

```
instance CanSubstitute Gamma where
  subst s (G g) = G $ M.map (subst s) g
```

La composicíon de sustituciones $\sigma_2 \circ \sigma_1$ (Def. 32)

```
subComp :: Substitution -> Substitution -> Substitution
subComp (S s2) (S s1) = S $ M.map (subst $ S s2) (M.union s1 s2)
-- si hay elementos duplicados en la unión se mantienen los de s1
```

8.2.3. Obteniendo las variables de tipo

Más adelante necesitaremos calcular el conjunto de variables de tipo contenidas en distintas estructuras. Por esta razón definimos la clase `WithTVars` la cual define las funciones `getTVars` y `getTVarsLs` que devuelven las variables de tipo como conjunto y como lista respectivamente.

```
class WithTVars a where
    getTVarsLs :: a -> [Ident]
    getTVars   :: a -> S.Set Ident
    getTVars   = S.fromList . getTVarsLs

instance WithTVars TVar where
    getTVarsLs (TVar v) = [v]

instance WithTVars Constraint where
    getTVarsLs (R c) = concatMap getTVarsLs tvs
        where tvs = foldr (\(a,b) xs -> a:b:xs) [] c

instance WithTVars Gamma where
    getTVarsLs (G g) = M.keys g ++ concatMap getTVarsLs (M.elems g)

instance WithTVars Type where
    getTVarsLs CType{}        = []
    getTVarsLs (Var (TVar x)) = [x]
    getTVarsLs (Func a b)     = getTVarsLs a ++ getTVarsLs b

instance WithTVars Iswim where
    getTVarsLs (IVar _)    = []
    getTVarsLs (C _)       = []
    getTVarsLs (Op _ xs)   = concatMap getTVarsLs xs
    getTVarsLs (Fix e)     = getTVarsLs e
    getTVarsLs (App a b)   = getTVarsLs a ++ getTVarsLs b
    getTVarsLs (Lam _ e)   = getTVarsLs e
    getTVarsLs (Let _ a b) = getTVarsLs a ++ getTVarsLs b
    getTVarsLs (If a b c)  = getTVarsLs a ++ getTVarsLs b ++ getTVarsLs c

instance WithTVars TScheme where
```

```
getTVarsLs (Forall vs t) = concatMap getTVarsLs vs ++ getTVarsLs t
```

8.2.4. El estado $(\mathbb{N}_{\mathbb{V}}, \bar{\Gamma}, \mathcal{R})$

Nuestro objetivo es dar una función `infer` que emule $\bar{\Gamma} \models M : T|_{\mathcal{R}}$.

El primer problema que deberemos resolver es emular el comportamiento del conjunto $\mathbb{N}_{\mathbb{V}}$ (Def. 37). Es decir que deberemos encontrar la manera de tomar una nueva variable de tipo `Z`, para la cual esté garantizado que `Z` no haya sido usada, ni será usada en ningún otro punto de la derivación.

Para esto definamos un alias de tipo para una lista infinita de identificadores disponibles que no hayan sido usados antes. De esta manera podemos acarrear con esta lista durante toda la derivación y cada que necesitemos una variable fresca `Z` $\in \mathbb{N}_{\mathbb{V}}$ tomaremos el identificador en la cabeza de lista y lo eliminaremos de ella en la siguiente llamada recursiva, asegurándonos así de que no volvamos a usar esta variable nunca más.

```
type VarsPool = [Ident]
```

Pensemos en el tipo de la función `infer`. Lo que queremos es que tome una expresión M y nos devuelva un tipo T junto con un conjunto de restricciones $\mathcal{R}$ que necesitaremos para poder calcular el tipo principal de M. Vamos a necesitar acarrear con el conjunto $\mathcal{R}$ durante todo el proceso de inferencia para poder ir agregando las restricciones que se generen, además de que también deberemos poder leer este conjunto para unificarlas al inferir el tipo de un `let`. Además de esto también necesitaremos acarrear durante todo el proceso con el contexto $\bar{\Gamma}$ por lo que un tipo apropiado para la función `infer` sería

$$\texttt{infer} : (\mathbb{N}_{\mathbb{V}}, \bar{\Gamma}, \mathcal{R}) \to \Lambda_p \to \langle T, (\mathbb{N}_{\mathbb{V}}, \bar{\Gamma}, \mathcal{R}) \rangle$$

La tripleta $(\mathbb{N}_{\mathbb{V}}, \bar{\Gamma}, \mathcal{R})$ funciona como un estado el cual querremos que pueda ser leido y modificado por la función `infer` y por otras funciones auxiliares. Por esta razón definiremos un alias de tipo para esta tripleta.

```
type VGC = (VarsPool, Gamma, Constraint)
```

8.2.5. La mónada de Estado + Error

A partir de esta sección haremos uso de un estilo conocido como programación monádica, el cual ha revolucionado el campo de la programación funcional al

permitir una mayor abstracción sobre el flujo de datos en un programa. El concepto de mónada se originó durante la década de 1960 en el contexto del estudio de la teoría de categorías, y por mucho tiempo fue un tema de poco interés para la ciencia de la computación. No fue sino hasta el año de 1991 cuando Eugenio Moggi descubrió su utilidad para el desarrollo de lenguajes de programación funcionales [27]. Más tarde Wadler describió algunos problemas, como el manejo de un estado global, que son elegantemente solucionados con el uso de mónadas [39, 40]. Dado que este es un tema complejo que queda fuera del alcance de este trabajo no ahondaremos en los detalles teóricos ni técnicos. Para un tratamiento detallado del impacto de las mónadas en la programación funcional y su correspondencia con las ternas de Kleisli consultar [34].

Consideremos la regla de inferencia `[p-λx]` (Def. 79)

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \bar{\Gamma}, x{:}\forall.\mathtt{Z} \models M : T \mid_{\mathcal{R}}}{\bar{\Gamma} \models \lambda x.M : \mathtt{Z} \to T \mid_{\mathcal{R}}}\ \texttt{[p-λx]}$$

Para implementar esta regla necesitaríamos usar las siguientes funciones.

```
fresh'    :: VGC                     -> (Type,  VGC)
infer'    :: VGC -> Iswim            -> (Type,  VGC)
putGamma' :: VGC -> Gamma            -> ((),    VGC)
extend'   :: VGC -> Ident -> TScheme -> (Gamma, VGC)
```

Cada una recibe por lo menos como argumento un estado $(\mathbb{N}_{\mathbb{V}}, \bar{\Gamma}, \mathcal{R})$ y devuelve una dupla $\langle t, (\mathbb{N}'_{\mathbb{V}}, \bar{\Gamma}', \mathcal{R}')\rangle$, donde la primera componente es el valor que de verdad nos interesa y la segunda componente es el nuevo estado. Observemos que para la función `putGamma'` el valor que nos devuelve es `()`, la tupla vacía; lo cual indica que en realidad lo que nos importa de esa función no es el valor que devuelva sino la modificación que haga al estado.

Usando estas funciones podríamos definir la regla `[p-λx]` que genera restricciones para $\lambda x.M$ de la siguiente manera

```
infer' vgc (Lam x m) =
  let (z, vgc1) = fresh' vgc
      (g, vgc2) = extend' vgc1 x (Forall [] z)
      (t, vgc3) = infer' vgc2 m
      (_, vgc4) = putGamma' vgc3 g
  in (Func z t, vgc4)
```

La función `fresh'` toma el estado y le modifica el conjunto $\mathbb{N}_{\mathbb{V}}$ eliminando el primer identificador de la lista, devolviendo una variable de tipo `Z` usando tal identificador. La función `extend'` toma el estado que devolvió `fresh'` y le modifica el contexto $\bar{\Gamma}$ para extenderlo con $\bar{\Gamma}, \forall.\mathtt{Z}$, y también devuelve el contexto $\bar{\Gamma}$ antes de ser extendido. Luego `infer'` toma este nuevo estado e

infiere recursivamente que el tipo de M es T devolviendo un nuevo estado en el cual posiblemente se hayan removido identificadores de $fresh$ y se hayan agregado restricciones a $\mathcal{R}$. Finalmente modificamos este último estado para volver a poner el contexto $\bar{\Gamma}$ que nos devolvió la función `extend'` y reportamos entonces que $\lambda x.M$ tiene tipo $\mathtt{Z} \to T$.

Todo este proceso es correcto, sin embargo tiene el inconveniente de que manejar explícitamente el flujo del estado entre cada llamada resulta sumamente tedioso, propenso a errores y dificulta su lectura. Una manera más sencilla de realizar este proceso es usando una mónada de estado, de manera que el estado aparente vivir en un plano superior ajeno al código.

Otro aspecto que es necesario tomar en cuenta es que durante la derivación es posible que ocurran errores de distinta clase que nos impidan tipar una expresión. Definiremos primero estos errores de la siguiente manera

- `UnboundVar` si M no es una expresión cerrada
- `WrongArity` si la lista de operandos no tiene la longitud correcta
- `InfiniteType` si hay restricciones de la forma $\mathtt{A} = T$ con $\mathtt{A} \in \mathcal{V}_\tau(T)$
- `CannotUnify` si es imposible unificar una restricción como $\mathtt{Bool} = \mathtt{A} \to \mathtt{B}$
- `UnimplementedOperator` si hace falta expandir la función `typeOp` de la Sección 8.2.9

```
type HasArity   = Int
type NeedsArity = Int
data TypeError  = UnboundVar Ident
                | WrongArity Op HasArity NeedsArity
                | InfiniteType Type Type
                | CannotUnify Type Type
                | UnimplementedOperator Op
                deriving(Eq,Ord)
```

Ahora definiremos `GK`, una mónada de estado que esté montada sobre una mónada de error.

```
type GK = StateT VGC (Except TypeError)
```

Usando esta mónada redefiniremos las funciones anteriores de la siguiente manera.

```
fresh    ::                     GK Type
infer    :: Iswim            -> GK Type
putGamma :: Gamma            -> GK ()
extend   :: Ident -> TScheme -> GK Gamma
```

Ahora podemos definir la función `infer` usando la notación `do` de Haskell de la siguiente manera.

```
infer (Lam x m) = do
    z <- fresh
    g <- extend x (Forall [] z)
    t <- infer m
    putGamma g  -- descarga el contexto
    return $ Func z t
```

Nótese que esta nueva versión de `infer` no menciona explícitamente al estado en ningún momento, ya que éste es manejado tras bambalinas por el operador monádico *bind* mediante la notacion `<-`.

8.2.6. Algunas funciones monádicas

En esta sección daremos las funciones auxiliares que nos harán falta para definir la función `infer`.

La mónada de estado define las siguientes funciones para obtener y modificar el estado

```
get :: MonadState s m => m s
put :: MonadState s m => s -> m ()
```

Nosotros definiremos las siguientes funciones en términos de las anteriores para poder modificar o acceder a cada una de las componentes del estado por separado.

```
putPool       :: VarsPool   -> GK ()
putGamma      :: Gamma      -> GK ()
putConstraint :: Constraint -> GK ()
putPool       p = get >>= \(_,g,c) -> put (p,g,c)
putGamma      g = get >>= \(p,_,c) -> put (p,g,c)
putConstraint c = get >>= \(p,g,_) -> put (p,g,c)


getPool       :: GK VarsPool
getGamma      :: GK Gamma
getConstraint :: GK Constraint
getPool       = get >>=  \(p,_,_) -> return p
getGamma      = get >>=  \(_,g,_) -> return g
getConstraint = get >>=  \(_,_,c) -> return c
```

$\mathbb{N}_{\mathbb{V}}$ **Variables frescas (Def. 37)**

```
fresh' :: GK TVar
fresh' = do
    p:ps <- getPool
    putPool ps
    return $ TVar p


fresh :: GK Type
fresh = fresh' >>= return . Var
```

`gen`$(\overline{\Gamma}, T)$ Generalización de variables (Def. 51)

```
generalize :: Type -> GK TScheme
generalize t = do
    g      <- getGamma
    let xs =  map TVar $ S.toList $ S.difference (getTVars t) (getTVars g)
    zs     <- mapM (const fresh') xs
    let s  =  S $ M.fromList (zip xs $ map Var zs)
    return $ Forall zs (subst s t)
```

`finst`$(\mathcal{E})$ Instanciación de esquemas de tipo (Def. 52)

```
instantiate :: TScheme -> GK Type
instantiate (Forall xs t) = do
    ys <- mapM (const fresh) xs
    let s = S $ M.fromList (zip xs ys)
    return $ subst s t
```

Extendiendo el esquema de tipos

La función `extend` toma un identificador x y un esquema de tipos $\mathcal{E}$; y actualiza el contexto $\overline{\Gamma}$ con $\overline{\Gamma}, x : \mathcal{E}$. También devuelve el contexto anterior para que podamos sustituir el contexto expandido por el anterior.

```
extend :: Ident -> TScheme -> GK Gamma
extend x sc = do
    G g <- getGamma
    putGamma (G $ M.insert x sc g)
    return $ G g
```

Agregando restricciones a $\mathcal{R}$

Usaremos la función `addConstraint` para agregar una restricción de tipos al estado.

```
addConstraint ::(Type,Type) -> GK ()
```

```
addConstraint c = do
    R cs <- getConstraint
    putConstraint $ R (c:cs)
```

8.2.7. Algoritmo de unificación $\mathcal{U}$

Definiremos el algoritmo de unificación $\mathcal{U}$ (Def. 43) de Martelli-Montanari fuera de nuestra mónada `GK` para dejar claro que la unificación no alterará el estado de ninguna manera.

```
unify :: Constraint -> Either TypeError Substitution
unify (R []) = Right emptySub
unify (R ((s,t):cs))
    | s == t = unify $ R cs
unify (R ((Func s1 s2, Func t1 t2):cs)) = unify cs'
    where cs' = R $  [(s1,t1), (s2,t2)] ++ cs
unify (R ((Var x, t):cs))
    | occursCheck x t = do
        let xt = mkSub x t
        u  <- unify $ subst xt (R cs)
        return $ subComp u xt
    | otherwise = Left $ InfiniteType (Var x) t
unify (R ((t, Var x):cs)) = unify $ R $ (Var x,t):cs
unify (R ((s,t):_))       = Left $ CannotUnify s t
```

El algoritmo devuelve ya sea una sustitución en caso de que $\mathcal{R}$ sea unificable o un error en caso contrario. Es posible que ocurran dos tipos de errores durante la unificación. Uno de ellos ocurre en el último caso del algoritmo que es cuando se intenta unificar una restricción con tipos incompatibles como `Nat` = `Bool`.

El otro posible error ocurre cuando se intenta unificar una restricción de la forma `A` $= T$ donde `A` es una variable de tipo y `A` $\in \mathcal{V}_\tau(T)$. Con esto nos aseguramos de que la solución no contenga una sustitución cíclica como `A` $\mapsto$ `A` $\to$ `A`. Esta condición es la que nos impedirá tipar expresiones como $(\lambda x.x\ x)$ o los combinadores de punto fijo con un tipo infinito. Verificaremos que esto no suceda con la función `occursCheck`.

```
occursCheck :: TVar -> Type -> Bool
occursCheck tv t = tv `notElem` ftv
    where ftv = map TVar (getTVarsLs t)
```

Finalmente definiremos una función que sí viva dentro de la mónada `GK` y que devuelva el unificador de $\mathcal{R}$ en caso de que éste exista o que lance el error correspondiente en caso contrario.

```
getUnifier :: GK Substitution
```

```
getUnifier = do
    c <- getConstraint
    case unify c of
        Left  e -> throwError e
        Right u -> return u
```

8.2.8. Generación de restricciones $\bar{\Gamma} \models M : T|_{\mathcal{R}}$

A partir de este momento podemos definir la función `infer`, para cada caso nos guiaremos con las reglas de la Definición 79. Es importante notar que el conjunto $\mathcal{R}$ no figura directamente en el código ya que éste ha quedado escondido en la mónada de estado; y la única manera en la que interactuamos con él es al agregar restricciones de tipo mediante la función `addConstraint`.

```
infer :: Iswim -> GK Type
```

Variables $x \in \mathbb{X}$

Para implementar esta regla primero obtenemos el contexto $\bar{\Gamma}$ del estado, y buscamos en él a la variable x, si la encontramos entonces instanciamos su respectivo esquema de tipos y devolvemos eso como resultado, en caso contrario lanzamos un error diciendo que la variable x no está ligada y ahí termina la derivación.

```
infer (IVar x) = do
    G g <- getGamma
    case M.lookup x g of
        Nothing     -> throwError $ UnboundVar x
        Just scheme -> instantiate scheme
```

$$\frac{\bar{\Gamma}(x) = \mathcal{E}}{\bar{\Gamma} \models x : \texttt{finst}(\mathcal{E}) \mid \varnothing}\ \texttt{[p-var]}$$

Constantes primitivas $\dot{c} \in \mathcal{C}_{\mathcal{P}}$

Para implementar la regla `[p-cte]` usaremos la función `typeCte` que es equivalente a la función $\mathcal{C}_{\mathcal{T}}$ (Def. 25). Definiremos `typeCte` en la Sección 8.2.9.

```
infer (C c) = return $ typeCte c
```

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\bar{\Gamma} \models \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c}) \mid \varnothing}\ \texttt{[p-cte]}$$

Funciones $(\lambda x.M)$

```
infer (Lam x m) = do
    z <- fresh
    g <- extend x (Forall [] z)
    t <- infer m
    putGamma g  -- descarga el contexto
    return $ Func z t
```

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \bar{\Gamma}, x{:}\forall.\mathtt{Z} \models M{:}T \mid \mathcal{R}}{\bar{\Gamma} \models \lambda x.M : \mathtt{Z} \to T \mid \mathcal{R}}\ \texttt{[p-λx]}$$

Aplicaciones $(M\ N)$

```
infer (App m n) = do
    z  <- fresh
    t1 <- infer m
    t2 <- infer n
    addConstraint (t1, Func t2 z)
    return z
```

$$\frac{\bar{\Gamma} \models M{:}T_1 \mid \mathcal{R}_1 \quad \bar{\Gamma} \models N{:}T_2 \mid \mathcal{R}_2 \quad \mathtt{Z} \in \mathbb{N}_{\mathbb{V}}}{\bar{\Gamma} \models M\ N : \mathtt{Z} \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = T_2 \to \mathtt{Z}\}}\ \texttt{[p-app]}$$

Condicional $(\mathtt{if}\ M\ N\ L)$

```
infer (If m n l) = do
    t1 <- infer m
    t2 <- infer n
    t3 <- infer l
    addConstraint (t1, tBool)
    addConstraint (t2, t3)
    return t2
```

$$\frac{\bar{\Gamma} \models M{:}T_1 \mid \mathcal{R}_1 \quad \bar{\Gamma} \models N{:}T_2 \mid \mathcal{R}_2 \quad \bar{\Gamma} \models L{:}T_3 \mid \mathcal{R}_3}{\bar{\Gamma} \models \mathtt{if}\ M\ N\ L : T_2 \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \{T_1 = \mathtt{Bool}, T_2 = T_3\}}\ \texttt{[p-if]}$$

Punto fijo $(\mathtt{Fix}\ M)$

```
infer (Fix m) = do
    z <- fresh
    t <- infer m
    addConstraint (t, Func z z)
    return z
```

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \bar{\Gamma} \models M{:}T \mid \mathcal{R}}{\bar{\Gamma} \models \mathtt{Fix}\ M : \mathtt{Z} \mid \mathcal{R} \cup \{T = \mathtt{Z} \to \mathtt{Z}\}}\ \texttt{[p-fix]}$$

Expresiones `let`

```
infer (Let x n m) = do
    t1'  <- infer n
    u    <- getUnifier
    getGamma >>= \g -> putGamma (subst u g)
    let t1 = subst u t1'
    e     <- generalize t1
    g     <- extend x e
    putConstraint emptyR
    t2    <- infer m
    putGamma g
    return t2
```

$$\frac{\begin{array}{c}\bar{\Gamma}' \models N : T_1' \mid \mathcal{R}_1 \\ \mu = \mathcal{U}(\mathcal{R}_1) \\ \bar{\Gamma} = \mu(\bar{\Gamma}') \\ T_1 = \mu(T_1') \\ \bar{\Gamma}, x : \mathtt{gen}(\bar{\Gamma}, T_1) \models M : T_2 \mid \mathcal{R}_2\end{array}}{\bar{\Gamma} \models \mathtt{let}\ x = N\ \mathtt{in}\ M : T_2 \mid \mathcal{R}_2}\ \texttt{[p-let]}$$

Operadores primitivos $o^{(n)} \in \mathcal{O}_{\mathcal{P}}$

Finalmente damos una regla para los operadores primitivos que queda definida en términos de la función `typeOp` la cual definimos en la Sección 8.2.9.

```
infer (Op op xs) = typeOp op xs
```

8.2.9. Expandiendo el lenguaje

En esta sección definiremos las funciones que habrá que expandir según sea necesario para agregar nuevos operadores primitivos o constantes al lenguaje.

Constantes primitivas

Usaremos `typeCte` para definir a la función $\mathcal{C}_{\mathcal{T}}$ (Def. 25) que determina el tipo de cada constante primitiva.

```
typeCte :: Cte -> Type
typeCte Nat{} = tNat
typeCte TF{}  = tBool


tBool :: Type
tNat  :: Type
tBool =  CType "Bool"
tNat  =  CType "Nat"
```

Operadores primitivos

Las reglas para la generación de restricciones (Def. 79) no contemplan a los operadores primitivos por lo que hace falta dar una regla distinta para cada uno de ellos. Usaremos la función `typeOp` para definir cada una de estas reglas.

```
typeOp :: Op -> [Iswim] -> GK Type
```

```
typeOp IsZero [m] = do
    t <- infer m
    addConstraint (t,tNat)
    return tBool
```

$$\frac{\Gamma \models M : T \mid \mathcal{R}}{\Gamma \models \texttt{isZero}\ [M] : \texttt{Bool} \mid \mathcal{R} \cup \{T = \texttt{Nat}\}}$$

```
typeOp Pred [m] = do
    t <- infer m
    addConstraint (t,tNat)
    return tNat
```

$$\frac{\Gamma \models M : T \mid \mathcal{R}}{\Gamma \models \texttt{pred}\ [M] : \texttt{Nat} \mid \mathcal{R} \cup \{T = \texttt{Nat}\}}$$

```
typeOp Prod [m1,m2] = do
    t1 <- infer m1
    t2 <- infer m2
    addConstraint (t1, tNat)
    addConstraint (t2, tNat)
    return tNat
```

$$\frac{\Gamma \models M_1 : T_1 \mid \mathcal{R}_1 \quad \Gamma \models M_2 : T_2 \mid \mathcal{R}_2}{\Gamma \models\ \texttt{*}\ [M_1\ M_2] : \texttt{Nat} \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = \texttt{Nat}, T_2 = \texttt{Nat}\}}$$

```
typeOp Sum [m1,m2] = do
    t1 <- infer m1
    t2 <- infer m2
    addConstraint (t1, tNat)
    addConstraint (t2, tNat)
    return tNat
```

$$\frac{\Gamma \models M_1 : T_1 \mid \mathcal{R}_1 \quad \Gamma \models M_2 : T_2 \mid \mathcal{R}_2}{\Gamma \models \texttt{+}\ [M_1\ M_2] : \texttt{Nat} \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = \texttt{Nat}, T_2 = \texttt{Nat}\}}$$

```
typeOp Not [m]  = do
    t <- infer m
    addConstraint(t,tBool)
    return tBool
```

$$\frac{\Gamma \models M : T \mid \mathcal{R}}{\Gamma \models \texttt{not}\ [M] : \texttt{Bool} \mid \mathcal{R} \cup \{T = \texttt{Bool}\}}$$

Cuando ninguno de los casos que hemos previsto se cumpla, entonces deberemos de reportar un error el cual puede deberse a que el operador no recibió la cantidad correcta de operandos o porque simplemente hemos olvidado escibir la regla para algún caso. Lidiamos con estos errores usando la función `arityError` que en caso de ser necesario lanza la excepción correspondiente a la mónada.

```
typeOp m xs = arityError m xs

arityError :: Op -> [Iswim] -> GK Type
arityError op@IsZero xs = throwError $ WrongArity op (length xs) 1
```

```
arityError op@Pred   xs = throwError $ WrongArity op (length xs) 1
arityError op@Prod   xs = throwError $ WrongArity op (length xs) 2
arityError op@Sum    xs = throwError $ WrongArity op (length xs) 2
arityError op@Not    xs = throwError $ WrongArity op (length xs) 1
arityError op        xs = throwError $ UnimplementedOperator op
```

8.2.10. Calculando el tipo principal

Para terminar combinaremos la generación de restricciones con la unificación para devolver el tipo principal de una expresión. Si $\models M : T|_{\mathcal{R}}$ y μ es el unificador principal de $\mathcal{R}$, entonces $\vdash M : \mu(T)$ es el tipo principal de M. Aplicaremos un renombre a las variables de tipo antes de devolver el tipo principal de manera que en lugar de devolver algo como $(\mathtt{C} \to \mathtt{F} \to \mathtt{D}) \to (\mathtt{C} \to \mathtt{F}) \to \mathtt{C} \to \mathtt{D}$ devolvamos en cambio $(\mathtt{A} \to \mathtt{C} \to \mathtt{B}) \to (\mathtt{A} \to \mathtt{C}) \to \mathtt{A} \to \mathtt{B}$

```
principalType' :: Iswim -> GK Type
principalType' m = do
    t <- infer m
    u <- getUnifier
    return $ shift $ subst u t

shift :: Type -> Type
shift t = subst s t
    where ks = map TVar $ S.toList $ getTVars t
          vs = map (Var . TVar) $ take (length ks) allVars
          s  = S $ M.fromList (zip ks vs)
```

El tipo que obtenemos con la función anterior aún vive en la mónada que definimos. Para obtener un valor de tipo `Either TypeError Type` necesitaremos definir el estado inicial donde Γ y $\mathcal{R}$ están vacíos y $\mathbb{N}_{\mathbb{V}}$ es un conjunto infinito de identificadores correspondiente con la expresión regular $(a - z)^+$.

```
principalType :: Iswim -> Either TypeError Type
principalType m = runExcept $ evalStateT (principalType' m)  initialState


initialState :: VGC
initialState = (allVars, emptyG, emptyR)

allVars :: VarsPool
allVars = [1..] >>= flip replicateM ['a'..'z']
```

8.3. La máquina CEK

En esta sección implementaremos la máquina CEK para definir un evaluador de Λ_p que solamente opere sobre expresiones que puedan ser tipadas correctamente.

```
module CEK where

import Iswim

import Data.Maybe
import qualified Data.Map as M
```

8.3.1. Cerraduras

Primero daremos las definiciones de cerraduras (Def. 62) y entornos de evaluación (Def. 63) las cuales son mutuamente recursivas.

```
data Closure = Cl Iswim Env  deriving(Eq,Show,Ord)
type Env     = M.Map Ident Closure
```

Definiremos la función $\varepsilon(x)$ que busca en ε la cerradura asociada a x. No tomaremos ninguna medida adicional para recuperarnos del error que ocurra en caso de que $x \notin dom(\varepsilon)$ ya que este error sería detectado durante la verificación de tipos.

```
env :: Env -> Ident -> (Iswim, Env)
env e x = case M.lookup x e of
    Just (Cl m e') -> (m, e')
    Nothing        -> error $ "Unbound var " ++ x
```

Si $x \mapsto \langle M, \varepsilon' \rangle \in \varepsilon$ entonces definimos las funciones $\varepsilon_\Lambda(x) = M$ y $\varepsilon_\varepsilon(x) = \varepsilon'$

```
envM :: Env -> Ident -> Iswim
envM e x = fst $ env e x

envE :: Env -> Ident -> Env
envE e x = snd $ env e x
```

También será necesario definir una función que extienda un entorno de evaluación ε con la asociación $x \mapsto \langle M, \varepsilon' \rangle$.

```
extend :: Env -> Ident -> Iswim -> Env -> Env
extend e x v e' = M.insert x (Cl v e') e
```

8.3.2. Continuaciones con cerraduras (Def. 64)

```
data Kont = Mt
          | Ar Iswim  Env Kont
          | Fn Lambda Env Kont
          | O [Value] Op [Iswim] Env Kont
          | I Iswim Iswim Env Kont  -- If
          | F Kont                  -- Fix
          deriving (Ord,Show,Eq)

type Lambda = Iswim
type Value  = Iswim
```

8.3.3. La máquina CEK

La máquina CEK (Def. 82) usa estados internos de la forma $\langle \Lambda_p, \varepsilon, \kappa \rangle$, por lo que definiremos un alias de tipo para esta triada.

```
type CEK = (Iswim, Env, Kont)
```

Definiremos la función `step` análogamente a la tabla de transiciones de la máquina CEK (Def. 82), que simula la transición entre dos estados.

```
step :: CEK -> CEK
```

Cuando la máquina llegue a un estado final devolverá el mismo estado. En este caso no consideraremos a las variables como valores ya que la regla `[cek-var]` se encarga de lidiar con estos casos.

```
isFinal :: CEK -> Bool
isFinal (c,_,k) = isValue c && k == Mt

isValue :: Iswim -> Bool
isValue Lam{}  = True
isValue C{}    = True
isValue _      = False
```

La tabla de transiciones es la siguiente: (Def. 82)

```
step  cek | isFinal cek = cek
step (IVar x      ,e, k                ) = (envM e x, envE e x, k)
step (v           ,e, Ar n e' k        ) | isValue v = (n, e', Fn v e k)
step (v           ,e, Fn (Lam x m) e' k ) | isValue v = (m, extend e' x v e, k)
step (v           ,e, O vs o [] e' k   ) | isValue v = (delta o $ reverse (v:vs), M.empty, k)
step (v           ,e, O vs o (m:ms) e' k) | isValue v = (m, e', O (v:vs) o ms e' k)
step (v           ,e, I n l e' k       ) | isValue v = (if' v n l, e', k)
step (Lam f m     ,e, F k              ) = (m, extend e f (Fix $ Lam f m) e, k)
step (App m n     ,e, k                ) = (m, e, Ar n e k)
```

```
step (Op o (m:ms) ,e, k                ) = (m, e, O [] o ms e k)
step (If m n l    ,e, k                ) = (m, e, I n l e k)
step (Fix m       ,e, k                ) = (m, e, F k)
step (Let x n m   ,e, k                ) = (App (Lam x m) n, e, k)
```

8.3.4. Evaluando un programa

Para evaluar un programa con la máquina CEK deberemos dar una función `load` que convierta un programa M en un estado inicial $\langle M, \varnothing, \texttt{Mt}\rangle$.

```
load :: Iswim -> CEK
load m = (m, M.empty, Mt)
```

Es posible que un estado final sea de la forma $\langle \lambda x.M, \varepsilon, \texttt{Mt}\rangle$ donde M contiene variables libres que deben de ser sustituidas por sus respectivas expresiones. Definiremos la función $\texttt{unload}(\langle M, \varepsilon, \texttt{Mt}\rangle) \stackrel{\text{def}}{=} \Psi(\langle M, \varepsilon\rangle)$ que toma un estado final de la máquina CEK y que devuelve una expresión de ISWIM. Para hacer esto usaremos la función `real` $\stackrel{\text{def}}{=} \Psi$ (Def. 66) que toma una cerradura $\langle M, \varepsilon\rangle$ y una lista de identificadores; la función desciende recursivamente sobre M de manera que cada que encuentre una variable la remplace por $\Psi(\varepsilon(x))$ si ésta aparecía libre en M. Para evitar sustituir una variable ligada usamos la lista `bs` a la cual agregamos el identificador x cuando se encuentre una expresión $\lambda x.M$ o `let` $x = N$ `in` M.

```
unload :: CEK -> Value
unload (m, e, _) = real [] e m

real :: [Ident] -> Env -> Iswim -> Iswim
real bs e (IVar x)
    | elem x bs       = IVar x
    | otherwise       = real bs (envE e x) (envM e x)
real _  _ c@C{}       = c
real bs e (Lam x m)   = Lam x $ real (x:bs) e m
real bs e (App m n)   = App (real bs e m) (real bs e n)
real bs e (Op op ms)  = Op op $ map (real bs e) ms
real bs e (If m n l)  = If (real bs e m) (real bs e n) (real bs e l)
real bs e (Fix m)     = Fix $ real bs e m
real bs e (Let x n m) = Let x (real bs e n) (real (x:bs) e m)
```

Para simular la ejecución de un programa usaremos la función `runCEK` que toma un estado de la máquina CEK y aplica la función `step` sobre él hasta llegar a un punto fijo, el cual corresponde con el estado final de la máquina.

```
runCEK :: CEK -> CEK
runCEK cek
    | step cek == cek = cek
```

```
    | otherwise = runCEK $ step cek
```

Finalmente definimos la función `eval` que toma una expresión de ISWIM, la carga a un estado inicial de la máquina CEK, la cual se ejecuta hasta detenerse en un estado final y finalmente devuelve el valor correspondiente usando la función `unload`.

```
eval :: Iswim -> Value
eval = unload . runCEK . load
```

8.3.5. Expandiendo el lenguaje

Para expandir el lenguaje con nuevos operadores primitivos será necesario hacer las modificaciones correspondientes a la función de evaluación δ (Def. 18).

```
delta :: Op -> [Value] -> Value
delta IsZero [C (Nat a)]             = C . TF  $ a == 0
delta Sum    [C (Nat a), C (Nat b)]  = C . Nat $ a + b
delta Prod   [C (Nat a), C (Nat b)]  = C . Nat $ a * b
delta Pred   [C (Nat a)]             = C . Nat $ a - 1
delta Not    [C (TF a) ]             = C . TF  $ not a
delta op xs = error $ "Unimplemented operator CEK:delta" ++ show op ++ show xs
```

Con esto finalizamos la implementación de ISWIM polimórfico.

Capítulo 9

Conclusiones y trabajo futuro

Este trabajo se originó como un intento de comprender el trasfondo teórico de los lenguajes de programación funcionales con interés particular en la inferencia de tipos. Es la esperanza del autor que al unificar diversos temas bajo una misma notación, estos resulten más accesibles para un lector inexperto, con la finalidad de despertarle el interés de continuar las investigaciones actuales en este campo.

Con este trabajo hemos tratado un pequeño subconjunto de características deseables en un buen lenguaje funcional. Notoriamente ha quedado fuera del alcance este trabajo temas como tipos algebraicos, evaluación perezosa, paralelismo y tipos polimórficos de orden superior. Naturalmente es deseable incorporar estas características en un trabajo futuro. Asimismo, en este trabajo solamente hemos mostrado los elementos fundamentales de ISWIM polimórfico, estos son: el cálculo, el sistema de tipos, y la máquina evaluadora. Todavía es necesario diseñar una representación física de Λ_p que sea más amigable para el programador final, de manera que sea posible implementar un parser que transforme esta sintaxis concreta en expresiones de Λ_p.

No obstante, consideramos que el objetivo principal de este trabajo ha sido cumplido, ya que hemos mostrado la eficacia del cálculo-λ como herramienta para diseñar y estudiar la semántica de los lenguajes de programación. Asimismo esperamos haber mostrado mediante el uso de programación monádica la belleza y elegancia inherentes al paradigma funcional, resultado de un amplio estudio de la teoría de categorías.

Apéndice A

Iswim simplemente tipado

Definición 68: Λ_t **Iswim simplemente tipado**

$$
\begin{array}{rcll}
\Lambda_t & = & x & \texttt{[var]} \\
& | & \dot{c} & \texttt{[cte]} \\
& | & (\lambda x{:}T.\, M_1) & \texttt{[}\lambda\texttt{x:T.M]} \\
& | & (M_1\ M_2) & \texttt{[app]} \\
& | & (\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ ) & \texttt{[op[M]]} \\
& | & (\texttt{if}\ \ M_1\ \ M_2\ \ M_3) & \texttt{[if]} \\
& | & (\texttt{Fix}\ \ M_1) & \texttt{[fix]}
\end{array}
$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad T \in \mathcal{T} \quad o^{(n)} \in \mathcal{O}_{\mathcal{P}} \quad M_i \in \Lambda_t$

Definición 69: $\mathcal{V}_{\mathcal{L}}$ **para ISWIM simplemente tipado**

$$\begin{aligned}
\mathcal{V}_{\mathcal{L}}(x) &= \{x\} \\
\mathcal{V}_{\mathcal{L}}(\dot{c}) &= \varnothing \\
\mathcal{V}_{\mathcal{L}}(\lambda x{:}T.\, M) &= \mathcal{V}_{\mathcal{L}}(M) \setminus \{x\} \\
\mathcal{V}_{\mathcal{L}}(M\ N) &= \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N) \\
\mathcal{V}_{\mathcal{L}}(\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ ) &= \cup_{i=1}^{n}\, \mathcal{V}_{\mathcal{L}}(M_i) \\
\mathcal{V}_{\mathcal{L}}(\texttt{if}\ M\ N\ L) &= \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N) \cup \mathcal{V}_{\mathcal{L}}(L) \\
\mathcal{V}_{\mathcal{L}}(\texttt{Fix}\ M) &= \mathcal{V}_{\mathcal{L}}(M)
\end{aligned}$$

Definición 70: **Sustitución para ISWIM simplemente tipado**

$$x[y \leftarrow N] = \begin{cases} N & \text{si } x = y \\ x & \text{si } x \neq y \end{cases}$$

$$\dot{c}[y \leftarrow N] = \dot{c}$$

$$(\lambda x{:}T.\, M)[y \leftarrow N] = \begin{cases} \lambda x{:}T.\, M & \text{si } x = y \\ \lambda z{:}T.\, M[x \leftarrow z][y \leftarrow N] & \text{si } x \neq y \end{cases}$$
$$\text{donde} \quad z \notin \mathcal{V}_{\mathcal{L}}(M) \cup \mathcal{V}_{\mathcal{L}}(N)$$

$$\begin{aligned}
(M_1\ M_2)[y \leftarrow N] &= M_1[y \leftarrow N]\ M_2[y \leftarrow N] \\
(\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ )[y \leftarrow N] &= o^{(n)}\ [\ M_i[y \leftarrow N]\ ]_{i=1}^{n} \\
(\texttt{if}\ M_1\ M_2\ M_3)[y \leftarrow N] &= \texttt{if}\ M_1[y \leftarrow N]\ M_2[y \leftarrow N]\ M_3[y \leftarrow N] \\
(\texttt{Fix}\ M)[y \leftarrow N] &= \texttt{Fix}\ (M[y \leftarrow N])
\end{aligned}$$

Definición 71: **Sistema de tipos simple**

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \texttt{[t-var]}$$

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\Gamma \vdash \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c})}\ \texttt{[t-cte]}$$

Definición 71: Sistema de tipos simple (cont.)

$$\frac{\Gamma,\ x{:}T_1 \ \vdash\ M{:}T_2}{\Gamma \ \vdash\ \lambda x{:}T_1.\ M\ :\ T_1 \to T_2}\ \texttt{[t-λx]}$$

$$\frac{\Gamma \vdash M{:}T_1 \to T_2 \qquad \Gamma \vdash N{:}T_1}{\Gamma \vdash M\ \ N : T_2}\ \texttt{[t-app]}$$

$$\frac{\Gamma \vdash M_i{:}T_i \quad 1 \le i \le n}{\Gamma \vdash o^{(n)}\ [\ M_i\ ]_{i=1}^{n} : \Delta(o^{(n)}, [T_i])}\ \texttt{[t-op]}$$

$$\frac{\Gamma \vdash M{:}\texttt{Bool} \qquad \Gamma \vdash N{:}T \qquad \Gamma \vdash L{:}T}{\Gamma \vdash \texttt{if}\ \ M\ \ N\ \ L{:}T}\ \texttt{[t-if]}$$

$$\frac{\Gamma \vdash M{:}T \to T}{\Gamma \vdash \texttt{Fix}\ \ M{:}T}\ \texttt{[t-fix]}$$

Definición 72: $\mathcal{V}_t$ Valores de Λ_t

$$\begin{aligned}\mathcal{V}_t = {} & x \\ \mid {} & \dot{c} \\ \mid {} & \lambda x{:}T.\ M\end{aligned}$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad T \in \mathcal{T} \quad M \in \Lambda_t$

Definición 73: Semántica operacional de Λ_t

$$\frac{M \to M'}{(M\ N) \to (M'\ N)}\ \texttt{[ev-app-i]}$$

$$\frac{v \in \mathcal{V} \qquad N \to N'}{(v\ N) \to (v\ N')}\ \texttt{[ev-app-d]}$$

$$\frac{v \in \mathcal{V}}{(\lambda x{:}T.\ M)\ \ v \to M[x \leftarrow v]}\ \texttt{[ev-red]}$$

Definición 73: Semántica operacional de Λ_t (cont.)

$$\frac{M \to M'}{\texttt{if}\ \ M\ \ N\ \ L \to \texttt{if}\ \ M'\ \ N\ \ L}\ \texttt{[ev-if-cnd]}$$

$$\frac{}{\texttt{if true}\ \ N\ \ L \to N}\ \texttt{[ev-if-tr]}$$

$$\frac{}{\texttt{if false}\ \ N\ \ L \to L}\ \texttt{[ev-if-fl]}$$

$$sel([M_i]) = \text{mín}(\{i \in \mathbb{N} | M_i \notin \mathcal{C}_\mathcal{P}\} \cup \{n+1\})$$

$$\frac{j = sel(\ [M_i]\ ) \quad j \leq n \quad M_j \to M'_j}{o^{(n)}\ [\ M_i\ ]_{i=1}^{n} \to o^{(n)}\ [M_1\ \ldots\ M'_j\ \ldots\ M_n]}\ \texttt{[ev-}o^n\texttt{-v]}$$

$$\frac{sel(\ [M_i]\ ) = n+1}{o^{(n)}\ [\ M_i\ ]_{i=1}^{n} \to \delta(o^{(n)},\ [\ M_i\ ]_{i=1}^{n})}\ \texttt{[ev-}o^n\texttt{-}\delta\texttt{]}$$

$$\frac{M \to M'}{\texttt{Fix}\ \ M \to \texttt{Fix}\ \ M'}\ \texttt{[ev-fix-b]}$$

$$\frac{}{\texttt{Fix}\ \ \lambda f{:}T.\ M\ \ \to\ \ M[f \leftarrow \texttt{Fix}\ \ \lambda f{:}T.\ M]}\ \texttt{[ev-fix]}$$

Apéndice B

Iswim con inferencia de tipos

Definición 74: Λ_a **Iswim sin anotaciones de tipo [32]**

$$
\begin{array}{rcll}
\Lambda_a & = & x & \texttt{[var]} \\
& | & \dot{c} & \texttt{[cte]} \\
& | & (\lambda x.\, M_1) & \texttt{[}\lambda\texttt{x:T.M]} \\
& | & (M_1\ M_2) & \texttt{[app]} \\
& | & (\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ ) & \texttt{[op[M]]} \\
& | & (\texttt{if}\ \ M_1\ \ M_2\ \ M_3) & \texttt{[if]} \\
& | & (\texttt{Fix}\ \ M_1) & \texttt{[fix]}
\end{array}
$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad T \in \mathcal{T} \quad o^{(n)} \in \mathcal{O}_{\mathcal{P}} \quad M_i \in \Lambda_a$

Definición 75: **Sistema de tipos simple sin anotaciones [32]**

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \texttt{[a-var]}$$

Definición 75: Sistema de tipos simple sin anotaciones [32] (cont.)

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\Gamma \vdash \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c})}\ \texttt{[a-cte]}$$

$$\frac{\Gamma,\ x{:}T_1\ \vdash\ M{:}T_2}{\Gamma\ \vdash\ \lambda x.\, M\ :\ T_1 \to T_2}\ \texttt{[a-}\lambda\texttt{x]}$$

$$\frac{\Gamma \vdash M{:}T_1 \to T_2 \qquad \Gamma \vdash N{:}T_1}{\Gamma \vdash M\ \ N : T_2}\ \texttt{[a-app]}$$

$$\frac{\Gamma \vdash M_i{:}T_i \quad 1 \leq i \leq n}{\Gamma \vdash o^{(n)}\ [\ M_i\ ]_{i=1}^{n} : \Delta(o^{(n)}, [T_i])}\ \texttt{[a-op]}$$

$$\frac{\Gamma \vdash M{:}\texttt{Bool} \qquad \Gamma \vdash N{:}T \qquad \Gamma \vdash L{:}T}{\Gamma \vdash \texttt{if}\ \ M\ \ N\ \ L{:}T}\ \texttt{[a-if]}$$

$$\frac{\Gamma \vdash M{:}T \to T}{\Gamma \vdash \texttt{Fix}\ \ M{:}T}\ \texttt{[a-fix]}$$

Definición 76: Generación de restricciones [32]

$$\frac{\Gamma(x) = T}{\Gamma \models x{:}T \mid \varnothing}\ \texttt{[c-var]}$$

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\Gamma \models \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c}) \mid \varnothing}\ \texttt{[c-cte]}$$

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \Gamma, x{:}\mathtt{Z} \models M{:}T \mid \mathcal{R}}{\Gamma \models \lambda x.M : \mathtt{Z} \to T \mid \mathcal{R}}\ \texttt{[c-}\lambda\texttt{x]}$$

$$\frac{\Gamma \models M{:}T_1 \mid \mathcal{R}_1 \quad \Gamma \models N{:}T_2 \mid \mathcal{R}_2 \quad \mathtt{Z} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models M\ N : \mathtt{Z} \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = T_2 \to \mathtt{Z}\}}\ \texttt{[c-app]}$$

$$\frac{\Gamma \models M{:}T_1 \mid \mathcal{R}_1 \quad \Gamma \models N{:}T_2 \mid \mathcal{R}_2 \quad \Gamma \models L{:}T_3 \mid \mathcal{R}_3}{\Gamma \models \texttt{if}\ M\ N\ L : T_2 \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \{T_1 = \texttt{Bool}, T_2 = T_3\}}\ \texttt{[c-if]}$$

Definición 76: Generación de restricciones [32] (cont.)

$$\frac{\Gamma \models M : T \mid_{\mathcal{R}} \quad \mathtt{Z} \in \mathbb{N}_{\mathbb{V}}}{\Gamma \models \mathtt{Fix}\ M : \mathtt{Z} \mid_{\mathcal{R} \cup \{T = \mathtt{Z} \to \mathtt{Z}\}}} \texttt{[c-fix]}$$

Apéndice C

Iswim polimórfico

Definición 77: Λ_p **Iswim polimórfico**

$$
\begin{array}{rcll}
\Lambda_p & = & x & \texttt{[var]} \\
& | & \dot{c} & \texttt{[cte]} \\
& | & (\lambda x.\, M_1) & \texttt{[}\lambda\texttt{x.M]} \\
& | & (M_1\ M_2) & \texttt{[app]} \\
& | & (\ o^{(n)}\ [\ M_i\ ]_{i=1}^{n}\ ) & \texttt{[op[M]]} \\
& | & (\texttt{if}\ \ M_1\ \ M_2\ \ M_3) & \texttt{[if]} \\
& | & (\texttt{Fix}\ \ M_1) & \texttt{[fix]} \\
& | & (\texttt{let}\ \ x = M_1\ \ \texttt{in}\ \ M_2) & \texttt{[let]}
\end{array}
$$

donde $x \in \mathbb{X} \quad \dot{c} \in \mathcal{C}_{\mathcal{P}} \quad o^{(n)} \in \mathcal{O}_{\mathcal{P}} \quad M_i \in \Lambda_p$

Definición 78: **Sistema de tipos simple para** Λ_p

$$\frac{\bar{\Gamma}(x) = \mathcal{E}}{\bar{\Gamma} \vdash x : \texttt{finst}(\mathcal{E})}\ \texttt{[tp-var]}$$

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\bar{\Gamma} \vdash \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c})}\ \texttt{[tp-cte]}$$

Definición 78: Sistema de tipos simple para Λ_p (cont.)

$$\frac{\bar{\Gamma},\ x{:}\forall.T_1 \ \vdash\ M{:}T_2}{\bar{\Gamma}\ \vdash\ \lambda x.\ M\ :\ T_1 \to T_2}\ \texttt{[tp-}\lambda\texttt{x]}$$

$$\frac{\bar{\Gamma} \vdash M{:}T_1 \to T_2 \qquad \bar{\Gamma} \vdash N{:}T_1}{\bar{\Gamma} \vdash M\ \ N : T_2}\ \texttt{[tp-app]}$$

$$\frac{\bar{\Gamma} \vdash M_i{:}T_i \quad 1 \le i \le n}{\bar{\Gamma} \vdash o^{(n)}\ [\ M_i\ ]_{i=1}^{n} : \Delta(o^{(n)}, [T_i])}\ \texttt{[tp-op]}$$

$$\frac{\bar{\Gamma} \vdash M{:}\texttt{Bool} \qquad \bar{\Gamma} \vdash N{:}T \qquad \bar{\Gamma} \vdash L{:}T}{\bar{\Gamma} \vdash \texttt{if}\ \ M\ \ N\ \ L{:}T}\ \texttt{[tp-if]}$$

$$\frac{\bar{\Gamma} \vdash M{:}T \to T}{\bar{\Gamma} \vdash \texttt{Fix}\ \ M{:}T}\ \texttt{[tp-fix]}$$

$$\frac{\bar{\Gamma} \vdash N{:}T_1 \qquad \bar{\Gamma}, x{:}\texttt{gen}(\bar{\Gamma}, T_1) \vdash M{:}T_2}{\bar{\Gamma} \vdash \texttt{let}\ \ x = N\ \ \texttt{in}\ \ M : T_2}\ \texttt{[tp-let]}$$

Definición 79: Generación de restricciones polimórficas

$$\frac{\bar{\Gamma}(x) = \mathcal{E}}{\bar{\Gamma} \models x : \texttt{finst}(\mathcal{E}) \mid \varnothing}\ \texttt{[p-var]}$$

$$\frac{\dot{c} \in \mathcal{C}_{\mathcal{P}}}{\bar{\Gamma} \models \dot{c} : \mathcal{C}_{\mathcal{T}}(\dot{c}) \mid \varnothing}\ \texttt{[p-cte]}$$

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \bar{\Gamma}, x{:}\forall.\mathtt{Z} \models M{:}T \mid \mathcal{R}}{\bar{\Gamma} \models \lambda x.M : \mathtt{Z} \to T \mid \mathcal{R}}\ \texttt{[p-}\lambda\texttt{x]}$$

$$\frac{\bar{\Gamma} \models M{:}T_1 \mid \mathcal{R}_1 \quad \bar{\Gamma} \models N{:}T_2 \mid \mathcal{R}_2 \quad \mathtt{Z} \in \mathbb{N}_{\mathbb{V}}}{\bar{\Gamma} \models M\ N : \mathtt{Z} \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \{T_1 = T_2 \to \mathtt{Z}\}}\ \texttt{[p-app]}$$

$$\frac{\bar{\Gamma} \models M{:}T_1 \mid \mathcal{R}_1 \quad \bar{\Gamma} \models N{:}T_2 \mid \mathcal{R}_2 \quad \bar{\Gamma} \models L{:}T_3 \mid \mathcal{R}_3}{\bar{\Gamma} \models \texttt{if}\ M\ N\ L : T_2 \mid \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3 \cup \{T_1 = \texttt{Bool}, T_2 = T_3\}}\ \texttt{[p-if]}$$

Definición 79: Generación de restricciones polimórficas (cont.)

$$\frac{\mathtt{Z} \in \mathbb{N}_{\mathbb{V}} \quad \bar{\Gamma} \models M : T \mid_{\mathcal{R}}}{\bar{\Gamma} \models \mathtt{Fix}\ M : \mathtt{Z} \mid_{\mathcal{R} \cup \{T = \mathtt{Z} \to \mathtt{Z}\}}} \text{[p-fix]}$$

$$\frac{\begin{array}{c}\bar{\Gamma}' \models N : T_1' \mid_{\mathcal{R}_1} \\ \mu = \mathcal{U}(\mathcal{R}_1) \\ \bar{\Gamma} = \mu(\bar{\Gamma}') \\ T_1 = \mu(T_1') \\ \bar{\Gamma}, x : \mathtt{gen}(\bar{\Gamma}, T_1) \models M : T_2 \mid_{\mathcal{R}_2}\end{array}}{\bar{\Gamma} \models \mathtt{let}\ \ x = N\ \ \mathtt{in}\ \ M : T_2 \mid_{\mathcal{R}_2}} \text{[p-let]}$$

Apéndice D

Máquinas abstractas

En este apéndice incluimos las definiciones ampliadas de las máquinas evaluadoras desarrolladas en el Capítulo 7.

Definición 80: Máquina CC [15]

Definimos el comportamiento de la máquina CC mediante la siguiente tabla de transiciones donde para cada estado $\langle M, C\rangle$ inspeccionamos la forma de M y C para especificar el siguiente estado de la máquina. Para el caso de los contextos de evaluación nos fijaremos en la forma del *marco base* mediante la notación $C[(\square\ N)]$ a veces sustituiremos el marco base por uno distinto pero mantendremos el resto del contexto intacto como en la regla `[cc-fn]`, en otros casos eliminaremos el marco base del contexto y este lo sustituiremos por un agujero como en el caso de la regla `[cc-ar]`. Usamos la letra v para especificar que M debe ser un valor.

M	C	$\overset{\text{CC}}{\mapsto}$	M	C	
v	$C\ [(\square\ N)]$		N	$C[(v\ \square)]$	`[cc-fn]`
v	$C\ [((\lambda x.N)\ \square)]$		$N[x \leftarrow v]$	C	`[cc-ar]`
v	$C\ [o^{(n)}\ [v_1 \ldots \square\ M_{j+1} \ldots M_n]]$		M_{j+1}	$C[o^{(n)}[v_1 \ldots v\ \square\ M_{j+2} \ldots M_n]]$	`[cc-opv]`
v	$C\ [o^{(n)}\ [v_1 \ldots v_{n-1}\ \square]]$		$\delta(o^{(n)}, [v_i])$	C	`[cc-δ]`
v	$C\ [\texttt{if}\ \square\ N\ L]$		$\begin{cases} N & \text{si } v = \texttt{true} \\ L & \text{si } v = \texttt{false} \end{cases}$	C	`[cc-frk]`
$\lambda f.M$	$C\ [\texttt{Fix}\ \square]$		$M[f \leftarrow \texttt{Fix}\ \lambda f.M]$	C	`[cc-fix]`
$(M\ N)$	C		M	$C[(\square\ N)]$	`[cc-ap]`
$o^{(n)}[M_i]$	C		M_1	$C[o^{(n)}[\square\ M_2 \ldots M_n]]$	`[cc-op]`
$\texttt{if}\ M\ N\ L$	C		M	$C[\texttt{if}\ \square\ N\ L]$	`[cc-if]`
$\texttt{Fix}\ M$	C		M	$C[\texttt{Fix}\ \square]$	`[cc-fix-b]`
$\texttt{let}\ x = N\ \texttt{in}\ M$	C		$(\lambda x.M)\ N$	C	`[cc-let]`
v	$\square$		*estado final*		`[cc-fin]`

Definición 81: La máquina CK [15]

Definimos la máquina CK como se muestra en la tabla de transiciones, para estados de la forma $\langle M, \kappa \rangle$ donde $M \in \Lambda_p$ y $\kappa \in K$. El estado inicial de la máquina es $\langle M, \texttt{Mt} \rangle$.

M	κ	$\underset{\text{CK}}{\mapsto}$	M	κ	
v	$\texttt{Ar}\ N\ \kappa$		N	$\texttt{Fn}\ v\ \kappa$	[ck-fn]
v	$\texttt{Fn}\ (\lambda x.M)\ \kappa$		$M[x \leftarrow v]$	κ	[ck-ar]
v	$\texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \kappa$		M_{j+1}	$\texttt{Op}\ [v_i]_{j}^{1}\ o^{(n)}\ [M_i]_{j+2}^{n}\ \kappa$	[ck-opv]
v	$\texttt{Op}\ [v_i]_{n-1}^{1}\ o^{(n)}\ []\ \kappa$		$\delta(o^{(n)}, [v_i]_1^n)$	κ	[ck-δ]
v	$\texttt{If}\ N\ L\ \kappa$		$\begin{cases} N & \text{si } v = \texttt{true} \\ L & \text{si } v = \texttt{false} \end{cases}$	κ	[ck-frk]
$\lambda f.M$	$\texttt{Fix}\ \kappa$		$M[f \leftarrow \texttt{Fix}\ \lambda f.M]$	κ	[ck-fix]
$(M\ N)$	κ		M	$\texttt{Ar}\ N\ \kappa$	[ck-ap]
$o^{(n)}[M_i]$	κ		M_1	$\texttt{Op}\ []\ o^{(n)}\ [M_i]_2^n\ \kappa$	[ck-op]
$\texttt{if}\ M\ N\ L$	κ		M	$\texttt{If}\ \ N\ L\ \kappa$	[ck-if]
$\texttt{Fix}\ M$	κ		M	$\texttt{Fix}\ \ \kappa$	[ck-fix-b]
$\texttt{let}\ x = N\ \ \texttt{in}\ \ M$	κ		$(\lambda x.M)\ N$	κ	[ck-let]
v	Mt		*estado final*		[ck-fin]

Definición 82: La máquina CEK [15]

Definimos el comportamiento de la máquina CEK que tiene estados de la forma $\langle M, \varepsilon, \kappa\rangle$ donde $\langle M, \varepsilon\rangle$ es una cerradura y $\kappa \in K_C$. El estado inicial de la máquina es $\langle M, \varnothing, \texttt{Mt}\,\rangle$.

M	ε	κ	$\underset{\text{CEK}}{\mapsto}$	M	ε	κ	
x	ε	κ		$\varepsilon_\Lambda(x)$	$\varepsilon_\varepsilon(x)$	κ	[cek-var]
v	ε	$\texttt{Ar}\ N\ \varepsilon'\ \kappa$		N	ε'	$\texttt{Fn}\ v\ \varepsilon\ \kappa$	[cek-fn]
v	ε	$\texttt{Fn}\ (\lambda x.M)\ \varepsilon'\ \kappa$		M	$\varepsilon'[x \mapsto \langle v,\, \varepsilon\rangle]$	κ	[cek-ar]
v	ε	$\texttt{Op}\ [v_i]_{n-1}^{1}\ o^{(n)}\ []\ \varepsilon'\ \kappa$		$\delta(o^{(n)}, [v_i]_1^n)$	$\varnothing$	κ	[cek-opv]
v	ε	$\texttt{Op}\ [v_i]_{j-1}^{1}\ o^{(n)}\ [M_i]_{j+1}^{n}\ \varepsilon'\ \kappa$		M_{j+1}	ε'	$\texttt{Op}\ [v_i]_j^1\ o^{(n)}\ [M_i]_{j+2}^n\ \varepsilon'\ \kappa$	[cek-δ]
v	ε	$\texttt{If}\ N\ L\ \varepsilon'\ \kappa$		$\begin{cases} N & \text{si } v = \texttt{true} \\ L & \text{si } v = \texttt{false} \end{cases}$	ε'	κ	[cek-frk]
$\lambda f.M$	ε	$\texttt{Fix}\ \kappa$		M	$\varepsilon[f \mapsto \langle \texttt{Fix}\ \lambda f.M,\, \varepsilon\rangle]$	κ	[cek-fix]
$(M\ N)$	ε	κ		M	ε	$\texttt{Ar}\ N\ \varepsilon\ \kappa$	[cek-ap]
$o^{(n)}[M_i]$	ε	κ		M_1	ε	$\texttt{Op}\ []\ o^{(n)}\ [M_i]_2^n\ \varepsilon\ \kappa$	[cek-op]
$\texttt{if}\ M\ N\ L$	ε	κ		M	ε	$\texttt{If}\ N\ L\ \varepsilon\ \kappa$	[cek-if]
$\texttt{Fix}\ M$	ε	κ		M	ε	$\texttt{Fix}\ \kappa$	[cek-fix-b]
$\texttt{let}\ x = N\ \texttt{in}\ M$	ε	κ		$(\lambda x.M)\ N$	ε	κ	[cek-let]
v	ε	$\texttt{Mt}$		*estado final*			[cek-fin]

Bibliografía

[1] Harold Abelson, Gerald Jay Sussman, y Julie Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996.

[2] Stephen Adams. Functional pearls efficient sets—a balancing act. *Journal of Functional Programming*, 3:553–561, 1993.

[3] Nada Amin y Ross Tate. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, páginas 838–848. ACM, 2016.

[4] Henk Barendregt. The impact of the lambda calculus in logic and computer science. *Bulletin of Symbolic Logic*, 3(02):181–215, 1997.

[5] HP Barendregt. The lambda calculus: Its syntax and semantics, number 103 in North-Holland studies in logic and the foundation of mathematics, 1981.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, y Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. *ACM SIGPLAN notices*, 33(10):183–200, 1998.

[7] Felice Cardone y J Roger Hindley. History of lambda-calculus and combinatory logic. *Handbook of the History of Logic*, 5:723–817, 2006.

[8] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[9] Alonzo Church. A proof of freedom from contradiction. *Proceedings of the National Academy of Sciences of the United States of America*, 21(5):275–281, 1935.

[10] Alonzo Church. *The calculi of lambda-conversion*. Number 6. Princeton University Press, 1941.

[11] Alonzo Church y J. B. Rosser. Some properties of conversion. *Trans. Amer. Math. Soc.*, 39(3):472–472, marzo 1936.

[12] Haskell Brooks Curry, Robert Feys, William Craig, y William Craig. Combinatory logic. 1972.

[13] Luis Damas. *Type assignment in programming languages*. PhD thesis, 1984.

[14] Luis Damas y Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 207–212. ACM, 1982.

[15] Matthias Felleisen, Robert Bruce Findler, y Matthew Flatt. *Semantics engineering with PLT Redex*. Mit Press, 2009.

[16] Matthias Felleisen y Daniel P. Friedman. Control operators, the SECD-machine and the lambda-calculus. páginas 193–217.

[17] J Roger Hindley y Jonathan P Seldin. *Lambda-calculus and combinators: an introduction*, volume 13. Cambridge University Press Cambridge, 2008.

[18] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[19] John E Hopcroft, Rajeev Motwani, y Jeffrey D Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[20] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, enero 1964.

[21] Peter John Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[22] Daan Leijen y Andriy Palamarchuk. Data.map.strict, 2008.

[23] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. no starch press, 2011.

[24] Simon Marlow et al. Haskell 2010 language report. *Available online http://www.haskell.org*, 2010.

[25] Alberto Martelli y Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[26] Robin Milner. *The definition of standard ML: revised*. MIT press, 1997.

[27] Eugenio Moggi. Notions of computation and monads. *Information and computation*, 93(1):55–92, 1991.

[28] J. Nievergelt y E. M. Reingold. Binary search trees of bounded balance. In *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, STOC '72, páginas 137–142, New York, NY, USA, 1972. ACM.

[29] Martin Odersky y Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, páginas 146–159. ACM, 1997.

[30] Oracle. JDK 5.0 Java programming language-related APIs & developer guides, 2004.

[31] Bryan O'Sullivan, John Goerzen, y Donald Bruce Stewart. *Real world Haskell: Code you can believe in*. "O'Reilly Media, Inc.", 2008.

[32] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[33] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical computer science*, 1(2):125–159, 1975.

[34] Cenobio Moisés Vázquez Reyes. Mónadas en la programación funcional: una prueba formal de su equivalencia con las ternas de Kleisli. Tesis para obtener el grado de licenciatura, 2016.

[35] J Alan Robinson. Computational logic: The unification computation. *Machine intelligence*, 6(63-72):10–1, 1971.

[36] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[37] J. B. Rosser S. C. Kleene. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935.

[38] M. Schönfinkel. *Über die Bausteine der mathematischen Logik*. publisher not identified, 1924.

[39] Philip Wadler. Comprehending monads. *Mathematical structures in computer science*, 2(04):461–493, 1992.

[40] Philip Wadler. Monads for functional programming. In *International School on Advanced Functional Programming*, páginas 24–52. Springer, 1995.