Facultad de Ciencias Exactas,
Ingeniería y Agrimensura

Departamento de Ciencias de la Computación

Formalización de Sistema I con tipo Top

Autor
Agustin Francisco Settimo

Directora
Cecilia Manzino

Co-Director
Cristian Sottile

Noviembre 2023

Índice general

1. Introducción **1**
- 1.1. Motivación . . . 1
- 1.2. Trabajo previo . . . 2
- 1.3. Estructura del trabajo . . . 2

2. Preliminares **4**
- 2.1. Sistemas módulo isomorfismos . . . 4
  - 2.1.1. Isomorfismos de tipos . . . 4
  - 2.1.2. Sistema I . . . 6
  - 2.1.3. SIP . . . 10
  - 2.1.4. λ^+ . . . 12
- 2.2. Tipos dependientes . . . 13
  - 2.2.1. Cubo Lambda . . . 14
- 2.3. Proposiciones como Tipos . . . 15
- 2.4. Teoría de Tipos Intuicionista . . . 16
- 2.5. Agda . . . 17
- 2.6. Indices de De Bruijn . . . 17
- 2.7. Substituciones explícitas . . . 18

3. Aportes **20**
- 3.1. Formalización . . . 20
  - 3.1.1. Tipos intrínsecos . . . 20
  - 3.1.2. Cálculo lambda con pares y tipo Top . . . 20
  - 3.1.3. Isomorfismo de tipos . . . 27
- 3.2. Equivalencia de términos . . . 28
  - 3.2.1. Preservación de tipos . . . 33
- 3.3. Progreso . . . 33
  - 3.3.1. Formas normales, neutrales y valores . . . 33
  - 3.3.2. Estrategia de reducción . . . 35
- 3.4. Evaluación . . . 39
- 3.5. Normalización Fuerte . . . 41
  - 3.5.1. Prueba para STLC con pares y Top . . . 42
  - 3.5.2. Prueba para Sistema I con tipo Top . . . 48
  - 3.5.3. Relación bien fundada . . . 54

4. Conclusiones **57**
4.1. Trabajo futuro 57
4.1.1. Inferencia de tipos 57
4.1.2. Formalización SIP 58

Resumen

Los sistemas de tipos desempeñan un papel esencial en la garantía del comportamiento adecuado de los programas. Sin embargo, estos imponen cierta rigidez a la hora de escribir dichos programas, ya que se deben adherir estrictamente a las reglas de tipado establecidas. Existen una serie de "sistemas módulo isomorfismos" [DD19; DD20; AD20; DL15; Sot20; SDL21] que proponen flexibilizar esta restricción considerando aquellos tipos que sean isomorfos como idénticos. En simples palabras, dos tipos se consideran isomorfos cuando es posible convertir uno en el otro, y viceversa, sin perder información en el proceso. Es decir, si dos tipos A y B son isomorfos, es posible usar un término de tipo A en cualquier lugar que se espere un término de tipo B aplicando la conversión correspondiente. Trabajar con tipos isomorfos provee mayor flexibilidad en la construcción de programas y permite combinarlos de formas que normalmente no serían posibles.

Este trabajo se centra principalmente en el primero de estos sistemas, llamado Sistema I. Aquí, se presenta una formalización de una adaptación de dicho sistema en el lenguaje Agda, además, se añade el tipo Top y los isomorfismos correspondientes para ampliar aún más la flexibilidad del sistema. Esta formalización va acompañada de pruebas formales que demuestran el cumplimiento de diversas propiedades, siendo la normalización fuerte una de las más cruciales. Es importante destacar que la prueba de normalización fuerte utiliza la técnica de *logical relations* pero con predicados distintos a los presentados en la prueba de candidatos de reducibilidad de Girard.

En esta tesina no solo se exploran los sistemas módulo isomorfismos sino también, los intricados mecanismos que sirven de motor para muchas implementaciones de cálculo lambda, como lo son las substituciones explícitas, que desempeñan un papel fundamental en la manipulación de términos dentro del sistema. Además, se exponen los fundamentos de la correspondencia de Curry-Howard, la cual explica la estrecha relación entre la programación y la lógica. Esta correspondencia permitirá comprender la doble mirada que propone Sistema I, por un lado, puede ser visto como un lenguaje de programación y por el otro como un sistema de pruebas. Coincidentemente, el isomorfismo de Curry-Howard es lo que dio origen a los asistentes de pruebas como Coq y Agda. Siendo este último el lenguaje escogido para realizar la formalización presentada en este trabajo.

Capítulo 1

Introducción

1.1. Motivación

Los sistemas de tipos permiten especificar el comportamiento de los programas. Estas especificaciones aseguran que los programas se comportarán de una forma esperada, pero al mismo tiempo imponen rigidez a la hora de escribirlos. Por ejemplo, el tipo $(A \times B) \to C$ representa los programas que permiten obtener un valor de tipo C a partir de un par de valores de tipo A y B. Mientas que, por otro lado, el tipo $A \to B \to C$ representa los programas que primero aceptan un valor de tipo A, luego uno de tipo B y finalmente devuelven un valor de tipo C. A pesar de que estos dos tipos son distintos en su forma, representan la misma idea, es decir, poseen el mismo significado desde el punto de vista de la utilidad e información que aportan.

Esto se puede demostrar usando las funciones *curry* y *uncurry*, que permiten transformar una función en su forma currificada y viceversa, sin pérdida de información en el proceso. Cuando ambos tipos contienen la misma cantidad de información, se dice que son isomorfos, en este caso, los tipos $(A \times B) \to C$ y $A \to B \to C$ del primer ejemplo, están relacionados por el proceso de currificación.

Un caso más simple que ilustra este concepto es el de los pares $A \times B$ y $B \times A$, resulta fácil notar que es posible transformar uno en el otro y viceversa sin perder información, ya que desde el punto de vista de la utilidad, cada uno aporta un elemento de tipo A y un elemento de tipo B, el orden no es relevante.

Al mismo tiempo, la correspondencia de Curry-Howard [SU06] define una estrecha relación entre la programación y la lógica, donde los tipos se corresponden con las fórmulas de la lógica. Debido a esto, se puede observar un comportamiento análogo cuando se trabaja con pruebas. Por ejemplo, una prueba de $(A \wedge B) \Rightarrow C$ no constituye una prueba de $A \Rightarrow B \Rightarrow C$, y viceversa, a pesar de que ambas tienen el mismo significado.

Programas con tipos isomorfos representan el mismo tipo de problema, por lo que tratarlos como si fueran idénticos tiene aplicaciones interesantes. Desde el punto de vista de los programas, nos permite construir expresiones que antes no eran posibles, por ejemplo, una función $f : (A \times B) \to C$ puede ser aplicada como $f\langle a, b\rangle$ ó $f\ a\ b$, es decir, es posible evadir cierta rigidez impuesta por el sistema de tipos. Si A y B son tipos isomorfos, se puede emplear una expresión de tipo A en cualquier lugar donde se espera una de tipo B. Mientras que desde el punto de vista de la lógica, los isomorfismos de fórmulas hacen que las pruebas sean más naturales, por ejemplo, para probar $(A \wedge (A \Rightarrow B)) \Rightarrow B$ usando deducción natural, deberíamos primero introducir la hipótesis $A \wedge (A \Rightarrow B)$, y luego descomponerla en A y $A \Rightarrow B$, en cambio, utilizando el isomorfismo de Curry, que también vale para la lógica, transformamos el objetivo en $A \Rightarrow (A \Rightarrow B) \Rightarrow B$ y

luego introducimos directamente las hipótesis A y $A \Rightarrow B$.

1.2. Trabajo previo

Se han realizado múltiples aportes sobre los denominados sistemas módulo isomorfismos. Estos sistemas utilizan el cálculo lambda simplemente tipado con pares como marco de trabajo, y lo extienden de forma que los tipos isomorfos se consideran idénticos. El más relevante para este trabajo es Sistema I [DD19], que se corresponde con el fragmento de la lógica que incorpora las conectivas $\Rightarrow$ y $\wedge$. Este sistema ignora las formas de los tipos y se centra solo en su utilidad final, permitiendo así realizar aplicaciones de funciones y proyecciones que no son posibles en el cálculo tradicional. Además, se presentan propiedades interesantes como el no determinismo y otras que pueden utilizarse en la optimización de programas. Desde el punto de vista de la lógica, el no determinismo es interesante, ya que funciona como una suerte de irrelevancia de pruebas. Sin embargo, desde el punto de vista de la programación es más útil que el comportamiento de los programas sea determinista. Afortunadamente, como se explicará en detalle más adelante, es posible recuperar el determinismo.

También, es importante destacar que todos los isomorfismos de tipos que se presentan en estos sistemas, fueron caracterizados y agrupados en conjuntos axiomáticos por Di Cosmo [Di 05].

En términos de expresividad, un siguiente paso natural para un cálculo lambda simplemente tipado consiste en la adición de polimorfismo, esto es justamente lo que propone Sistema I Polimórfico [Sot20; SDL21]. Dicho trabajo incorpora el constructor $\forall$ a nivel de tipos, la abstracción y aplicación de tipos a nivel de términos, y algunos de los isomorfismos faltantes correspondientes al fragmento de la lógica con $\forall$, $\Rightarrow$ y $\wedge$.

Desde el punto de vista práctico, existe una implementación de Sistema I desarrollada en Haskell, denominada λ^+ [DL15]. Este sistema posee un sistema de reescritura módulo isomorfismos, por lo que su implementación no es trivial. Además, añade números naturales y recursión general. En este trabajo se puede observar la complejidad que trae implementar un lenguaje de estas características.

1.3. Estructura del trabajo

En el capítulo 2 se presentan los fundamentos previos necesarios para comprender los aportes realizados en esta tesina. Por un lado, se introducen los isomorfismos de tipos y se explican en detalle la sintaxis y semántica de Sistema I. Por otro lado, se introducen una serie de conceptos que abarcan desde tipos dependientes hasta la Teoría de Tipos Intuicionista, y culminan en la presentación de Agda, que será el lenguaje elegido para la formalización realizada en este trabajo. Además, se explicará de forma sucinta la representación de términos con índices de De Brujin, y el funcionamiento de las substituciones explícitas, ambos conceptos no solo serán una pieza central de la formalización aquí presentada, sino que también son comúnmente empleados en el desarrollo de muchas implementaciones de cálculo lambda.

Luego, en el capítulo 3 se presenta la formalización de Sistema I con tipo Top, que será el aporte principal de esta tesina. El orden de presentación del código y los fragmentos expuestos se escogieron con el objetivo de hacer la explicación más intuitiva y facilitar la comprensión de las pruebas. La propiedad de normalización fuerte se presenta en dos partes debido a su complejidad, donde primero se realiza la prueba para el cálculo lambda simplemente tipado con pares, y luego se extiende dicha prueba añadiendo isomorfismos de tipos. El código completo de toda la formalización puede obtenerse en `https://github.com/AgusSett/thesis`.

Por último, en el capítulo 4 se presentan las conclusiones y trabajo futuro que se derivan del presente trabajo de tesina.

Capítulo 2

Preliminares

2.1. Sistemas módulo isomorfismos

2.1.1. Isomorfismos de tipos

Para poder abstraernos de la forma de los programas y centrarnos en su significado, es necesario establecer cuáles son las formas que son combinables y cómo combinarlas. Para ello nos valdremos de la noción de isomorfismo entre tipos. El primer paso es considerar dos tipos A y B como isomorfos si existen dos programas $A \to B$ y $B \to A$, tal que al componerlos, en ambos sentidos, obtenemos como resultado la identidad. Por ejemplo, para probar que los tipos $(A \times B) \to C$ y $A \to B \to C$ son efectivamente isomorfos, se define un término *curry* que dada una función devuelve su forma currificada, y un término *uncurry* que realiza la operación inversa:

$$curry = \lambda x^{(A\times B)\to C}.\lambda a^{A}.\lambda b^{B}.x\langle a, b\rangle$$
$$uncurry = \lambda x^{A\to B\to C}.\lambda y^{A\times B}.x(\pi_1 y)(\pi_2 y)$$

Luego, se demuestra que para cualquier par $\langle u, v\rangle$ y función f se cumple:

$$\begin{aligned}
& (uncurry \circ curry) f\langle u, v\rangle \\
= \\
& uncurry(curry\ f)\langle u, v\rangle \\
= \\
& uncurry((\lambda x^{(A\times B)\to C}.\lambda a^{A}.\lambda b^{B}.x\langle a, b\rangle) f)\langle u, v\rangle \\
\hookrightarrow \\
& uncurry(\lambda a^{A}.\lambda b^{B}.f\langle a, b\rangle)\langle u, v\rangle \\
= \\
& (\lambda x^{A\to B\to C}.\lambda y^{A\times B}.x(\pi_1 y)(\pi_2 y))(\lambda a^{A}.\lambda b^{B}.f\langle a, b\rangle)\langle u, v\rangle \\
\hookrightarrow \\
& (\lambda y^{A\times B}.(\lambda a^{A}.\lambda b^{B}.f\langle a, b\rangle)(\pi_1 y)(\pi_2 y))\langle u, v\rangle \\
\hookrightarrow \\
& (\lambda a^{A}.\lambda b^{B}.f\langle a, b\rangle)(\pi_1\langle u, v\rangle)(\pi_2\langle u, v\rangle)
\end{aligned}$$

$$
\begin{array}{l}
\hookrightarrow \\
\quad (\lambda a^A.\lambda b^B.f\langle a,b\rangle)u\ v \\
\hookrightarrow \\
\quad f\langle u,v\rangle
\end{array}
$$

Luego, se concluye que $uncurry \circ curry = id_{(A\times B)\to C}$ por extensionalidad. De forma similar se puede probar que $curry \circ uncurry = id_{A\to B\to C}$

Di Cosmo [Di 05] caracterizó para Sistema F con pares y tipo top, los conjuntos mínimos de isomorfismos que permiten construir todos los demás a partir de reglas de congruencia y transitividad. Se escribe $A \equiv B$ cuando A y B son isomorfos.

$$
\begin{array}{lll}
\text{swap}\ \ A \to (B \to C) \equiv B \to (A \to C) & \Big\} & Th^1 \\
\left.\begin{array}{ll}
1.\ A \times B \equiv B \times A & \\
2.\ A \times (B \times C) \equiv (A \times B) \times C & \\
3.\ (A \times B) \to C \equiv A \to (B \to C) & \\
4.\ A \to (B \times C) \equiv (A \to B) \times (A \to C) & \\
5.\ A \times \mathbf{T} \equiv A & \\
6.\ A \to \mathbf{T} \equiv \mathbf{T} & \\
7.\ \mathbf{T} \to A \equiv A &
\end{array}\right\} & Th^1_{\times T} & \\
\left.\begin{array}{ll}
8.\ \forall X.\forall Y.A \equiv \forall Y.\forall X.A & \\
9.\ \forall X.A \equiv \forall Y.A[Y/X] & \\
10.\ \forall X.(A \to B) \equiv A \to \forall X.B &
\end{array}\right\} & + \text{swap} = Th^2 & \\
11.\ \forall X.(A \times B) \equiv \forall X.A \times \forall X.B & & \\
12.\ \forall X.\mathbf{T} \equiv \mathbf{T} & & \Big\} Th^2_{\times T}
\end{array}
$$

Cuadro 2.1: Isomorfismos de tipo en cálculo lambda tipado

Estos isomorfismos están agrupados en distintas categorías. Por ejemplo, la categoría Th^1 corresponde al cálculo lambda simplemente tipado, dentro de esta, solo existe el isomorfismo *swap*. Si se añaden pares al cálculo, se obtienen los isomorfismos 1, 2, 3 y 4 de la tabla, dicho conjunto se denomina $Th^1_\times$, notar que no se incluye *swap*, ya que es posible construirlo a partir de 1 y 3. Luego, se puede obtener el conjunto $Th^1_{\times T}$ añadiendo el tipo top al cálculo.

Por otro lado, las categorías Th^2 corresponden al cálculo lambda de segundo orden, también llamado Sistema F. Este conjunto abarca los isomorfismos que se pueden construir usando el conector $\forall$, además del isomorfismo *swap*. De forma análoga, añadiendo pares y el tipo top a Sistema F se obtienen las categorías $Th^2_\times$ y $Th^2_{\times T}$ respectivamente.

2.1.2. Sistema I

Sistema I [DD19] es un cálculo lambda simplemente tipado con pares donde los tipos isomorfos son considerados iguales. Su gramática para tipos es la misma que en el cálculo lambda tradicional, mientras que su gramática para términos difiere únicamente en la proyección, que se da con respecto al tipo y no a la posición:

$$A := \tau \mid A \to A \mid A \times A$$
$$t := x \mid \lambda x^A.t \mid tt \mid \langle t, t\rangle \mid \pi_A t$$

Las reglas de tipado estarán dadas por las del cálculo lambda simplemente tipado con pares, con la modificación de que la eliminación de la conjunción se produce respecto al tipo y no a la posición. Además, se añade la definición de una nueva regla de tipado llamada $(\equiv)$

$$\frac{}{\Gamma, x : A \vdash x : A}\ (ax) \qquad \frac{A \equiv B \qquad \Gamma \vdash r : A}{\Gamma \vdash r : B}\ (\equiv)$$

$$\frac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x.r : A \to B}\ (\Rightarrow_i) \qquad \frac{\Gamma \vdash r : A \to B \qquad \Gamma \vdash s : A}{\Gamma \vdash rs : B}\ (\Rightarrow_e)$$

$$\frac{\Gamma \vdash r : A \qquad \Gamma \vdash s : B}{\Gamma \vdash \langle r, s\rangle : A \times B}\ (\times_i) \qquad \frac{\Gamma \vdash r : A \times B}{\Gamma \vdash \pi_A(r) : A}\ (\times_e)$$

Figura 2.1: Reglas de tipado

El conjunto de isomorfismos que abarca este sistema es denominado $Th^1_\times$. A continuación se presentan los cuatro isomorfismos que pertenecen a dicho conjunto axiomático.

$$A \times B \equiv B \times A \qquad (\text{COMM})$$
$$A \times (B \times C) \equiv (A \times B) \times C \qquad (\text{ASSO})$$
$$(A \times B) \to C \equiv A \to (B \to C) \qquad (\text{CURRY})$$
$$A \to (B \times C) \equiv (A \to B) \times (A \to C) \qquad (\text{DIST})$$

$$\frac{}{A \equiv A} \qquad \frac{A \equiv B}{B \equiv A} \qquad \frac{A \equiv B \qquad B \equiv C}{A \equiv C}$$

$$\frac{A \equiv B}{A \times C \equiv B \times C} \qquad \frac{A \equiv B}{C \times A \equiv C \times B} \qquad \frac{A \equiv B}{A \to C \equiv B \to C} \qquad \frac{A \equiv B}{C \to A \equiv C \to B}$$

Figura 2.2: Isomorfismos de tipo en Sistema I

La regla $(\equiv)$ permite cambiar el tipo de un término por otro isomorfo a este. Por ejemplo:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : C \vdash r : A}{\Gamma \vdash \lambda x.r : C \to A}\ (\Rightarrow_i) \qquad \dfrac{\Gamma, x : C \vdash s : B}{\Gamma \vdash \lambda x.s : C \to B}\ (\Rightarrow_i)}{\Gamma \vdash \langle \lambda x.r, \lambda x.s\rangle : (C \to A) \times (C \to B)}\ (\times_i)}{\Gamma \vdash \langle \lambda x.r, \lambda x.s\rangle : C \to (A \times B)}\ (\equiv) \qquad \Gamma \vdash t : C}{\Gamma \vdash \langle \lambda x.r, \lambda x.s\rangle\ t : A \times B}\ (\Rightarrow_e)$$

La introducción de una relación de equivalencia en el nivel de los tipos, necesariamente deberá tener consecuencias en el nivel de los términos. Con las reglas de reducción usuales, un término como el del ejemplo anterior estaría en forma normal, cuando en realidad no lo está, ya que contiene un β-redex. Entonces, el siguiente paso es extender la relación de reducción con isomorfismos a nivel de términos. Estas nuevas reglas permiten desatascar los términos que tipamos usando $(\equiv)$. Por ejemplo, es necesario definir la siguiente equivalencia de términos $\langle \lambda x^A.r, \lambda x^A.s\rangle \rightleftarrows \lambda x^A.\langle r, s\rangle$ para reducir el programa del ejemplo anterior:

$$\begin{array}{l} \quad \langle \lambda x.r, \lambda x.s\rangle\ t \\ \rightleftarrows \\ \quad (\lambda x.\langle r, s\rangle)\ t \\ \hookrightarrow \\ \quad \langle r[t/x], s[t/x]\rangle \end{array}$$

A continuación, se presentan las reglas propuestas por Sistema I, las cuales fueron elegidas con el objetivo de obtener un cálculo fuertemente normalizante y consistente:

$$\langle r, s\rangle \rightleftarrows \langle s, r\rangle \qquad (\textsc{comm})$$
$$\langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle \qquad (\textsc{asso})$$
$$\lambda x^A.\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s\rangle \qquad (\textsc{dist}_\lambda)$$
$$\langle r, s\rangle t \rightleftarrows \langle rt, st\rangle \qquad (\textsc{dist}_{app})$$
$$r\langle s, t\rangle \rightleftarrows rst \qquad (\textsc{curry})$$

$$\frac{t \rightleftarrows r}{s \rightleftarrows t} \qquad \frac{t \rightleftarrows r}{\lambda x.t \rightleftarrows \lambda x.r} \qquad \frac{t \rightleftarrows r}{ts \rightleftarrows rs} \qquad \frac{t \rightleftarrows r}{st \rightleftarrows sr}$$

$$\frac{t \rightleftarrows r}{\langle t, s\rangle \rightleftarrows \langle r, s\rangle} \qquad \frac{t \rightleftarrows r}{\langle s, t\rangle \rightleftarrows \langle s, r\rangle} \qquad \frac{t \rightleftarrows r}{\pi_A t \rightleftarrows \pi_A r}$$

Figura 2.3: Reglas de equivalencia entre términos

Luego, se escribe $\rightleftarrows^*$ para denotar la clausura reflexiva y transitiva de la relación $\rightleftarrows$. Notar que $\rightleftarrows^*$ es una relación de equivalencia entre términos.

Es importante destacar que dependiendo del conjunto de equivalencias entre términos que se incluyen, se obtienen sistemas de cálculo con distintas propiedades, incluir demasiados puede desencadenar en no terminación, y si se incluyen pocos pueden aparecer eliminaciones en formas normales. Por ejemplo, el término $(\lambda x^A.\lambda y^B.r)\, s$ donde $s : B$ contiene una eliminación, sin embargo, no es posible reducirlo en Sistema I. Pero si se añade la regla "Si $t : A \to B$ entonces

$t \hookrightarrow_\eta \lambda x^A.tx$" entonces es posible derivar:

$$\begin{aligned}
&(\lambda x^A.\lambda y^B.r)\, s \\
\hookrightarrow_\eta \\
&\lambda z^A.((\lambda x^A.\lambda y^B.r)\, s\, z) \\
\rightleftarrows_{\text{CURRY}} \\
&\lambda z^A.((\lambda x^A.\lambda y^B.r)\, \langle s, z\rangle) \\
\rightleftarrows_{\text{COMM}} \\
&\lambda z^A.((\lambda x^A.\lambda y^B.r)\, \langle z, s\rangle) \\
\rightleftarrows_{\text{CURRY}} \\
&\lambda z^A.((\lambda x^A.\lambda y^B.r)\, z\, s) \\
\hookrightarrow_\beta \\
&\lambda z^A.((\lambda x^A.\lambda y^B.r[z/x])\, s)
\end{aligned}$$

En [DD19] se conjetura que la extensión de Sistema I con una regla de η-expansión produce un sistema donde no existen eliminaciones en formas normales, y dicha extensión se propone como un posible trabajo futuro.

El siguiente paso surge de notar que las nuevas reglas introducen problemas en la reducción clásica. Por un lado, las reglas de proyección π_1 y π_2 acceden al par a través de la posición de los elementos, pero el isomorfismo COMM permite cambiar el orden de un par, permitiendo así proyectar cualquiera de los dos elementos:

$$\begin{aligned}
&\pi_1\langle r, s\rangle \hookrightarrow_{\pi_1} r \\
&\pi_1\langle r, s\rangle \rightleftarrows_{\text{COMM}} \pi_1\langle s, r\rangle \hookrightarrow_{\pi_1} s
\end{aligned}$$

Esto supone un problema para la preservación de tipos y además introduce no determinismo. La solución es acceder al elemento a través de su tipo, por lo que se define una nueva regla:

$$\text{si } r : A \quad \pi_A\langle r, s\rangle \hookrightarrow_\pi r$$

Una consecuencia importante es que esta regla resuelve la preservación de tipos, pero mantiene el no determinismo en el cálculo. Si r y s tienen tipo A, entonces $\pi_A\langle r, s\rangle$ puede reducir indistintamente a cualquiera de los dos términos. Sin embargo, es posible codificar una proyección determinista incluso cuando ambos términos son del mismo tipo, por ejemplo, el par $\langle r, s\rangle : A \times A$ se codifica como:

$$\langle \lambda x^{\mathbb{X}}.r, \lambda x^{\mathbb{Y}}.s\rangle : \mathbb{X} \to A \times \mathbb{Y} \to A$$

Y la proyección $\pi_1\langle r, s\rangle$ se codifica como:

$$(\pi_{\mathbb{X}\to A}\langle \lambda x^{\mathbb{X}}.r, \lambda x^{\mathbb{Y}}.s\rangle)\ y^{\mathbb{X}}$$

Donde $\mathbb{X}$ y $\mathbb{Y}$ son dos tipos distintos, e $y^{\mathbb{X}}$ es cualquier elemento de tipo $\mathbb{X}$. De forma análoga se puede codificar π_2.

La β-reducción clásica también entra en conflicto con la preservación de tipos. Por ejemplo, el término $(\lambda x^A.\lambda y^B.r)ts$ donde $t : B$ y $s : A$ está bien tipado, pero para reducirlo correctamente primero se deben aplicar equivalencias para intercambiar el orden de t y s. Sin embargo, no hay nada que impida aplicar la regla β, lo cual podría terminar reduciendo a un tipo equivocado.

Para solucionar esto, se modifica la β-reducción y se añade una condición que requiere que el término que se está aplicando tenga el mismo tipo que el argumento de la abstracción:

$$\text{si } s : A \quad (\lambda x^A.r)s \hookrightarrow_\beta r[s/x]$$

Por último se define la relación de reducción módulo isomorfismos:

$$\rightsquigarrow := \rightleftarrows^* \circ \hookrightarrow \circ \rightleftarrows^*$$

Por lo tanto, $r \rightsquigarrow s \iff r \rightleftarrows^* r' \hookrightarrow s' \rightleftarrows^* s$ para cierto r' y s'. Es decir, que primero los términos son transformados a una forma equivalente donde sea posible aplicar las eliminaciones, y luego se aplican las β-reducciones y proyecciones.

Ejemplos

El término $\lambda x^A.\langle r, s\rangle$ tiene tipo $A \to (B \times C)$ aplicando el isomorfismo DIST se obtiene el tipo $(A \to B) \times (A \to C)$. Es decir, que una función que recibe un argumento y retorna un par, puede ser vista como un par de funciones. Por lo tanto, es posible proyectar la función antes de aplicarla, y dado $a : A$, se puede aplicar la función proyectada $\pi_{A\to B}(\lambda x^A.\langle r, s\rangle)\, a$

A continuación se detalla la forma de reducir dicho término:

$$\begin{aligned}
&\pi_{A\to B}(\lambda x^A.\langle r, s\rangle)\, a \\
\rightleftarrows_{\text{DIST}_\lambda}\ &\pi_{A\to B}(\langle \lambda x^A.r, \lambda x^A.s\rangle)\, a \\
\hookrightarrow_\pi\ &(\lambda x^A.r)\, a \\
\hookrightarrow_\beta\ &r[a/x]
\end{aligned}$$

El término $\lambda x^A.\lambda y^B.r$ tiene tipo $A \to B \to C$ aplicando los isomorfismos CURRY y COMM se obtiene el tipo $B \to A \to C$:

$$\begin{aligned}
&A \to B \to C \\
\equiv_{\text{CURRY}}\ &(A \times B) \to C \\
\equiv_{\text{COMM}}\ &(B \times A) \to C \\
\equiv_{\text{CURRY}}\ &B \to A \to C
\end{aligned}$$

Es decir que, dada una función con múltiples argumentos es posible aplicarlos en cualquier orden. Dados $a : A$ y $b : B$ la aplicación $(\lambda x^A.\lambda y^B.r)\, b\, a$ es válida.

A continuación se detalla la forma de reducir dicho término:

$$
\begin{aligned}
&(\lambda x^A.\lambda y^B.r)\, b\, a \\
\rightleftarrows_{\text{CURRY}}\ &(\lambda x^A.\lambda y^B.r)\, \langle b, a\rangle \\
\rightleftarrows_{\text{COMM}}\ &(\lambda x^A.\lambda y^B.r)\, \langle a, b\rangle \\
\rightleftarrows_{\text{CURRY}}\ &(\lambda x^A.\lambda y^B.r)\, a\, b \\
\hookrightarrow_{\beta}\ &(\lambda y^B.r[a/x])\, b \\
\hookrightarrow_{\beta}\ &r[a/x, b/y]
\end{aligned}
$$

2.1.3. SIP

Sistema I Polimórfico [Sot20; SDL21], es una extensión que agrega polimorfismo a Sistema I, toma algunos de los isomorfismos del conjunto axiomático $Th^2_\times$. A la gramática de tipos y términos definida en Sistema I, le agrega la gramática de tipos y términos correspondientes a Sistema F con pares. Sus tipos y términos son entonces los siguientes:

$$A := X \mid A \to A \mid A \times A \mid \forall X.A$$
$$t := x \mid \lambda^A x.t \mid tt \mid \langle t, t\rangle \mid \pi_A t \mid \Lambda X.t \mid t[A]$$

Las reglas de tipado de este cálculo surgen de tomar las reglas de Sistema I y agregar las correspondientes a polimorfismo de Sistema F:

$$\frac{\Gamma \vdash r : A \qquad X \notin FTV(\Gamma)}{\Gamma \vdash \Lambda X.r : \forall X.A}\ (\forall_i) \qquad \frac{\Gamma \vdash r : \forall X.A}{\Gamma \vdash r[B] : A[B/X]}\ (\forall_e)$$

Donde $FTV(\Gamma)$ es el conjunto de variables de tipo libres en el contexto Γ.
Los isomorfismos de carácter axiomático que se considerarán en este sistema son los siguientes:

$$\forall X.(A \to B) \equiv A \to \forall X.B \qquad \text{(P-COMM)}$$
$$\forall X.(A \times B) \equiv \forall X.A \times \forall X.B \qquad \text{(P-DIST)}$$

Del mismo modo que Sistema I permite proyectar abstracciones de términos, en SIP es posible proyectar abstracciones de tipos. Por ejemplo, utilizando los isomorfismos (DIST) y (P-DIST) se puede construir el siguiente término:

$$\pi_{\forall X.(A\to B)}(\Lambda X.\lambda x^A.\langle r, s\rangle) : \forall X.(A \to B)$$

Su derivación de tipos es:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash r : B \quad \Gamma \vdash s : C}{\Gamma \vdash \langle r, s\rangle : B \times C}\,(\times_i)}{\Gamma \vdash \lambda x^A.\langle r, s\rangle : A \to (B \times C)}\,(\Rightarrow_i)}{\Gamma \vdash \Lambda X.\lambda x^A.\langle r, s\rangle : \forall X.(A \to (B \times C))}\,(\forall_i)}{\Gamma \vdash \Lambda X.\lambda x^A.\langle r, s\rangle : \forall X.(A \to B) \times \forall X.(A \to C)}\,(\equiv)}{\Gamma \vdash \pi_{\forall X.(A\to B)}(\Lambda X.\lambda x^A.\langle r, s\rangle) : \forall X.(A \to B)}\,(\times_e)$$

Una forma de reducir este término es la siguiente:

$$\begin{aligned}
&\pi_{\forall X.(A\to B)}(\Lambda X.\lambda x^A.\langle r, s\rangle)\\
\rightleftarrows\\
&\pi_{\forall X.(A\to B)}(\Lambda X.\langle \lambda x^A.r, \lambda x^A.s\rangle)\\
\rightleftarrows\\
&\pi_{\forall X.(A\to B)}(\langle \Lambda X.\lambda x^A.r, \Lambda X.\lambda x^A.s\rangle)\\
\hookrightarrow_{\pi}\\
&\Lambda X.\lambda x^A.r
\end{aligned}$$

Este trabajo presenta en detalle la técnica utilizada para obtener las equivalencias de términos inducidas por un isomorfismo de tipos. Dado un isomorfismo de tipos con dos constructores involucrados, se construyen términos aplicando de diferentes maneras las reglas de tipado relacionadas con los constructores. Para cualquier par de conectivas, tenemos cuatro posibles formas de combinar sus reglas de tipado: introducción-introducción, introducción-eliminación, eliminación-introducción, y eliminación-eliminación. Cada una induce una equivalencia, y cada lado de la equivalencia estará dado por el orden en que se apliquen las conectivas.

Por ejemplo, el isomorfismo $A \to (B \times C) \equiv (A \to B) \times (A \to C)$ emplea los conectores $\to$ y $\times$. SIP incluye solamente dos de las cuatro equivalencias de términos. Las siguientes derivaciones prueban que ambos términos tienen los tipos involucrados en el isomorfismo de tipo dado:

- Introducción de $\to$ e introducción de $\times$: $\lambda x.\langle r, s\rangle \rightleftarrows \langle \lambda x.r, \lambda x.s\rangle$

$$\dfrac{\dfrac{\Gamma, x : A \vdash r : B \quad \Gamma, x : A \vdash s : C}{\Gamma, x : A \vdash \langle r, s\rangle : B \times C}\,(\times_i)}{\Gamma \vdash \lambda x.\langle r, s\rangle : A \to B \times C}\,(\Rightarrow_i)$$

$$\dfrac{\dfrac{\dfrac{\Gamma, x : A \vdash r : B}{\Gamma \vdash \lambda x.r : A \to B}\,(\Rightarrow_i) \quad \dfrac{\Gamma, x : A \vdash s : C}{\Gamma \vdash \lambda x.s : A \to C}\,(\Rightarrow_i)}{\Gamma \vdash \langle \lambda x.r, \lambda x.s\rangle : (A \to B) \times (A \to C)}\,(\times_i)}{\Gamma \vdash \langle \lambda x.r, \lambda x.s\rangle : A \to (B \times C)}\,(\equiv)$$

- Eliminación de $\to$ e introducción de $\times$: $\langle r, s\rangle t \rightleftarrows \langle rt, st\rangle$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash r : A \to B \quad \Gamma \vdash s : A \to C}{\Gamma \vdash \langle r, s\rangle : (A \to B) \times (A \to C)}\,(\times_i)}{\Gamma \vdash \langle r, s\rangle : A \to (B \times C)}\,(\equiv) \qquad \Gamma \vdash t : A}{\Gamma \vdash \langle r, s\rangle t : B \times C}\,(\Rightarrow_e)$$

$$\dfrac{\dfrac{\Gamma \vdash r : A \to B \quad \Gamma \vdash t : A}{\Gamma \vdash rt : B}\ (\Rightarrow_e) \quad \dfrac{\Gamma \vdash s : A \to C \quad \Gamma \vdash t : A}{\Gamma \vdash st : C}\ (\Rightarrow_e)}{\Gamma \vdash \langle rt, st\rangle : B \times C}\ (\times_i)$$

2.1.4. λ^+

λ^+ [DL15] es una implementación en Haskell de una extensión que agrega números naturales y recursión general a Sistema I. Este trabajo muestra las dificultades de implementar un sistema de tipos módulo isomorfismos y las aplicaciones prácticas de un lenguaje de programación con dichas capacidades.

Por ejemplo, utilizando los isomorfismos CURRY y COMM es posible aplicar una función a sus argumentos dados en cualquier orden e incluso aplicar la función a solo un subconjunto de argumentos. La idea es convertir todos los argumentos en forma de pares utilizando CURRY, y luego reordenarlos empleando conmutatividad y asociatividad de los pares:

$$\begin{aligned} R \to S \to T &\equiv_{\text{CURRY}} \\ (R \times S) \to T &\equiv_{\text{COMM}} \\ (S \times R) \to T &\equiv_{\text{CURRY}} \\ S \to R \to T & \end{aligned}$$

Un detalle interesante de este sistema, es que a nivel de la implementación, se añade una nueva regla de reescritura llamada $dist_e$ que no surge de ningún isomorfismo, sino que es necesaria para reducir términos que de otra forma quedarían atascados. Esta regla reescribe términos de la forma $\pi_{R\times S}(\langle r, s\rangle) \hookrightarrow \langle \pi_R(r), \pi_S(s)\rangle$ y permite desatascar proyecciones donde el argumento es una abstracción más algún otro término.

Por ejemplo, dados los términos $\lambda x^{R\times S}.x : (R \times S) \to (R \times S)$ y $t : T$ se puede construir el par $\langle \lambda x^{R\times S}.x, t\rangle : (((R \times S) \to (R \times S)) \times T)$, luego aplicando las reglas DIST y ASSO se obtiene la siguiente equivalencia:

$$\begin{aligned} ((R \times S) \to (R \times S)) \times T &\equiv_{\text{DIST}} \\ (((R \times S) \to R) \times ((R \times S) \to S)) \times T &\equiv_{\text{ASSO}} \\ ((R \times S) \to R) \times (((R \times S) \to S) \times T) & \end{aligned}$$

Por lo tanto, la proyección $\pi_{((R\times S)\to S)\times T}\langle \lambda x^{R\times S}.x, t\rangle$ está bien tipada. A continuación se muestra como emplear la regla $dist_e$ para reducir dicha proyección:

$$\begin{aligned} & \pi_{((R\times S)\to S)\times T}\langle \lambda x^{R\times S}.x, t\rangle \\ \hookrightarrow_{dist_e} & \\ & \langle \pi_{(R\times S)\to S}(\lambda x^{R\times S}.x), \pi_T(t)\rangle \\ \hookrightarrow & \\ & \langle \pi_{(R\times S)\to S}(\lambda x^{R\times S}.x), t\rangle \\ \rightleftarrows & \\ & \langle \lambda x^{R\times S}.\pi_S(x), t\rangle \end{aligned}$$

Esto muestra que desde el punto de vista de la implementación, puede ser necesario agregar nuevas reglas de reescritura para evitar que aparezcan eliminaciones en formas normales. Este punto es importante, ya que la formalización de la propiedad de progreso implica demostrar que es posible reducir todas las eliminaciones hasta llegar a una forma normal, en simples palabras, el conjunto de reglas es lo suficientemente grande como para desatascar cualquier término que pueda presentarse.

2.2. Tipos dependientes

Un tipo de datos dependiente es aquel cuya definición depende de valores. Son usados para expresar una propiedad sobre los términos en el tipo de los mismos. Un ejemplo clásico que se suele dar a la hora de presentar los tipos dependientes, es el de los vectores de largo n:

```
data Vec a (n :: ℕ) where
  nil  :: Vec a 0
  cons :: Vec a n -> Vec a (suc n)
```

Es importante notar que el segundo parámetro de `Vec` no es el tipo ℕ, sino un término de tipo ℕ, por ejemplo, `Vec ℕ 3` representa el tipo de los vectores de naturales de largo 3. Esto quiere decir que dentro de los tipos pueden aparecer términos, lo cual permite escribir programas más precisos y seguros.

Para entender la practicidad de un sistema de tipos con dicho nivel de especificidad, se comparan los siguientes programas:

```
zerosL :: ℕ -> List ℕ
zerosL 0       = []
zerosL (suc n) = 0 :: (zerosL n)

zerosV :: (n :: ℕ) -> Vec ℕ n
zerosV 0       = nil
zerosV (suc n) = cons 0 (zerosV n)
```

Notar que el tipo de *zerosV* es dependiente e introduce una variable. Si bien la implementación dada para *zerosL* es correcta, nada impediría simplemente retornar `[]` en todos los casos, por el contrario, en el caso `(suc n)` de *zerosV* se estaría cometiendo un error de tipo si se tratara de retornar `[]`, ya que se espera un término de tipo `Vec ℕ (suc n)` y la lista vacía tiene tipo `Vec ℕ 0`. De cierta forma el tipo de las funciones está guiando su implementación, impidiendo, en este caso, retornar un vector de largo incorrecto.

Otro ejemplo interesante que se logra con la programación con tipos dependientes es el operador de acceso. Sobre las listas, este operador se define como:

```
_!!_ :: List a -> ℕ -> Maybe a
[] !! n             = nothing
(x :: xs) !! zero   = just x
(x :: xs) !! (suc n) = xs !! n
```

Notar el uso del tipo de datos `Maybe` para capturar los casos en que el acceso no es posible.

Antes de presentar la implementación para vectores, es necesario definir el tipo de datos de los conjuntos finitos:

```
data Fin ℕ where
  zero : Fin (suc n)
  suc  : Fin n -> Fin (suc n)
```

Cada tipo `Fin n` está habitado por n elementos, estos serían, el elemento `zero` más el constructor `suc` combinado con los $n-1$ elementos de `Fin (n-1)`. Notar que `Fin 0` es un conjunto vacío, es decir que no existe ningún término con dicho tipo, o dicho de otra forma, el tipo `Fin 0` no está habitado por ningún término. Luego, un vector de largo n estará indexado por los elementos del conjunto `Fin n`, que tiene cardinalidad exactamente igual a n.

```
_!!v_ :: Vec a n -> Fin n -> a
(cons x v) !!v zero  = x
(cons x v) !!v (suc i) = v !!v i
```

La definición del operador sobre vectores no requiere el uso del tipo `Maybe`, ya que el segundo argumento está restringido a ser un natural entre cero y el largo del vector menos uno. Notar que en la implementación ya no aparece el caso `nil`, porque eso implicaría que $n = 0$, pero el parámetro n del tipo dependiente está forzando al argumento `Fin` a tener la misma cantidad de elementos que el vector, por lo que dicho argumento debería ser un elemento de `Fin 0`, el cual es un conjunto vacío. Esto es lo que se denomina un "patrón absurdo", y como es imposible que dicho patrón ocurra, se lo excluye de la implementación.

Utilizando estos tipos es imposible tratar de acceder a un elemento que no esté en el rango del vector. Lo más interesante es que los invariantes son verificados estáticamente por el sistema de tipos en tiempo de compilación. Un término que trata de romper alguna invariante, estará mal tipado, por lo que no es posible construirlo.

2.2.1. Cubo Lambda

Hasta ahora se presentaron distintos tipos de cálculo lambda que se podrían agrupar en dos grandes categorías, el simplemente tipado ($\lambda_{\rightarrow}$), y el cálculo lambda polimórfico, también denominado *Sistema F* o cálculo lambda de segundo orden ($\lambda 2$).

La diferencia fundamental entre estos dos, son las dependencias entre tipos y términos que se permiten. Por ejemplo $\lambda_{\rightarrow}$ solo permite que las abstracciones liguen términos, es decir, los términos solo pueden depender de otros términos. Luego, $\lambda 2$ añade nuevas reglas de tipado que permiten construir términos que dependen de tipos.

La inclusión de tipos dependientes, es decir, construcciones que permiten a los tipos depender de términos, da como resultado un cálculo denominado $\lambda\Pi$. Si se incluyen estas tres dependencias, es decir, términos que dependen de términos o tipos y tipos que dependen de términos, se obtiene el cálculo lambda dependiente de segundo orden ($\lambda\Pi 2$).

Notar como cada dependencia es, en cierta forma, ortogonal a las demás, y cada cálculo obtenido a medida que se añaden dependencias, es un superconjunto de los cálculos anteriores, es decir, se obtienen generalizaciones cada vez más grandes. Se pueden visualizar las distintas dimensiones en las que es posible moverse hacia un cálculo más general utilizando un diagrama llamado "cubo lambda" [Bar91]:

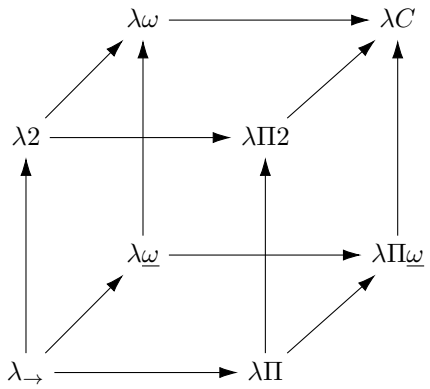

Figura 2.4: El cubo lambda representa las dependencias entre tipos y términos como dimensiones ortogonales, las flechas corresponden a la relación $\subsetneq$.

- ($\uparrow$): términos que dependen de tipos. El polimorfismo incrementa el poder de cálculo del sistema, ya que permite la auto-aplicación preservando la propiedad de terminación, por lo tanto, es posible representar cualquier función recursiva primitiva.
- ($\rightarrow$): tipos que dependen de términos. La inclusión de tipos dependientes no aumenta el poder de cálculo, pero permite expresar en el sistema de tipos propiedades sobre los programas. Como se explicará más adelante, los tipos dependientes permiten dar un salto a un sistema, que desde el punto de vista de la lógica, es equivalente a la lógica de predicados.
- ($\nearrow$): tipos que dependen de tipos. Corresponde a los constructores de tipos, es decir, operadores que permiten crear nuevos tipos a partir de otros tipos más básicos. El cálculo $\lambda\omega$, que permite representar funciones polimórficas y constructores de tipos, es considerado como la base de muchos lenguajes de programación funcionales.

El sistema que incluye a todos los demás, es denominado Cálculo de Construcciones (λC), este posee el mayor poder de cómputo y expresividad. Si bien los sistemas de tipos imponen restricciones sobre las construcciones que son posibles en los sistemas de cálculo, permiten obtener propiedades interesantes. Por ejemplo, la propiedad de terminación es necesaria para que la lógica representada por estos cálculos sea consistente. En particular, λC tiene un sistema de tipos lo suficientemente expresivo como para representar la lógica de predicados de alto orden, mientras que es lo suficientemente restrictivo como para que dicha lógica sea consistente.

2.3. Proposiciones como Tipos

En teoría de tipos, el paradigma de "proposiciones como tipos" describe la correspondencia entre la lógica y los lenguajes de programación. Básicamente, dice que a cada proposición en la lógica le corresponde un tipo, y viceversa. De hecho, esta relación es más profunda, ya que a cada prueba de una proposición dada, le corresponde un programa del tipo correspondiente, y viceversa. Es decir, "pruebas como programas". Incluso, es más profunda aún, en el sentido de que para cada forma de simplificar una prueba, existe una forma correspondiente de evaluar un programa, y viceversa. Por lo que se tiene "simplificación de pruebas como evaluación de programas".

Tal como lo explica Wadler en su artículo [Wad15], no se trata de una simple biyección entre proposiciones y tipos, sino de un verdadero isomorfismo que preserva la compleja estructura de pruebas y programas, simplificaciones y evaluaciones.

Este principio surge de las observaciones realizadas por Curry [CF58] sobre la lógica proposicional, y más tarde extendidas por Howard [How69] a la lógica de predicados. La clave de esta extensión es la introducción de los tipos dependientes para representar los predicados y cuantificadores en la lógica de predicados.

Las correspondencias que surgen de esta interpretación pueden resumirse de la siguiente forma:

- La conjunción $A \wedge B$ se corresponde con el par $A \times B$. Una prueba de la proposición $A \wedge B$ consiste de una prueba de A y una prueba de B.
- La disjunción $A \vee B$ se corresponde con la suma disjunta $A + B$. Una prueba de la proposición $A \vee B$ consiste de una prueba de A o una prueba de B.
- La implicación $A \Rightarrow B$ se corresponde con el espacio de funciones $A \to B$. Una prueba de la proposición $A \Rightarrow B$ consiste de una función que dada una prueba de A devuelve una prueba de B.
- El cuantificador existencial $\exists x : A.B$ se corresponde con el tipo $\Sigma x : A.B$. Básicamente, esto es una familia de tipos indexada por $a : A$ donde a cada término a le corresponde un tipo $B(a)$. Los elementos canónicos de $\Sigma x : A.B$ son pares dependientes $\langle a, b \rangle$ donde $a : A$ y $b : B(a)$. Cuando $B(x)$ es una función constante, este tipo es equivalente al producto cartesiano $A \times B$.
- El cuantificador universal $\forall x : A.B$ se corresponde con $\Pi x : A.B$, al igual que para el tipo Σ, B es una familia de tipos indexada por los términos de tipo A. Los elementos canónicos de $\Pi x : A.B$ son funciones dependientes $a \to B(a)$. Cuando $B(x)$ es una función constante, el tipo Π es equivalente al tipo de las funciones ordinarias $A \to B$.

2.4. Teoría de Tipos Intuicionista

La Teoría de Tipos Intuicionista, también llamada Teoría de Tipos de Martin-Löf [Mar98; Mar75; Mar82; Mar84] propone un sistema lógico formal y los fundamentos filosóficos para las matemáticas constructivas.

Directamente influenciado por las ideas de Howard, Martin-Löf se basó en el principio de proposiciones como tipos y el constructivismo matemático para el desarrollo de su teoría. Este constructivismo requiere que las pruebas contengan un "testigo", una prueba de una proposición dada es un programa, por lo tanto, las proposiciones son verdaderas cuando su tipo está habitado por algún término. Las pruebas son términos que atestiguan la veracidad del teorema, y pueden ser manipulados como cualquier otro término del lenguaje. Vistas como programas, las pruebas son procedimientos que al ejecutarlos permiten obtener valores del tipo indicado por la proposición, por este motivo se dice que las pruebas son constructivas.

Esto tiene algunas consecuencias interesantes, por ejemplo, en la lógica clásica, a las proposiciones se les asigna valores de verdad sin importar si existe evidencia directa de que sea verdadera o falsa. Esto es lo que comúnmente se denomina "principio del tercero excluido", ya que excluye la posibilidad de un tercer valor distinto de verdadero o falso. Sin embargo, puede que no siempre sea posible obtener una prueba constructiva de dicha proposición para cualquier tipo A, por lo tanto, no es posible probar $A \vee \neg A$ en la lógica intuicionista.

Se puede pensar a la teoría de tipos intuicionista como un lenguaje de programación funcional donde el sistema de tipos es tan rico que prácticamente cualquier propiedad concebible de un programa puede expresarse como un tipo. Todas las funciones de este lenguaje deben ser totales y decidibles, por lo que todos los programas deben necesariamente cumplir con la propiedad de terminación.

Filosófica y prácticamente, la Teoría de Tipos de Martin-Löf es un marco fundamental donde las matemáticas constructivas y la programación son, en un sentido profundo, lo mismo.

2.5. Agda

Agda es un lenguaje de programación con tipos dependientes, desarrollado originalmente por la Universidad de Chalmers (Suecia). Debido al paradigma de las proposiciones como tipos, Agda también funciona como un asistente de pruebas. A diferencia de otros asistentes, como Coq, carece de un sistema de tácticas, por lo que las pruebas son escritas en un estilo de programación funcional, de hecho su sintaxis es similar a la de Haskell.

Agda es una implementación de la Teoría de Tipos de Martin-Löf, para que la lógica sea consistente, los programas deben ser totales, es decir que todos los programas deben terminar, y todos los posibles casos de un *pattern matching* deben ser cubiertos. Por este motivo, no todas las funciones recursivas están permitidas, Agda posee un mecanismo de comprobación de terminación que acepta aquellas funciones para las que puede probar mecánicamente su terminación.

2.6. Indices de De Bruijn

Generalmente, los términos en cálculo lambda se presentan utilizando letras para nombrar las variables, por ejemplo:

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

Esta forma de representación permite ver a simple vista cuáles variables están ligadas y a qué abstracción pertenecen, también suelen utilizarse palabras para dar nombres más descriptivos a las variables. Se puede notar que la forma de escribir un término no es única, ya que es posible cambiar los nombres de algunas variables sin alterar su significado, por ejemplo, el término $\lambda c.(\lambda b.b(\lambda d.d))(\lambda a.ca)$ es equivalente al término anterior. Cuando esto ocurre se dice que los términos son α-equivalentes.

El principal problema de esta representación es que para implementar la β-reducción, se debe tener cuidado de no capturar una variable libre cuando se substituye un término en el cuerpo de una abstracción, en caso de que eso ocurra, se renombra la variable capturada con un nuevo nombre fresco. En el siguiente ejemplo la variable x es renombrada a z para evitar la captura:

$$(\lambda y.(\lambda x.xy))x \hookrightarrow_\beta (\lambda x.xy)[y := x] = \lambda z.zx$$

Utilizar variables con nombres hace que la implementación se vuelva más engorrosa e ineficiente. Una alternativa más adecuada es la representación de De Bruijn [Bru72], que reemplaza los nombres por números naturales llamados *índices de De Bruijn*. Por ejemplo, la forma de escribir el término del primer ejemplo con índices es la siguiente:

$$\lambda(\lambda\ 0\ (\lambda\ 0))(\lambda\ 1\ 0)$$

Los índices indican cuantas abstracciones se deben "saltar" para llegar a la que está ligando la variable. Los nombres de las ligaduras ya no son necesarios, por lo que la escritura se simplifica. Además, los términos tienen una única representación, es decir, no es necesario tener en cuenta

las α-equivalencias. A nivel de la implementación, esta representación facilita la manipulación de términos, en especial, cuando se implementa la β-reducción.

2.7. Substituciones explícitas

Utilizando la representación de De Bruijn, las sustituciones son simplemente mapeos de números naturales a términos, es decir que pueden interpretarse como una secuencia infinita de términos. Por ejemplo:

$$(0\ 1\ 3)\{0 \mapsto a, 1 \mapsto b, 2 \mapsto 0, 3 \mapsto 1, \dots\} \to a\ b\ 1$$

Si se quisiera definir la β-reducción utilizando esta notación, una primera definición sería:

$$(\lambda a)b \to_\beta a\{0 \mapsto b, 1 \mapsto 1, 2 \mapsto 2, \dots\}$$

El problema es que al eliminar un λ todas las variables libres de a quedarán desfasadas, por lo que se les debe restar uno:

$$(\lambda a)b \to_\beta a\{0 \mapsto b, 1 \mapsto 0, 2 \mapsto 1, \dots\}$$

El siguiente problema surge al intentar empujar la substitución dentro de una abstracción, por ejemplo si $a = \lambda c$, la sustitución que se aplique sobre c no debería sustituir el índice 0, ya que éste representa a la variable capturada por la abstracción, además los demás índices a sustituir en c estarán bajo una ligadura. Para solucionar este problema se modifica la sustitución a aplicar sobre el cuerpo de una abstracción, eliminando la sustitución de 0 y aumentando en un 1 los demás índices a sustituir:

$$(\lambda c)\{0 \mapsto b, i+1 \mapsto i\} = \lambda c\{0 \mapsto 0, 1 \mapsto b, 2 \mapsto 1, 3 \mapsto 2, \dots\}$$

Un último problema puede presentarse si b tiene variables libres, para evitar que estas sean capturadas por el λ de c se les debe sumar uno:

$$(\lambda c)\{0 \mapsto b, i+1 \mapsto i\} = \lambda c\{0 \mapsto 0, 1 \mapsto b\{0 \mapsto 1, 1 \mapsto 2, \dots\}, 2 \mapsto 1, 3 \mapsto 2, \dots\}$$

El siguiente ejemplo muestra la forma correcta de realizar una β reducción:

$$(\lambda\ \lambda\ 3\ 1\ (\lambda\ 0\ 2))\ (\lambda\ 4\ 0) \hookrightarrow_\beta \lambda\ 2\ (\lambda\ 5\ 0)\ (\lambda\ 0\ (\lambda\ 6\ 0))$$

Notar como las variables libres del cuerpo de la abstracción disminuyen en uno, mientras que en el argumento de la abstracción las variables libres aumentan en uno por cada λ que atraviesan, y las variables ligadas quedan intactas.

En [Aba+91] se presenta el álgebra-σ y los cuatro operadores que permiten construir estas secuencias. Donde se escribe $\langle\langle s \rangle\rangle a$ para expresar la aplicación de la substitución s sobre el término a.

- id es la substitución identidad $\{i \mapsto i\}$.
- $\uparrow$ es el operador shift, y suma uno a cada índice $\{i \mapsto i+1\}$.
- $a \bullet s$ es la concatenación del término a con la substitución s, que se define como $\{0 \mapsto a, i+1 \mapsto s(i)\}$. Por ejemplo $a \bullet id = \{0 \mapsto a, 1 \mapsto 0, 2 \mapsto 1, 3 \mapsto 2, \dots\} = \{0 \mapsto a, i+1 \mapsto i\}$
- $s \circ t$ corresponde a la composición de substituciones, donde primero se aplica s y luego t $\{i \mapsto \langle\langle t \rangle\rangle (s(i))\}$.

Dada una substitución s, se escribe $\langle\langle s \rangle\rangle a$ para denotar la aplicación de la substitución s sobre el término a.

$$\begin{aligned} \langle\langle s \rangle\rangle n &= s(n) \\ \langle\langle s \rangle\rangle (a\ b) &= \langle\langle s \rangle\rangle a\ \langle\langle s \rangle\rangle b \\ \langle\langle s \rangle\rangle (\lambda a) &= \lambda \langle\langle 0 \bullet (s \circ \uparrow) \rangle\rangle a \end{aligned}$$

Finalmente, la β-reducción se define como:

$$(\lambda a) b \hookrightarrow_\beta \langle\langle b \bullet id \rangle\rangle a$$

El trabajo de Abadi también presenta algunas propiedades algebraicas de los operadores que resultan útiles a la hora de probar propiedades sobre las substituciones:

$$\begin{aligned} 0 \bullet \uparrow &= id \\ \uparrow \circ (a \bullet s) &= s \\ (\langle\langle s \rangle\rangle 0) \bullet (\uparrow \circ s) &= s \\ (a \bullet s) \circ t &= (\langle\langle t \rangle\rangle a) \bullet (s \circ t) \end{aligned}$$

Capítulo 3

Aportes

3.1. Formalización

3.1.1. Tipos intrínsecos

Existen dos enfoques fundamentales para la introducción de sistemas de tipos en el cálculo lambda. Por un lado, se pueden definir primero los términos y luego los tipos, por lo tanto, los términos existen independientemente de los tipos y tienen significado por sí solos. Tiene sentido entonces definir el subconjunto de términos para los cuales existe alguna derivación de tipos, a este subconjunto se lo denomina términos "bien tipados". De hecho, para un término dado pueden existir más de un tipo posible, por ejemplo, la función identidad $\lambda x.x$ puede ser tipada como $\tau \to \tau$ o como $(\tau \to \tau) \to (\tau \to \tau)$. Desde este punto de vista, los juicios de tipado aseguran que los términos poseen ciertas propiedades. Este estilo es llamado "tipos a la Curry".

Por otro lado, es posible definir los tipos en primer lugar y luego los términos sobre esos tipos. Aquí no tiene sentido hablar de términos "bien tipados", ya que no pueden existir los términos "mal tipados". Por lo tanto, todo el significado recae sobre los juicios de tipado, en lugar de los términos. En cierta forma, los términos y las reglas de tipado están entrelazados. Aquí, las derivaciones de tipos son únicas, puesto que los términos $\lambda x^{\tau}.x$ y $\lambda x^{\tau \to \tau}.x$ se consideran distintos. A este enfoque se lo denomina "tipos a la Church".

Reynolds [Rey98] acuñó las expresiones *tipos extrínsecos* y *tipos intrínsecos* para referirse a cada uno de estos dos enfoques.

3.1.2. Cálculo lambda con pares y tipo Top

En esta sección se presenta una formalización en Agda de un cálculo lambda simplemente tipado con pares extendido con isomorfismos de tipo. Gran parte del código presentado en este trabajo está basado y adaptado a partir de un libro de Philip Wadler [WKS22]. La adaptación consiste principalmente en la adición de los isomorfismos de tipos, la regla de tipado ($\equiv$) y la relación de equivalencia entre términos.

En primera instancia se definen los tipos del lenguaje, existen tres constructores de tipos. El tipo top $\top$, el tipo de las funciones $\_\Rightarrow\_$ y los pares $\_\times\_$.

Código 1. Tipo de dato de los tipos

```
data Type : Set where
  ⊤   : Type
```

```
_⇒_ : Type → Type → Type
_×_ : Type → Type → Type
```

Ejemplo 1. Construcción de tipos

```
fun : Type
fun = ⊤ ⇒ ⊤

prod : Type
prod = fun × ⊤
```

Dado que se utiliza la representación de De Brujin, los entornos de tipado, simplemente se formalizan como listas de tipos. Al contrario de las listas clásicas, los entornos se leen de derecha a izquierda.

Luego se formalizan las variables intrínsecamente tipadas, que son representadas por los índices propiamente dichos.

Código 2. Tipo de dato de los contextos de tipado y variables intrínsecamente tipadas. Los argumentos escritos entre {} son implícitos, cuando se aplican los constructores, estos argumentos se omiten y Agda se encargará de inferir los valores correspondientes.

```
data Context : Set where
  ∅   : Context
  _,_ : Context → Type → Context

data _∋_ : Context → Type → Set where
  Z : ∀ {Γ A}
      ---------
    → Γ , A ∋ A

  S_ : ∀ {Γ A B}
    → Γ ∋ B
      ---------
    → Γ , A ∋ B
```

La proposición $\Gamma \ni A$ indica que existe una variable de tipo A dentro del entorno Γ, es decir que este es el tipo de las variables que tienen tipo A en el entorno Γ. Los términos construidos con `Z` y `S` corresponden a los índices de De Bruijn, los cuales constituyen pruebas para dichas proposiciones.

Ejemplo 2. Las siguientes variables con tipos $\top$ y $\top \Rightarrow \top$ tipan en el contexto $\emptyset, \top \Rightarrow \top, \top$ y constituyen una prueba de ello.

Cuando el nombre de una definición no es relevante, Agda permite omitirlo utilizando un guion bajo _.

```
_ : ∅ , ⊤ ⇒ ⊤ , ⊤ ∋ ⊤
_ = Z

_ : ∅ , ⊤ ⇒ ⊤ , ⊤ ∋ ⊤ ⇒ ⊤
_ = S Z
```

A continuación se presentan los juicios de tipado, debido a que se utilizan tipos intrínsecos, estos también representarán a los términos del lenguaje.

Código 3. Tipo de dato de los términos

```
data _⊢_ : Context → Type → Set where

  ‘_ : ∀ {Γ A}       -- (ax)
    → Γ ∋ A
      -----
    → Γ ⊢ A

  ⋆ : ∀ {Γ} → Γ ⊢ ⊤ -- (⊤ᵢ)

  [_]≡_ : ∀ {Γ A B} -- (≡)
    → A ≡ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B

  ƛ_  : ∀ {Γ A B}    -- (⇒ᵢ)
    → Γ , A ⊢ B
      ---------
    → Γ ⊢ A ⇒ B

  _·_ : ∀ {Γ A B}    -- (⇒ₑ)
    → Γ ⊢ A ⇒ B
    → Γ ⊢ A
      ---------
    → Γ ⊢ B

  ⟨_,_⟩ : ∀ {Γ A B} -- (×ᵢ)
    → Γ ⊢ A
    → Γ ⊢ B
      -----------
    → Γ ⊢ A × B

  π : ∀ {Γ A B}      -- (×ₑ)
    → (C : Type)
    → {proof : (C ≅ A) ⊎ (C ≅ B)}
    → Γ ⊢ A × B
      -----------
    → Γ ⊢ C
```

De manera similar a las variables, un término de tipo $\Gamma \vdash C$ constituye una prueba de que el mismo tiene tipo C en el entorno Γ. Notar que el constructor `[_]≡_` corresponde a la regla de tipado (≡) de Sistema I. En el constructor π se observa un detalle importante, además de tomar como argumento el tipo C, toma un argumento implícito que funciona como evidencia de que el tipo C es o bien igual al tipo A, o bien igual al tipo B. Para ello se utilizan el tipo suma `_⊎_` definido en el módulo `Data.Sum` y la igualdad proposicional `_≅_` definida en el módulo `Relation.Binary.PropositionalEquality`. Estos dos tipos forman parte de la librería estándar

de Agda. Generalmente, se utiliza el símbolo (≡) para denotar la igualdad proposicional, pero en ese caso, ese símbolo se utilizará para definir los isomorfismos de tipos, por lo tanto, se renombra el operador de la igualdad al símbolo (≅).

A continuación se muestran algunos ejemplos de términos que pueden construirse con la definición dada. Cada uno de ellos constituye una prueba de que tipa en el contexto dado.

Ejemplo 3. Construcción de términos

```
_ : ∅ , ⊤ ⇒ ⊤ ⊢ ⊤ ⇒ ⊤
_ = ƛ (` S Z · (` S Z · ` Z)) -- λ x . y (y x)

_ : ∅ ⊢ (⊤ ⇒ ⊤) ⇒ ⊤ ⇒ ⊤
_ = ƛ ƛ (` S Z · (` S Z · ` Z)) -- λ y . λ x . y (y x)

T₁ : ∅ , ⊤ ⊢ fun
T₁ = ƛ ` Z

T₂ : ∅ , ⊤ ⊢ prod
T₂ = ⟨ T₁ , ⋆ ⟩

T₃ : ∅ , ⊤ ⊢ ⊤
T₃ = (π fun {inj₁ refl} T₂) · ` Z

swap× : ∀ {A B} → ∅ ⊢ A × B ⇒ B × A
swap× {A} {B} = ƛ ⟨ π B {inj₂ refl} (` Z) , π A {inj₁ refl} (` Z) ⟩
```

Aquí se introducen las definiciones de los renombres de variables, substitución de variables y operadores del álgebra-σ presentados en capítulos anteriores.

```
Rename : Context → Context → Set
Rename Γ Δ = ∀{A} → Γ ∋ A → Δ ∋ A

Subst : Context → Context → Set
Subst Γ Δ = ∀{A} → Γ ∋ A → Δ ⊢ A

ids : ∀{Γ} → Subst Γ Γ
ids x = ` x

_•_ : ∀{Γ Δ A} → (Δ ⊢ A) → Subst Γ Δ → Subst (Γ , A) Δ
(M • σ) Z = M
(M • σ) (S x) = σ x
```

Los renombres son simplemente funciones que mapean las variables de un entorno a otro preservando el tipo de las mismas. Por otro lado, las substituciones mapean cada variable de un entorno a un término que tenga el mismo tipo que dicha variable. Se pueden pensar a estas substituciones como las secuencias de términos que se utilizarán para reemplazar las variables libres cuando se aplica dicha substitución sobre otro término.

La implementación de las substituciones explícitas se divide en dos partes. Primero se implementa la operación `rename` que dada una función de renombrado ρ y un término, aplica el renombre de variables ρ a dicho término. Se debe tener cuidado de no capturar el índice 0 cuando se empuja el renombre debajo de una abstracción. Por este motivo, se define la función `ext` que toma un renombre y retorna uno nuevo con los entornos extendidos. Notar que esta definición sigue la forma propuesta por Abadi et al.: $Z \bullet (\rho \circ S\_)$

Código 4. Aplicación de renombres sobre términos

```
ext : ∀ {Γ Δ A}
  → Rename Γ Δ
    --------------------
  → Rename (Γ , A) (Δ , A)
ext ρ Z     = Z
ext ρ (S x) = S (ρ x)


rename : ∀ {Γ Δ}
  → Rename Γ Δ
    ----------------------
  → (∀ {A} → Γ ⊢ A → Δ ⊢ A)
rename ρ (` x)          = ` (ρ x)
rename ρ (ƛ N)          = ƛ (rename (ext ρ) N)
rename ρ (L · M)        = (rename ρ L) · (rename ρ M)
rename ρ ⟨ M , N ⟩      = ⟨ rename ρ M , rename ρ N ⟩
rename ρ (π C {p} L)    = π C {p} (rename ρ L)
rename ρ ⋆              = ⋆
rename ρ ([ iso ]≡ N) = [ iso ]≡ rename ρ N
```

Ejemplo 4. La función `rename S_` simplemente suma uno a todas las variables libres del término. Al aplicar dicho renombre sobre el término M_0 se obtiene el término M_1.

```
M₀ : ∅ , ⊤ ⇒ ⊤ ⊢ ⊤ ⇒ ⊤
M₀ = ƛ (` S Z · (` S Z · ` Z)) -- λ 1 (1 0)

M₁ : ∅ , ⊤ ⇒ ⊤ , ⊤ ⊢ ⊤ ⇒ ⊤
M₁ = ƛ (` S S Z · (` S S Z · ` Z)) -- λ 2 (2 0)

_ : rename S_ M₀ ≅ M₁
_ = refl
```

Una vez definida la función `rename`, se puede implementar la aplicación de substituciones. Para ello, se define de forma análoga, una función `exts` que extiende la sustitución cuando se atraviesa una abstracción. Utilizando la notación del álgebra-σ, se puede observar que aquí también se sigue la misma estructura: $`Z \bullet (\sigma \circ \text{rename } S\_)$.

Luego se define la substitución simple `_[_]`, que dado un término N cuya primera variable libre es de tipo B y un término M de tipo B, reemplaza la primera variable libre de N por M y reduce en uno el resto de las variables libres.

Código 5. Aplicación de substituciones sobre términos

```
exts : ∀ {Γ Δ A}
  → Subst Γ Δ
    --------------------
  → Subst (Γ , A) (Δ , A)
exts σ Z     = ` Z
exts σ (S x) = rename S_ (σ x)
```

```
subst : ∀ {Γ Δ} → Subst Γ Δ → (∀ {C} → Γ ⊢ C → Δ ⊢ C)
subst σ (` k)         = σ k
subst σ (ƛ N)         = ƛ (subst (exts σ) N)
subst σ (L · M)       = (subst σ L) · (subst σ M)
subst σ ⟨ M , N ⟩     = ⟨ subst σ M , subst σ N ⟩
subst σ (π C {p} L)   = π C {p} (subst σ L)
subst σ ⋆             = ⋆
subst σ ([ iso ]≡ N) = [ iso ]≡ subst σ N

⟪_⟫ : ∀{Γ Δ A} → Subst Γ Δ → Γ ⊢ A → Δ ⊢ A
⟪ σ ⟫ = λ M → subst σ M

_[_] : ∀ {Γ A B}
  → Γ , B ⊢ A
  → Γ ⊢ B
    ---------
  → Γ ⊢ A
_[_] {Γ} {A} {B} N M = ⟪ M • ids ⟫ N
```

Notar como las definiciones de `rename` y `subst` poseen una estructura similar, esto se debe a que los renombres son un caso particular de substitución, donde todos los términos retornados por la substitución son variables. Esta implementación está basada en la técnica de McBride[McB05], donde los renombres de variables y substituciones son dos instancias de una misma operación de recorrido. El motivo principal por el cual es necesario implementar `rename`, es que si se quisiera definir `exts` utilizando `subst` $(\lambda\ v \to$ ` (S` $v))$ $(\sigma\ x)$, la llamada recursiva sería sobre el término retornado por σ, de modo que el chequeo de terminación de Agda fallaría.

Ejemplo 5. La substitución simple reemplaza la primera variable libre por un término del mismo tipo que dicha variable. En el siguiente ejemplo se sustituye la primer variable libre del termino M_2 (el índice 1) por el término M_3, obteniéndose como resultado el término M_4.

```
M₂ : ∅ , ⊤ ⇒ ⊤ ⊢ ⊤ ⇒ ⊤
M₂ = ƛ ` S Z · (` S Z · ` Z)      -- λ 1 (1 0)

M₃ : ∅ ⊢ ⊤ ⇒ ⊤
M₃ = ƛ ` Z                        -- λ 0

M₄ : ∅ ⊢ ⊤ ⇒ ⊤
M₄ = ƛ (ƛ ` Z) · ((ƛ ` Z) · ` Z) -- λ (λ 0) ((λ 0) 0)

_ : M₂ [ M₃ ] ≅ M₄
_ = refl
```

En el ejemplo se puede ver que el contexto del término resultante tiene un elemento menos, esto es debido a que la substitución reemplaza la primer variable libre eliminándola.

Ejemplo 6. Cuando se reemplaza un término con variables libres en el cuerpo de una abstracción, las mismas son renombradas para evitar que sean capturadas.

```
M₅ : ∅ , ⊤ ⇒ ⊤ , ⊤ ⊢ (⊤ ⇒ ⊤) ⇒ ⊤
M₅ = ƛ ` Z · ` S Z        -- λ 0 1
```

```
M₆ : ∅ , ⊤ ⇒ ⊤ ⊢ ⊤
M₆ = ‘ Z · ⋆                -- 0 ⋆

M₇ : ∅ , ⊤ ⇒ ⊤ ⊢ (⊤ ⇒ ⊤) ⇒ ⊤
M₇ = ƛ (‘ Z · (‘ S Z · ⋆)) -- λ 0 (1 ⋆)

_ : M₅ [ M₆ ] ≅ M₇
_ = refl
```

Finalmente, se presenta la relación de reducción. Los constructores ξ y ζ representan las reglas de congruencia que permiten construir reducciones sobre subtérminos. El constructor β-λ corresponde a la beta reducción y los constructores β-π_1 y β-π_2 a las aplicaciones de las proyecciones.

Código 6. Tipo de dato de la relación de reducción

```
data _↪_ : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  ξ-·₁ : ∀ {t t' : Γ ⊢ A ⇒ B} {s : Γ ⊢ A}
    → t ↪ t'
      ---------------
    → t · s ↪ t' · s

  ξ-·₂ : ∀ {t : Γ ⊢ A ⇒ B} {s s' : Γ ⊢ A}
    → s ↪ s'
      ---------------
    → t · s ↪ t · s'

  β-ƛ : ∀ {t : Γ , A ⊢ B} {s : Γ ⊢ A}
      --------------------
    → (ƛ t) · s ↪ t [ s ]

  ξ-⟨,⟩₁ : ∀ {r r' : Γ ⊢ A} {s : Γ ⊢ B}
    → r ↪ r'
      ------------------------
    → ⟨ r , s ⟩ ↪ ⟨ r' , s ⟩

  ξ-⟨,⟩₂ : ∀ {r : Γ ⊢ A} {s s' : Γ ⊢ B}
    → s ↪ s'
      ------------------------
    → ⟨ r , s ⟩ ↪ ⟨ r , s' ⟩

  ξ-π : ∀ {t t' : Γ ⊢ A × B}
    → t ↪ t'
      --------------------
    → π C {p} t ↪ π C {p} t'

  β-π₁ : ∀ {r : Γ ⊢ A} {s : Γ ⊢ B}
      ---------------------
    → π A {inj₁ refl} ⟨ r , s ⟩ ↪ r

  β-π₂ : ∀ {r : Γ ⊢ A} {s : Γ ⊢ B}
```

```
    ----------------------
  → π B {inj₂ refl} ⟨ r , s ⟩ ↪ s

ξ-≡ : ∀ {t t' : Γ ⊢ A} {iso : A ≡ B}
  → t ↪ t'
  → [ iso ]≡ t ↪ [ iso ]≡ t'

ζ : ∀ {t t' : Γ , B ⊢ A}
  → t ↪ t'
  → ƛ t ↪ ƛ t'
```

Es importante destacar que el tipo de $\_\hookrightarrow\_$, está indexado por dos términos del mismo tipo. Por lo tanto, no es necesario demostrar que la reducción preserva tipos, ya que es imposible construir una reducción que arribe a un término de distinto tipo. Esta es una de las ventajas más importantes de utilizar tipos intrínsecos.

Ejemplo 7. En el siguiente ejemplo se construye una reducción del término $\mathtt{T}_3$ al término $\mathtt{T}_1 \cdot \text{`} \mathtt{Z}$ usando los constructores de la relación $\_\hookrightarrow\_$. Notar como las congruencias permiten construir reducciones sobre subtérminos.

```
T₁ : ∅ , ⊤ ⊢ fun
T₁ = ƛ ` Z

T₂ : ∅ , ⊤ ⊢ prod
T₂ = ⟨ T₁ , ⋆ ⟩

T₃ : ∅ , ⊤ ⊢ ⊤
T₃ = (π fun {inj₁ refl} T₂) · ` Z

_ : T₃ ↪ T₁ · ` Z
_ = ξ-·₁ β-π₁
```

3.1.3. Isomorfismo de tipos

Los isomorfismos de tipos que se incluyen en esta formalización corresponden al conjunto axiomático $Th^1_{\times\top}$. Además de los axiomas, se agregan el constructor `sym` que representa la simetría y los constructores con el prefijo `cong` que representan las reglas de congruencia.

Código 7. Relación de equivalencia entre tipos isomorfos

```
data _≡_ : Type → Type → Set where
  comm  : ∀ {A B} → A × B ≡ B × A
  asso  : ∀ {A B C} → A × (B × C) ≡ (A × B) × C
  dist  : ∀ {A B C} → (A ⇒ B) × (A ⇒ C) ≡ A ⇒ B × C
  curry : ∀ {A B C} → A ⇒ B ⇒ C ≡ (A × B) ⇒ C
  id-×  : ∀ {A} → A × ⊤ ≡ A
  id-⇒  : ∀ {A} → ⊤ ⇒ A ≡ A
  abs   : ∀ {A} → A ⇒ ⊤ ≡ ⊤

  sym     : ∀ {A B} → A ≡ B → B ≡ A
```

```
cong⇒₁ : ∀ {A B C} → A ≡ B → A ⇒ C ≡ B ⇒ C
cong⇒₂ : ∀ {A B C} → A ≡ B → C ⇒ A ≡ C ⇒ B
cong×₁ : ∀ {A B C} → A ≡ B → A × C ≡ B × C
cong×₂ : ∀ {A B C} → A ≡ B → C × A ≡ C × B
```

Utilizando estos constructores es posible construir cualquier isomorfismo. Notar que no existe una regla para la transitividad, ya que es posible obtenerla aplicando dos veces el constructor $[\_]\equiv\_$. Por ejemplo, el término `[ trans` iso_1 iso_2 `]≡` t puede ser construido como `[` iso_2 `]≡ ([` iso_1 `]≡` t`)`. También se omite el constructor correspondiente a la reflexividad, dado que este solo permite obtener isomorfismos de la forma $A \equiv A$, por lo que no aporta mayor expresividad a la formalización.

Ejemplo 8. La relación $\_\equiv\_$ se utiliza para probar que dos tipos son isomorfos. A su vez, estos isomorfismos se emplean en conjunto con la regla $[\_]\equiv\_$ para construir términos equivalentes.

```
_ : (⊤ × ⊤) ⇒ ⊤ ≡ ⊤ ⇒ ⊤ ⇒ ⊤
_ = sym curry

_ : ∀ {A B} → A × B ⇒ ⊤ ≡ B × A ⇒ ⊤
_ = cong⇒₁ comm

T₄ : ∀ {A B} → ∅ ⊢ (A ⇒ B ⇒ A) × (A ⇒ B ⇒ B)
T₄ = [ sym dist ]≡ (ƛ [ sym dist ]≡ (ƛ ‹ ‘ (S Z) , ‘ Z ›))
```

Una observación importante es que este conjunto de isomorfismos permite obtener la ecuación $\top \to \top \equiv \top$, esto posibilita tipar el término $\Omega = (\lambda x^{\top}.xx)(\lambda x^{\top}.xx) : \top$. A continuación se muestra una posible derivación de tipos para dicho término:

$$\boldsymbol{\omega} \quad = \quad \dfrac{\dfrac{\dfrac{\top \equiv \top \to \top \qquad \dfrac{}{x : \top \vdash x : \top}\,(ax)}{x : \top \vdash x : \top \to \top}\,(\equiv) \qquad \dfrac{}{x : \top \vdash x : \top}\,(ax)}{x : \top \vdash xx : \top}\,(\Rightarrow_e)}{\vdash \lambda x.xx : \top \to \top}\,(\Rightarrow_i)$$

$$\dfrac{\boldsymbol{\omega} \qquad \dfrac{\top \to \top \equiv \top \qquad \boldsymbol{\omega}}{\vdash \lambda x.xx : \top}\,(\equiv)}{\vdash (\lambda x.xx)(\lambda x.xx) : \top}\,(\Rightarrow_e)$$

Si bien la existencia de este término parece suponer un impedimento para el cumplimiento de la propiedad de normalización fuerte, más adelante, luego de formalizar la evaluación, se mostrará la secuencia de reducción del término Ω y se explicarán los detalles de la implementación que permiten preservar dicha propiedad.

3.2. Equivalencia de términos

A continuación se presenta la formalización de la relación de simetría entre términos correspondientes a tipos isomorfos ($\rightleftarrows$). La selección de isomorfismos entre términos aquí presentada, es el resultado de varias decisiones de diseño basadas principalmente en dos objetivos.

En primer lugar, se agregaron todas las equivalencias necesarias para que ningún término quede atascado y no existan eliminaciones en formas normales, esto tiene como consecuencia que

todo término cerrado reduce siempre a un valor. Este punto es más evidente cuando se tiene en cuenta la prueba de progreso presentada más adelante.

El segundo objetivo es preservar la propiedad de normalización y evitar que su prueba se torne demasiado compleja. Una consecuencia directa de esto, son los constructores con el prefijo `sym-`, todos ellos se podrían obtener a partir de las equivalencias base combinadas con un constructor `sym` tal y como se construyen los isomorfismos de tipos. Sin embargo, incluir el constructor `sym` genera problemas a la hora de formalizar la prueba de normalización fuerte. Por otro lado, para desatascar algunos términos, es necesario incluir la η-expansión y una regla SPLIT $r \rightleftarrows \langle \pi_A\ r, \pi_B\ r \rangle$ donde $r : A \times B$. Si se incluyen estas reglas directamente, se pierde la propiedad de normalización, ya que por ejemplo nada impediría aplicar la η-expansión un número infinito de veces. La solución es definir nuevos constructores que embeban estas reglas, como por ejemplo `asso-split`.

Para facilitar la comprensión del código, se presenta una tabla de equivalencias utilizando la misma notación que la tabla presentada anteriormente para Sistema I. En esta se omiten las reglas de congruencia y los constructores `[_]≡_` de los términos de la izquierda.

	$\langle r, s \rangle \rightleftarrows \langle s, r \rangle$	(COMM)
	$\langle r, \langle s, t \rangle \rangle \rightleftarrows \langle \langle r, s \rangle, t \rangle$	(ASSO)
	$\langle r, s \rangle \rightleftarrows \langle \langle r, \pi_B(s) \rangle, \pi_C(s) \rangle$	(ASSO-SPLIT)
	$\lambda x^A.\langle r, s \rangle \rightleftarrows \langle \lambda x^A.r, \lambda x^A.s \rangle$	(DIST_λ)
Si $r : B \times C$	$\lambda x^A.r \rightleftarrows \langle \lambda x^A.\pi_B(r), \lambda x^A.\pi_C(r) \rangle$	(DIST_λ-SPLIT)
Si $r : A \to B, s : A \to C$	$\langle r, s \rangle \rightleftarrows \lambda x^A.\langle r\ x, s\ x \rangle$	($\text{DIST}_{\lambda\eta}$)
	$\lambda x^A.\lambda y^B.t \rightleftarrows \lambda z^{A \times B}.t[\pi_A(z)/x, \pi_B(z)/y]$	(CURRY)
Si $t : B \to C$	$\lambda x^A.t \rightleftarrows \lambda z^{A \times B}.t[\pi_A(z)/x]\ \pi_B(z)$	(CURRY_η)
	$\lambda x^{A \times B}.t \rightleftarrows \lambda y^A.\lambda z^B.t[\langle y, z \rangle / x]$	(UNCURRY)
	$\langle r, \star \rangle \rightleftarrows r$	($\text{ID}_\times$)
Si $r : \top \to A$	$r \rightleftarrows r \star$	($\text{ID}_\Rightarrow$)
	$r \rightleftarrows \lambda x^\top.r$	($\text{ID}_\Rightarrow$)
Si $r : A \to \top$	$r \rightleftarrows \star$	(ABS)
Si $t : \top$	$t \rightleftarrows \lambda x^A.t$	(ABS)

Figura 3.1: Reglas de equivalencias entre términos

Código 8. Relación de equivalencia entre términos

```
σ-cong⇒₁ : ∀ {Γ A B} → (iso : B ≡ A) → Subst (Γ , A) (Γ , B)
σ-cong⇒₁ iso = [ iso ]≡ (` Z) • (ids ∘ S_)


σ-uncurry : ∀ {Γ A B} → Subst (Γ , A × B) (Γ , A , B)
σ-uncurry = ⟨ ` S Z , ` Z ⟩ • λ x → ` (S (S x))
```

```
σ-curry : ∀ {Γ A B} → Subst (Γ , A , B) (Γ , A × B)
σ-curry = π B {inj₂ refl} (` Z) • π A {inj₁ refl} (` Z) • ids ∘ S_


data _⇄_ : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  comm : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B}
    → [ comm ]≡ ⟨ r , s ⟩ ⇄ ⟨ s , r ⟩
  sym-comm : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B}
    → [ sym comm ]≡ ⟨ r , s ⟩ ⇄ ⟨ s , r ⟩

  asso : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B} → {t : Γ ⊢ C}
    → [ asso ]≡ ⟨ r , ⟨ s , t ⟩ ⟩ ⇄ ⟨ ⟨ r , s ⟩ , t ⟩
  sym-asso : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B} → {t : Γ ⊢ C}
    → [ sym asso ]≡ ⟨ ⟨ r , s ⟩ , t ⟩ ⇄ ⟨ r , ⟨ s , t ⟩ ⟩
  asso-split : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ B × C}
    → [ asso ]≡ ⟨ r , s ⟩ ⇄ ⟨ ⟨ r , π B {inj₁ refl} s ⟩ , π C {inj₂ refl} s ⟩
  sym-asso-split : ∀ {r : Γ ⊢ A × B} → {s : Γ ⊢ C}
    → [ sym asso ]≡ ⟨ r , s ⟩ ⇄ ⟨ π A {inj₁ refl} r , ⟨ π B {inj₂ refl} r , s ⟩ ⟩

  dist-ƛ : ∀ {r : Γ , C ⊢ A} → {s : Γ , C ⊢ B}
    → [ dist ]≡ ⟨ ƛ r , ƛ s ⟩ ⇄ ƛ ⟨ r , s ⟩
  dist-ƛη_l_r : ∀ {r : Γ ⊢ C ⇒ A} → {s : Γ ⊢ C ⇒ B}
    → [ dist ]≡ ⟨ r , s ⟩ ⇄ ƛ ⟨ rename S_ r · ` Z , rename S_ s · ` Z ⟩
  dist-ƛη_l : ∀ {r : Γ ⊢ C ⇒ A} → {s : Γ , C ⊢ B}
    → [ dist ]≡ ⟨ r , ƛ s ⟩ ⇄ ƛ ⟨ rename S_ r · ` Z , s ⟩
  dist-ƛη_r : ∀ {r : Γ , C ⊢ A} → {s : Γ ⊢ C ⇒ B}
    → [ dist ]≡ ⟨ ƛ r , s ⟩ ⇄ ƛ ⟨ r , rename S_ s · ` Z ⟩
  sym-dist-ƛ : ∀ {r : Γ , C ⊢ A} → {s : Γ , C ⊢ B}
    → [ sym dist ]≡ (ƛ ⟨ r , s ⟩) ⇄ ⟨ ƛ r , ƛ s ⟩
  sym-dist-ƛ-split : ∀ {r : Γ , C ⊢ A × B}
    → [ sym dist ]≡ (ƛ r) ⇄ ⟨ ƛ π A {inj₁ refl} r , ƛ π B {inj₂ refl} r ⟩

  curry : ∀ {r : Γ , A , B ⊢ C}
    → [ curry ]≡ (ƛ ƛ r) ⇄ ƛ subst σ-curry r
  curry-η : ∀ {r : Γ , A ⊢ B ⇒ C}
    → [ curry ]≡ (ƛ r) ⇄ ƛ subst σ-curry (rename S_ r · ` Z)
  uncurry : ∀ {r : Γ , A × B ⊢ C}
    → [ sym curry ]≡ (ƛ r) ⇄ ƛ ƛ subst σ-uncurry r

  id-× : ∀ {r : Γ ⊢ A} → {t : Γ ⊢ ⊤}
    → [ id-× ]≡ ⟨ r , t ⟩ ⇄ r
  sym-id-× : ∀ {r : Γ ⊢ A}
    → [ sym id-× ]≡ r ⇄ ⟨ r , ⋆ ⟩

  id-⇒ : ∀ {r : Γ ⊢ ⊤ ⇒ A}
    → [ id-⇒ ]≡ r ⇄ r · ⋆
  sym-id-⇒ : ∀ {r : Γ ⊢ A}
    → [ sym id-⇒ ]≡ r ⇄ ƛ rename S_ r

  abs : ∀ {r : Γ ⊢ A ⇒ ⊤}
    → [ abs ]≡ r ⇄ ⋆
```

```
sym-abs : ∀ {t : Γ ⊢ ⊤}
  → [ sym abs ]≡ t ⇄ ƛ rename (S_ {A = A}) t

sym-sym : ∀ {r : Γ ⊢ A} → {iso : A ≡ B}
  → [ sym (sym iso) ]≡ r ⇄ [ iso ]≡ r

cong⇒₁ : ∀ {r : Γ , A ⊢ C} → {iso : A ≡ B}
  → [ cong⇒₁ iso ]≡ (ƛ r) ⇄ ƛ subst (σ-cong⇒₁ (sym iso)) r
sym-cong⇒₁ : ∀ {r : Γ , B ⊢ C} → {iso : A ≡ B}
  → [ sym (cong⇒₁ iso) ]≡ (ƛ r) ⇄ ƛ subst (σ-cong⇒₁ iso) r

cong⇒₂ : ∀ {r : Γ , C ⊢ A} → {iso : A ≡ B}
  → [ cong⇒₂ iso ]≡ (ƛ r) ⇄ ƛ ([ iso ]≡ r)
sym-cong⇒₂ : ∀ {r : Γ , C ⊢ B} → {iso : A ≡ B}
  → [ sym (cong⇒₂ iso) ]≡ (ƛ r) ⇄ ƛ ([ sym iso ]≡ r)

cong×₁ : ∀ {r : Γ ⊢ A} → {s : Γ ⊢ C} → {iso : A ≡ B}
  → [ cong×₁ iso ]≡ ⟨ r , s ⟩ ⇄ ⟨ [ iso ]≡ r , s ⟩
sym-cong×₁ : ∀ {r : Γ ⊢ B} → {s : Γ ⊢ C} → {iso : A ≡ B}
  → [ sym (cong×₁ iso) ]≡ ⟨ r , s ⟩ ⇄ ⟨ [ sym iso ]≡ r , s ⟩

cong×₂ : ∀ {r : Γ ⊢ C} → {s : Γ ⊢ A} → {iso : A ≡ B}
  → [ cong×₂ iso ]≡ ⟨ r , s ⟩ ⇄ ⟨ r , [ iso ]≡ s ⟩
sym-cong×₂ : ∀ {r : Γ ⊢ C} → {s : Γ ⊢ B} → {iso : A ≡ B}
  → [ sym (cong×₂ iso) ]≡ ⟨ r , s ⟩ ⇄ ⟨ r , [ sym iso ]≡ s ⟩

ξ-·₁ : ∀ {r r' : Γ ⊢ A ⇒ B} {s : Γ ⊢ A}
  → r ⇄ r'
  → r · s ⇄ r' · s

ξ-·₂ : ∀ {r : Γ ⊢ A ⇒ B} {s s' : Γ ⊢ A}
  → s ⇄ s'
  → r · s ⇄ r · s'

ξ-⟨,⟩₁ : ∀ {r r' : Γ ⊢ A} {s : Γ ⊢ B}
  → r ⇄ r'
  → ⟨ r , s ⟩ ⇄ ⟨ r' , s ⟩

ξ-⟨,⟩₂ : ∀ {r : Γ ⊢ A} {s s' : Γ ⊢ B}
  → s ⇄ s'
  → ⟨ r , s ⟩ ⇄ ⟨ r , s' ⟩

ξ-π : ∀ {t t' : Γ ⊢ A × B}
  → t ⇄ t'
  → π C {p} t ⇄ π C {p} t'

ξ-≡ : ∀ {t t' : Γ ⊢ A} {iso : A ≡ B}
  → t ⇄ t'
  → ([ iso ]≡ t) ⇄ ([ iso ]≡ t')
```

```
ζ : ∀ {t t' : Γ , B ⊢ A}
  → t ⇄ t'
  → ƛ t ⇄ ƛ t'
```

La mayoría de estas reglas tiene una definición intuitiva, sin embargo, algunas de ellas requieren una explicación más detallada para comprender su comportamiento:

- El constructor `asso-split` se aplica cuando se quiere asociar un par $\langle r, s\rangle$ donde el término de la derecha no tiene forma de par. El posible llevar el término s a una forma equivalente, donde luego sea posible aplicar la asociatividad $\langle r, s\rangle \rightleftarrows \langle r, \langle \pi_B(s), \pi_C(s)\rangle\rangle \rightleftarrows \langle\langle r, \pi_B(s)\rangle, \pi_C(s)\rangle$. La regla `asso-split` combina esto dos pasos en uno solo. Las demás reglas con el sufijo `split` tienen un comportamiento análogo.

- El constructor `dist-`$\lambda\eta_{lr}$ se aplica cuando se quiere mover las abstracciones fuera del producto, pero los elementos del par no tienen forma de abstracción. Por lo tanto, primero se debe η-expandir ambos lados del par y luego se extraen las abstracciones $\langle r, s\rangle \rightleftarrows \langle \lambda x^A.r\, x, \lambda x^A.s\, x\rangle \rightleftarrows \lambda x^A.\langle r\, x, s\, x\rangle$. Cabe destacar que a nivel de la implementación, la η-expansión del término `t` se escribe `ƛ rename S_ t · ‘ Z`, donde el renombre suma uno a todas las variables libres, por lo que el índice 0 pasa a estar fresco en t. La regla `dist-`$\lambda\eta_{lr}$ combina esto dos pasos en uno solo. Las demás reglas con el sufijo η tienen un comportamiento análogo.

- El constructor `curry` tiene la particularidad de que aparece una substitución en su definición $\lambda x^A.\lambda y^B.t \rightleftarrows \lambda z^{A\times B}.t[\pi_A(z)/x, \pi_B(z)/y]$, se debe notar que el término t tiene dos variables libres que deben ser substituidas por dos proyecciones de una única nueva variable. Coloquialmente, la substitución σ`-curry` reemplaza el índice 0 por la proyección $\pi_B(0)$, el índice 1 por la proyección $\pi_A(0)$, y le resta uno al resto de los índices, este último paso es necesario ya que se está eliminando uno de los lambdas que envuelven al término t.

- El constructor `uncurry` se define como $\lambda x^{A\times B}.t \rightleftarrows \lambda y^A.\lambda z^B.t[\langle y, z\rangle/x]$, en este caso, el término t tiene una sola variable libre que debe ser reemplazada por un par de nuevas variables. Coloquialmente, la substitución σ`-uncurry` reemplaza el índice 0 por el par $\langle 1, 0\rangle$, y le suma uno al resto de los índices, este último paso es necesario ya que se está añadiendo un nuevo lambda.

- El constructor `cong`$\Rightarrow_1$ cambia el tipo del argumento de una abstracción, para ello substituye el índice 0 por el término [iso]$\equiv$ 0 y deja el resto de los índices intactos. Es decir, el término resultante será indentico al original, solo que su primera variable libre estará envuelta por un nuevo constructor `[_]≡_`.

Al igual que en la relación de reducción, los tipos intrínsecos garantizan que los términos transformados por la relación `_⇄_` preservan sus tipos. Notar como todos los constructores presentan una regla ($\equiv$) del lado izquierdo, la cual o bien desaparece del lado derecho, o bien se simplifica el isomorfismo de tipo aplicado. Por ejemplo, el constructor `comm` elimina la regla `[ comm ]≡_` del término, mientras que el constructor `cong`$\Rightarrow_2$ simplifica el isomorfismo `[ cong`$\Rightarrow_2$ `iso ]≡_` a `[ iso ]≡_`. Por lo tanto, las equivalencias solo pueden aplicarse para eliminar o simplificar un constructor ($\equiv$) del término.

Ejemplo 9. El siguiente ejemplo muestra como la regla `cong`$\times_2$ se aplica para simplificar el isomorfismo de tipo, esto hace que luego sea posible aplicar la regla `comm`. Las congruencias funcionan de forma análoga a las de la relación `_↪_`

```
_ : [ cong×₂ comm ]≡ ⟨ ⋆ , ⟨ ƛ ‘ Z , ⋆ ⟩ ⟩ ⇄ ⟨ ⋆ , [ comm ]≡ ⟨ ƛ ‘ Z , ⋆ ⟩ ⟩
_ = cong×₂

_ : ⟨ ⋆ , [ comm ]≡ ⟨ ƛ ‘ Z , ⋆ ⟩ ⟩ ⇄ ⟨ ⋆ , ⟨ ⋆ , ƛ ‘ Z ⟩ ⟩
_ = ξ-⟨,⟩₂ comm
```

Ejemplo 10. El isomorfismo de término `sym-dist-ƛ` distribuye la abstracción sobre el par y elimina el constructor `[ sym dist ]≡` del término resultante.

```
T₄ : ∀ {A B} → ∅ ⊢ (A ⇒ B ⇒ A) × (A ⇒ B ⇒ B)
T₄ = [ sym dist ]≡ (ƛ [ sym dist ]≡ (ƛ ⟨ ‘ (S Z) , ‘ Z ⟩))

_ : T₄ ⇄ [ sym dist ]≡ (ƛ ⟨ ƛ ‘ (S Z) , ƛ ‘ Z ⟩)
_ = ξ-≡ (ζ sym-dist-ƛ)
```

3.2.1. Preservación de tipos

Es importante destacar que la representación con tipos intrínsecos provee implícitamente la propiedad de preservación. La relación de equivalencia entre términos `_⇄_` y la relación de reducción `_↪_` solo pueden relacionar términos del mismo tipo. En consecuencia, no puede darse el caso en el que un término reduzca, de forma errónea, a otro de un tipo distinto.

3.3. Progreso

3.3.1. Formas normales, neutrales y valores

La propiedad de progreso dice que todo término puede o bien dar un paso de reducción, o bien se encuentra en un estado final. Si solo se consideran términos cerrados, es decir, que no contienen variables libres, los estados finales son los valores. Existen tres constructores para los valores: los pares de valores, las abstracciones y el término ⋆.

Código 9. Tipo de dato de los valores

```
data Value : ∀ {Γ A} → Γ ⊢ A → Set where
  V-ƛ : ∀ {Γ A B} {t : Γ , A ⊢ B}
      ---------------------------
    → Value (ƛ t)

  V-⋆ : ∀ {Γ}
      --------------------------
    → Value (⋆ {Γ})

  V-⟨_,_⟩ : ∀ {Γ A B} {r : Γ ⊢ A} {s : Γ ⊢ B}
    → Value r
    → Value s
      ----------------
    → Value ⟨ r , s ⟩
```

Sin embargo, en la definición de progreso que se presentará a continuación, se implementa el orden de reducción *strong call-by-value*, es decir, se deben reducir los cuerpos de las abstracciones. Cuando se reduce debajo de las abstracciones es posible encontrar variables libres. Cuando se trabaja con términos que pueden contener variables libres, los estados finales que se deben considerar son las formas normales.

A continuación daremos una definición inductiva de las formas normales. Un término está en forma normal si no puede ser reducido. La siguiente sintaxis caracteriza a los términos en forma normal del lenguaje dado:

$$\begin{aligned} \mathbf{norm} &:= \langle \mathbf{norm}, \mathbf{norm} \rangle \mid \lambda x.\mathbf{norm} \mid \star \mid \mathbf{neu} \\ \mathbf{neu} &:= \mathit{var} \mid \pi\ \mathbf{neu} \mid \mathbf{neu} \cdot \mathbf{norm} \mid [\mathit{iso}] \equiv \mathbf{neu} \end{aligned}$$

Donde *var* son las variables y la categoría sintáctica *neu* caracteriza a las formas neutrales, que son términos que no pueden reducirse y no son valores. En la formalización se utilizan los símbolos $\Downarrow$ para las formas neutrales y $\Uparrow$ para las formas normales.

Código 10. Definición inductiva de las formas neutrales y normales

```
data ⇓ : ∀ {Γ A} → Γ ⊢ A → Set
data ⇑ : ∀ {Γ A} → Γ ⊢ A → Set

data ⇓ where
  ‘_ : ∀ {Γ A} (x : Γ ∋ A)
     -------------
    → ⇓ (‘ x)

  _·_ : ∀ {Γ A B} {r : Γ ⊢ A ⇒ B} {s : Γ ⊢ A}
    → ⇓ r → ⇑ s
      ---------------
    → ⇓ (r · s)

  π : ∀ {Γ A B C p} {t : Γ ⊢ A × B}
    → ⇓ t
      ---------------
    → ⇓ (π C {p} t)

  [_]≡_ : ∀ {Γ A B} {t : Γ ⊢ A}
    → (iso : A ≡ B)
    → ⇓ t
      ------------------
    → ⇓ ([ iso ]≡ t)


data ⇑ where
  ^_ : ∀ {Γ A} {t : Γ ⊢ A}
    → ⇓ t
      ---------
    → ⇑ t

  N-ƛ : ∀ {Γ A B} {t : Γ , A ⊢ B}
      ------------
```

```
  → ⇑ (ƛ t)

N-⟨_,_⟩ : ∀ {Γ A B} {r : Γ ⊢ A} {s : Γ ⊢ B}
  → ⇑ r → ⇑ s
    ----------------
  → ⇑ ⟨ r , s ⟩

N-⋆ : ∀ {Γ}
    --------------------------
  → ⇑ (⋆ {Γ})
```

Notar como las definiciones de ⇑ y ⇓ son mutuamente recursivas.

Se puede observar que en la definición de las formas neutrales, el único constructor que no es recursivo es el de las variables. Es decir que, para poder construir una forma neutral, necesariamente se debe utilizar el caso base de la variable al menos una vez. Por lo tanto, un término está en forma neutral solo si contiene variables libres. Luego, es fácil ver que todo término cerrado en forma normal es un valor.

Código 11. Todo término cerrado en forma normal es un valor

```
closed→¬⇓ : ∀ {A} → (t : ∅ ⊢ A) → ¬ (⇓ t)
closed→¬⇓ ⋆          = λ ()
closed→¬⇓ (ƛ _)      = λ ()
closed→¬⇓ ⟨ _ , _ ⟩  = λ ()
closed→¬⇓ ([ _ ]≡ x) = λ { ([ _ ]≡ ⇓x) → closed→¬⇓ x ⇓x }
closed→¬⇓ (x · _)    = λ { (⇓x · _) → closed→¬⇓ x ⇓x }
closed→¬⇓ (π _ x)    = λ { (π ⇓x) → closed→¬⇓ x ⇓x }

closed⇑→Value : ∀ {A} → {t : ∅ ⊢ A} → ⇑ t → Value t
closed⇑→Value N-⋆            = V-⋆
closed⇑→Value N-ƛ            = V-ƛ
closed⇑→Value N-⟨ a , b ⟩    = V-⟨ closed⇑→Value a , closed⇑→Value b ⟩
closed⇑→Value {t = t} (ˆ ⇓t) = ⊥-elim (closed→¬⇓ t ⇓t)
```

3.3.2. Estrategia de reducción

Para representar la propiedad de progreso se define una nueva relación `Progress` sobre los términos. Los constructores de esta relación se corresponden con los tres posibles casos en los cuales se cumple la propiedad de progreso para un término, estos son:

1. El término puede ser transformado a otro equivalente _⇄_.
2. El término puede dar un paso de reducción _↪_.
3. El término se encuentra en forma normal, en cuyo caso no es posible seguir reduciéndolo.

Código 12. Propiedad de progreso

```
data Progress {Γ A} (t : Γ ⊢ A) : Set where
  step⇄ : ∀ {t' : Γ ⊢ A}
```

```
  → t ⇄ t'
    ----------
  → Progress t

step↪ : ∀ {t' : Γ ⊢ A}
  → t ↪ t'
    ----------
  → Progress t

done :
    ⇑ t
    ----------
  → Progress t
```

Luego, la prueba de progreso consiste en probar que la propiedad `Progress` se cumple para todo término dado. La formalización implementa una estrategia de reducción *strong call-by-value*, es decir, reduce primero los argumentos de las aplicaciones antes de aplicar la β-reducción, y además, reduce debajo de los lambdas. El orden de reducción escogido no tiene importancia, la prueba puede ser realizada siguiendo cualquier orden, aquí se lo menciona simplemente para facilitar la comprensión del código.

Debido a la longitud de esta función se la presentará separada por casos. En primer lugar, se presentan los casos para las variables y el término $\star$, donde simplemente se concluye que ambos se encuentran en formal normal.

```
progress : ∀ {Γ A} → (t : Γ ⊢ A) → Progress t
progress (‘ x) = done (ˆ (‘ x))
progress ⋆     = done N-⋆
```

Para el caso de la abstracción, se realiza una llamada recursiva de `progress` sobre el cuerpo de la función, en caso de que esta nueva llamada retorne un paso, entonces se extiende dicho paso usando la congruencia ζ. En otro caso, se retorna que la abstracción está en forma normal.

```
progress (ƛ t) with progress t
... | step⇄ t⇄t' = step⇄ (ζ t⇄t')
... | step↪ t↪t' = step↪ (ζ t↪t')
... | done ⇑t     = done N-ƛ
```

Para los pares se sigue un razonamiento análogo con la diferencia de que se realizan dos llamadas recursivas, una por cada lado del par. Primero se realizan todos los pasos que sean posibles sobre el lado izquierdo, y luego sobre el lado derecho.

```
progress ⟨ r , s ⟩ with progress r
... | step⇄ r⇄r' = step⇄ (ξ-⟨,⟩₁ r⇄r')
... | step↪ r↪r' = step↪ (ξ-⟨,⟩₁ r↪r')
... | done ⇑r with progress s
...     | step⇄ s⇄s' = step⇄ (ξ-⟨,⟩₂ s⇄s')
...     | step↪ s↪s' = step↪ (ξ-⟨,⟩₂ s↪s')
...     | done ⇑s     = done N-⟨ ⇑r , ⇑s ⟩
```

Notar que las evidencias retornadas por las llamadas recursivas que indican que r y s están en forma normal, son utilizadas para probar que todo el par se encuentra en formal normal `N-⟨ ⇑r , ⇑s ⟩`.

Para las proyecciones se reduce primero el término que se está proyectando. Aquí se debe notar que existen dos posibles formas en las que la llamada recursiva puede alcanzar una forma normal. Por un lado, podría llegar a una forma neutral, en cuyo caso toda la proyección es neutral. Por otro lado, podría alcanzar un par en formal normal, en cuyo caso es posible proyectar dicho par.

```
progress (π _ {p} t) with progress t
... | step⇄ t⇄t' = step⇄ (ξ-π t⇄t')
... | step↪ t↪t' = step↪ (ξ-π t↪t')
... | done (^ ⇓t)  = done (^ π ⇓t)
progress (π _ {inj₁ refl} t) | done N-⟨ ⇑r , ⇑s ⟩ = step↪ β-π₁
progress (π _ {inj₂ refl} t) | done N-⟨ ⇑r , ⇑s ⟩ = step↪ β-π₂
```

La aplicación sigue la misma lógica, solo que en este caso se reduce primero el término de la izquierda, luego el término de la derecha, y, cuando ambos términos llegan a una forma normal, se aplica la β-reducción. Sin embargo, si la función se encuentra en formal neutral, entonces no es posible realizar la aplicación, por lo que todo el término será neutral.

```
progress (r · s) with progress r
... | step⇄ r⇄r'     = step⇄ (ξ-·₁ r⇄r')
... | step↪ r↪r'     = step↪ (ξ-·₁ r↪r')
progress (r · s) | done (^ ⇓r) with progress s
...     | step⇄ s⇄s' = step⇄ (ξ-·₂ s⇄s')
...     | step↪ s↪s' = step↪ (ξ-·₂ s↪s')
...     | done ⇑s    = done (^ (⇓r · ⇑s))
progress ((λ _) · s) | done N-λ with progress s
...     | step⇄ s⇄s' = step⇄ (ξ-·₂ s⇄s')
...     | step↪ s↪s' = step↪ (ξ-·₂ s↪s')
...     | done ⇑s    = step↪ β-λ
```

Por último se presenta el constructor `[ iso ]≡ t`. Al igual que en el resto de los casos, primero se reduce el subtérmino `t`. Luego, para cada forma normal se deben tener en cuenta todos los isomorfismos que sean aplicables en cada caso.

```
progress ([ iso ]≡ t) with progress t
... | step⇄ t⇄t' = step⇄ (ξ-≡ t⇄t')
... | step↪ t↪t' = step↪ (ξ-≡ t↪t')
progress ([ comm ]≡ _)     | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ comm
progress ([ sym comm ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ (sym-comm)

progress ([ asso ]≡ _)     | done N-⟨ ⇑r , N-⟨ ⇑s , ⇑t ⟩ ⟩ = step⇄ asso
progress ([ asso ]≡ _)     | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ (asso-split)
progress ([ sym asso ]≡ _) | done N-⟨ N-⟨ ⇑r , ⇑s ⟩ , ⇑t ⟩ = step⇄ (sym-asso)
progress ([ sym asso ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ (sym-asso-split)

progress ([ dist ]≡ _) | done N-⟨ N-λ , N-λ ⟩ = step⇄ dist-λ
progress ([ dist ]≡ _) | done N-⟨ N-λ , ⇑s ⟩  = step⇄ (dist-λη_r)
progress ([ dist ]≡ _) | done N-⟨ ⇑r , N-λ ⟩  = step⇄ (dist-λη_l)
progress ([ dist ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩   = step⇄ (dist-λη_lr)
progress ([ sym dist ]≡ (λ ⟨ ⇑r , ⇑s ⟩)) | done N-λ = step⇄ (sym-dist-λ)
progress ([ sym dist ]≡ (λ ⇑t))          | done N-λ = step⇄ (sym-dist-λ-split)
```

```
progress ([ curry ]≡ (λ λ _)) | done N-λ = step⇄ curry
progress ([ curry ]≡ _)       | done N-λ = step⇄ (curry-η)
progress ([ sym curry ]≡ _)   | done N-λ = step⇄ uncurry

progress ([ id-× ]≡ _)      | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ id-×
progress ([ id-⇒ ]≡ _)      | done ⇑t            = step⇄ id-⇒
progress ([ sym id-× ]≡ _)  | done ⇑t            = step⇄ (sym-id-×)
progress ([ sym id-⇒ ]≡ _) | done ⇑t            = step⇄ sym-id-⇒

progress ([ abs ]≡ _)     | done ⇑t = step⇄ abs
progress ([ sym abs ]≡ _) | done ⇑t = step⇄ (sym-abs)

progress ([ cong⇒₁ iso ]≡ _) | done N-λ = step⇄ cong⇒₁
progress ([ cong⇒₂ iso ]≡ _) | done N-λ = step⇄ cong⇒₂
progress ([ cong×₁ iso ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ cong×₁
progress ([ cong×₂ iso ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ cong×₂
progress ([ sym (cong⇒₁ iso) ]≡ _) | done N-λ = step⇄ sym-cong⇒₁
progress ([ sym (cong⇒₂ iso) ]≡ _) | done N-λ = step⇄ sym-cong⇒₂
progress ([ sym (cong×₁ iso) ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ sym-cong×₁
progress ([ sym (cong×₂ iso) ]≡ _) | done N-⟨ ⇑r , ⇑s ⟩ = step⇄ sym-cong×₂

progress ([ sym (sym iso) ]≡ _) | done ⇑t = step⇄ sym-sym
progress ([ iso ]≡ _) | done (ˆ ⇓t) = done (ˆ [ iso ]≡ ⇓t)
```

Notar como solo es posible aplicar `step⇄ asso` cuando el elemento de la derecha es un par. En caso de que dicho elemento esté en forma neutral, es necesario convertirlo a un par aplicando la regla $r \rightleftarrows \langle \pi_A(r), \pi_B(r) \rangle$ antes de poder aplicar `asso`. Por ejemplo, para reducir el siguiente término:

```
_ : Γ , A × B ⊢ (⊤ × A) × B
_ = [ asso ]≡ ⟨ ⋆ , ‘ Z ⟩
```

Se requiere aplicar el isomorfismo `asso-split` que combina ambas reglas en un solo paso:

```
_ : [ asso ]≡ ⟨ ⋆ , ‘ Z ⟩ ⇄ ⟨ ⟨ ⋆ , π A (‘ Z) ⟩ , π B (‘ Z) ⟩
_ = asso-split
```

Lo mismo ocurre con el resto de los isomorfismos que presentan los sufijos `split` y η.

Finalmente, si el término `t` es neutral, se concluye que todo el término también es neutral.

La definición de `progress` muestra claramente como cada isomorfismo de término se corresponde con un caso donde es necesario eliminar una regla ($\equiv$) para poder continuar con la reducción. En la formalización, el lenguaje es dirigido por sintaxis. Cuando se representan los términos usando tipos intrínsecos, la sintaxis del lenguaje contiene los isomorfismos de tipos. Por lo tanto, tiene sentido que la aplicación de los isomorfismos de términos también sea dirigida por sintaxis.

Otra observación interesante es que gracias a la dualidad entre pruebas y programas que propone el paradigma de Proposiciones como Tipos, implementar y demostrar resultan ser la misma tarea. Por lo tanto, la prueba `progress` es también un programa, que al ser aplicado sobre un término retorna el siguiente paso de reducción correspondiente a dicho término.

Ejemplo 11. Aplicar progreso sobre un término puede tener como resultado un paso de reducción, la aplicación de un isomorfismo, o devolver la evidencia de que el término ya se encuentra en forma normal

```
_ : progress T₃ ≅ step↪ (ξ-·₁ β-π₁) -- (π ⟨ λ 0 , ⋆ ⟩) · 0 ↪ (λ 0) · 0
_ = refl

_ : progress T₄ ≅ step⇄ (ξ-≡ (ζ sym-dist-λ)) -- λ λ ⟨ 1 , 0 ⟩ ⇄ λ ⟨ λ 1 , λ 0 ⟩
_ = refl

_ : progress T₂ ≅ done ⟨ N-λ , N-⋆ ⟩ -- ⟨ λ 0 , ⋆ ⟩ está en forma normal
_ = refl
```

3.4. Evaluación

A través de la aplicación sucesiva de la prueba de progreso se puede definir la evaluación. Para esto, se define la relación _⇝*_, que representa la clausura reflexiva y transitiva de la unión entre las relaciones _↪_ y _⇄_.

Código 13. Clausura reflexiva y transitiva de la unión entre las relaciones _↪_ y _⇄_.

```
data _⇝*_ {Γ A} : (Γ ⊢ A) → (Γ ⊢ A) → Set where
  _∎ : (t : Γ ⊢ A)
     ------
   → t ⇝* t

_⇄⟨_⟩_ : (t : Γ ⊢ A) {r t' : Γ ⊢ A}
   → t ⇄ r
   → r ⇝* t'
     ------
   → t ⇝* t'

_↪⟨_⟩_ : (t : Γ ⊢ A) {r t' : Γ ⊢ A}
   → t ↪ r
   → r ⇝* t'
     ------
   → t ⇝* t'
```

La evaluación se define sobre términos cerrados, de modo que estos siempre reduzcan a un valor. La definición aplica la prueba de progreso para obtener el nuevo término sobre el cual hacer el llamado recursivo y el paso de reducción que extenderá la relación _⇝*_.

Debido a que la recursión se realiza sobre el término resultante luego de un paso, Agda no tiene forma de determinar que dicho término es estructuralmente más pequeño que el argumento inicial. De hecho, probar la terminación de la evaluación, implicaría que todo término reduce a un valor en una cantidad finita de pasos, es decir, se estaría probando la propiedad de normalización. La prueba de dicha propiedad no es trivial, y se presentará en la siguiente sección.

Por este motivo, es necesario añadir un argumento de tipo $\mathbb{N}$ para que la definición sea aceptada por el chequeo de terminación de Agda. De este modo, la evaluación realizará a lo sumo n llamadas recursivas. Además, se utiliza `Maybe` en el tipo de retorno, en caso de que el término no converja a un valor luego de n pasos, simplemente se retornará `nothing`.

Código 14. Evaluación

```
data Steps {A} : ∅ ⊢ A → Set where
  steps : {t t' : ∅ ⊢ A}
    → t ⇝* t'
    → Maybe (Value t')
      ----------
    → Steps t

eval : ∀ {A} → (t : ∅ ⊢ A) → ℕ → Steps t
eval t zero = steps (t ■) nothing
eval t (suc gas) with progress t
eval t _          | done ⇑t              = steps (t ■) (just (closed⇑→Value ⇑t))
eval t (suc gas) | step⇄ {r} t⇄r with eval r gas
...                  | steps r⇝t' fin = steps (t ⇄⟨ t⇄r ⟩ r⇝t') fin
eval t (suc gas) | step↪ {r} t↪r with eval r gas
...                  | steps r⇝t' fin = steps (t ↪⟨ t↪r ⟩ r⇝t') fin
```

Ejemplo 12. Cuando la cantidad de pasos máxima es suficiente, la evaluación retorna la secuencia de reducción y la evidencia de que el término final es un valor

```
N : ∅ ⊢ ⊤
N = π _ ([ sym dist ]≡ ([ curry ]≡ (ƛ ƛ ⟨ ‘ S Z , ‘ Z ⟩))) · ⟨ ★ , ƛ ‘ Z ⟩
-- N = π (λ λ ⟨ 1 , 0 ⟩) · ⟨ ★ , λ 0 ⟩

N⇝★ : N ⇝* ★
N⇝★ =
    N
      ⇄⟨ ξ-·₁ (ξ-π (ξ-≡ curry)) ⟩
    π _ ([ sym dist ]≡ (ƛ ⟨ π ⊤ (‘ Z) , π _ (‘ Z) ⟩)) · ⟨ ★ , ƛ ‘ Z ⟩
    -- π (λ ⟨ π 0 , π 0 ⟩) · ⟨ ★ , λ 0 ⟩
      ⇄⟨ ξ-·₁ (ξ-π sym-dist-ƛ) ⟩
    π _ ⟨ ƛ π ⊤ (‘ Z) , ƛ π _ (‘ Z) ⟩ · ⟨ ★ , ƛ ‘ Z ⟩
    -- π ⟨ λ π 0 , λ π 0 ⟩ · ⟨ ★ , λ 0 ⟩
      ↪⟨ ξ-·₁ β-π₁ ⟩
    (ƛ π ⊤ (‘ Z)) · ⟨ ★ , ƛ ‘ Z ⟩
    -- λ π 0 · ⟨ ★ , λ 0 ⟩
      ↪⟨ β-ƛ ⟩
    π ⊤ ⟨ ★ , ƛ ‘ Z ⟩
    -- π ⟨ ★ , λ 0 ⟩
      ↪⟨ β-π₁ ⟩
    ★
  ■

_ : eval' N 10 ≅ steps N⇝★ (just V-★)
_ = refl
```

Ejemplo 13. En caso de que la cantidad de pasos no sea suficiente para arribar a un valor, la evaluación retorna la secuencia de reducción incompleta y el constructor `nothing`

```
¬N⇝⋆ : N ⇝* _
¬N⇝⋆ =
    N
      ⇄⟨ ξ-·₁ (ξ-π (ξ-≡ curry)) ⟩
    π _ ([ sym dist ]≡ (ƛ ⟨ π ⊤ (‘ Z) , π _ (‘ Z) ⟩)) · ⟨ ⋆ , ƛ ‘ Z ⟩
      ⇄⟨ ξ-·₁ (ξ-π sym-dist-ƛ) ⟩
    π _ ⟨ ƛ π ⊤ (‘ Z) , ƛ π _ (‘ Z) ⟩ · ⟨ ⋆ , ƛ ‘ Z ⟩
      ↪⟨ ξ-·₁ β-π₁ ⟩
    (ƛ π ⊤ (‘ Z)) · ⟨ ⋆ , ƛ ‘ Z ⟩
  ∎

_ : eval’ N 3 ≅ steps ¬N⇝⋆ nothing
_ = refl
```

3.5. Normalización Fuerte

La propiedad de normalización fuerte dice que para cualquier término dado, toda secuencia de reducción eventualmente concluye, sin importar el camino elegido. O dicho de otra forma, no es posible construir secuencias de reducción infinitas.

El conjunto de términos fuertemente normalizantes puede definirse de manera inductiva mediante la siguiente regla:

$$\frac{\forall t.s \rightsquigarrow t \implies t \in SN}{s \in SN}$$

Es decir, si todos los términos t, a los cuales puede reducir s son fuertemente normalizantes, entonces s también lo es.

En Agda se puede definir el conjunto SN mediante el siguiente tipo de datos inductivo:

Código 15. Propiedad de normalización fuerte

```
_⇝_ : ∀ {Γ A} → (t t’ : Γ ⊢ A) → Set
t ⇝ t’ = t ↪ t’ ⊎ t ⇄ t’

data SN {Γ A} (t : Γ ⊢ A) : Set where
  sn : (∀ {t’} → t ⇝ t’ → SN t’) → SN t
```

Notar que es trivial construir un término de tipo `SN` para un valor, ya que, al no existir ninguna reducción posible para los valores, simplemente se debe pasar la función vacía como argumento a `sn`. Estos representan los casos base del tipo `SN`. Notar que la finitud de las secuencias de reducción termina siendo dada por la caracterización inductiva, dado que para construir un término de tipo `SN` se deben aplicar una cantidad necesariamente finita de constructores `sn`.

Ejemplo 14. Cuando se construye un término de tipo `SN t`, para cierto término `t`, se deben tener en cuenta todos los pasos de reducción posibles a partir de `t`. Para términos normales, todos los pasos concluyen en un patrón absurdo. En los casos que si es posible aplicar una eliminación, se debe construir recursivamente `SN` para el término resultante.

```
SN⋆ : SN ⋆
SN⋆ = sn (λ ())
```

```
SN⟨,⟩ : SN ⟨ ⋆ , ‘ Z ⟩
SN⟨,⟩ = sn λ { (ξ-⟨,⟩₁ ())
             ; (ξ-⟨,⟩₂ ())}

SNπ : SN (π ⊤ {inj₁ refl} ⟨ ⋆ , ‘ Z ⟩)
SNπ = sn λ { (ξ-π (ξ-⟨,⟩₁ ()))
           ; (ξ-π (ξ-⟨,⟩₂ ()))
           ; β-π₁ → SN⋆}
```

El hecho de que exista un término de tipo `SN t` implica que no existen secuencias de reducción infinitas a partir de `t`, de otro modo, no hubiera sido posible construir dicho término. El objetivo final de esta sección será entonces, definir una función que dado un término cualquiera, permita obtener un término de tipo `SN t`:

```
strong-norm : ∀ {Γ A} (t : Γ ⊢ A) → SN t
```

La prueba aquí presentada está basada y extendida a partir de una formalización realizada por Andras Kovacs[1] para cálculo lambda simplemente tipado. A su vez, el trabajo de Kovacs se basa en la tesis de Steven Schäfer [Sch19], donde se formaliza la prueba de normalización fuerte para Sistema F en Coq. La técnica utilizada por Schäfer difiere en ciertos aspectos con la prueba de candidatos de reducibilidad de Girard [GLT89], en el capítulo 5.6 de [Sch19] se discuten en detalle estas diferencias. Además, se utilizó como referencia una formalización de candidatos de reducibilidad en Agda realizada por Pablo Barenbaum[2].

3.5.1. Prueba para STLC con pares y Top

Con el objetivo de facilitar la comprensión de la prueba, la misma será presentada en dos partes. En esta sección se explicará la prueba para cálculo lambda simplemente tipado con pares, es decir, sin incluir la relación de equivalencia entre términos. Y en la siguiente sección se presentarán los cambios necesarios para extender la prueba a Sistema I con tipo Top.

Por lo tanto, se define el tipo de datos `SN` utilizando solo la relación `_↪_`

```
data SN {Γ A} (t : Γ ⊢ A) : Set where
  sn : (∀ {t'} → t ↪ t' → SN t') → SN t
```

La idea de la prueba es definir una generalización del tipo `SN` que añade un predicado sobre el término `t`. Como se explicará más adelante, el objetivo de este predicado es añadir información extra sobre el término cuando se construyen las introducciones, y luego usar esa información cuando se prueban los casos de las eliminaciones.

```
data SN* {Γ A} (P : Γ ⊢ A → Set) (t : Γ ⊢ A) : Set where
  sn* : P t → (∀ {t'} → t ↪ t' → SN* P t') → SN* P t
```

Es fácil ver que si un término cumple con el predicado `SN*` también cumple con `SN`

```
SN*-SN : ∀ {Γ P A} {t : Γ ⊢ A} → SN* P t → SN t
SN*-SN (sn* _ SNt) = sn (λ step → SN*-SN (SNt step))
```

[1] https://github.com/AndrasKovacs/misc-stuff/blob/master/agda/STLCStrongNorm/StrongNorm2.agda
[2] https://github.com/foones/formal-proofs/blob/master/strong-normalization/reducibility-candidates.agda

Luego, se define lo que se denomina *interpretación del término*. Esta interpretación añade algunas hipótesis extra que permiten desatascar la prueba para los casos de las eliminaciones (proyecciones y aplicaciones). Para el caso de los productos, la interpretación provee evidencia de que los términos a y b del par son fuertemente normalizantes. Mientras que para el caso de la abstracción, la interpretación se define de modo que dado cualquier término u fuertemente normalizante, permite concluir que $\rho(t)[u]$ es fuertemente normalizante, donde t es el cuerpo de la abstracción.

```
open import Data.Product renaming (_,_ to ⟨_,_⟩; _×_ to _⊗_)
open import Data.Unit renaming (⊤ to Top)

⟦_⟧ : ∀ {Γ A} → Γ ⊢ A → Set
⟦ (ƛ t) ⟧     = ∀ {Δ}{ρ : Rename _ Δ}{u} → SN* ⟦_⟧ u → SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ t)
⟦ ⟨ a , b ⟩ ⟧ = SN* ⟦_⟧ a ⊗ SN* ⟦_⟧ b
⟦ t ⟧         = Top
```

La definición utiliza los productos `_×_` y el tipo top `⊤` definidos en los módulos `Data.Product` y `Data.Unit` de la librería estándar. Para evitar una colisión de nombres con los constructores de tipos del cálculo, se renombran los tipos de la librería estándar como `_⊗_` y `Top` respectivamente.

Se puede extender la definición de interpretación a substituciones, para ello se define un predicado que establece que una substitución σ es adecuada en un contexto Γ, y se escribe $\Gamma \vDash \sigma$, cuando todos los términos por los cuales son substituidas las variables en σ cumplen con el predicado `SN* ⟦_⟧` .

```
_⊨_ : ∀{Δ} → (Γ : Context) → (σ : Subst Γ Δ) → Set
Γ ⊨ σ = ∀{A} (v : Γ ∋ A) → SN* ⟦_⟧ (σ {A} v)
```

En particular, la substitución identidad `ids` es una substitución adecuada

```
lemma-var : ∀ {Γ A}{v : Γ ∋ A} → SN* {Γ}{A} ⟦_⟧ (` v)
lemma-var = sn* tt (λ ())

⊨ids : ∀{Γ} → Γ ⊨ ids
⊨ids _ = lemma-var
```

Toda la complejidad de la prueba se centra en probar el teorema fundamental, el cual implica que para todo término t y substitución adecuada σ se cumple `SN* ⟦_⟧ (⟪ σ ⟫ t)`. La prueba de este teorema corresponde a la definición de la función `adequacy`, que se define por inducción sobre el término t. A continuación se presentará la función mostrando su definición para cada uno de los constructores de términos.

```
adequacy : ∀ {Γ Δ A} {σ : Subst Γ Δ} → (t : Γ ⊢ A) → Γ ⊨ σ → SN* ⟦_⟧ (⟪ σ ⟫ t)
```

La prueba de este teorema es extensa y requiere probar algunos lemas extra, que a su vez utilizan propiedades de las substituciones y renombres. Para simplificar la explicación, se presenta la prueba separada por casos y solo se mostrará el código completo de los lemas más relevantes, mientras que para los lemas auxiliares solo se presentará su tipo.

Caso variable

Este caso requiere que la substitución σ aplicada a la variable v cumpla con la propiedad `SN* ⟦_⟧` . La hipótesis dice que σ es una substitución adecuada, por lo tanto, basta con simplemente aplicar dicha hipótesis a la variable v.

```
adequacy (‘ v) ⊨σ = ⊨σ v
```

Caso top

Debido a que no existe ningún paso de reducción posible para el término $\star$, la construcción del predicado es trivial.

```
lemma-⊤ : ∀ {Γ} → SN* {Γ} ⟦_⟧ ⋆
lemma-⊤ = sn* tt (λ ())

adequacy ⋆ _ = lemma-⊤
```

Caso producto

Para el caso del producto, se puede definir un lema que permite construir el predicado si se tiene como hipótesis que `SN* ⟦_⟧` vale para los subtérminos a y b del par.

```
lemma-⟨,⟩ : ∀ {Γ A B} → {a : Γ ⊢ A} {b : Γ ⊢ B} →
                                    SN* ⟦_⟧ a → SN* ⟦_⟧ b → SN* ⟦_⟧ (⟨ a , b ⟩)
lemma-⟨,⟩ SN*a SN*b = sn* ( SN*a , SN*b ) λ step → aux SN*a SN*b step
  where aux : ∀ {Γ A B} {a : Γ ⊢ A} {b : Γ ⊢ B} {t' : Γ ⊢ A × B} →
          SN* ⟦_⟧ a → SN* ⟦_⟧ b → ⟨ a , b ⟩ ↪ t' → SN* ⟦_⟧ t'
        aux (sn* La SNa) SN*b (ξ-⟨,⟩₁ step) = lemma-⟨,⟩ (SNa step) SN*b
        aux SN*a (sn* Lb SNb) (ξ-⟨,⟩₂ step) = lemma-⟨,⟩ SN*a (SNb step)
```

Para construir el segundo argumento de `sn*` se define una función auxiliar `aux` que toma cualquier paso de reducción y muestra que el reducto es `SN* ⟦_⟧` . En este caso, para el paso de reducción `step` existen dos posibilidades, la congruencia a izquierda o la congruencia a derecha del par:

- En el caso de la congruencia a izquierda, el paso `step` tiene forma $a \hookrightarrow a'$ por lo que se debe mostrar que $\langle a', b\rangle$ es `SN* ⟦_⟧` . Para esto se realiza una llamada recursiva al lema. El lema requiere `SN* ⟦_⟧ a'`, por lo que se utiliza la hipótesis `SNa`, la cual dice que todo reducto de `a` es `SN* ⟦_⟧` , en conjunto con el paso de reducción $a \hookrightarrow a'$.
- El caso de la congruencia a derecha es análogo al primero, la diferencia es que se usa la hipótesis `SNb` para obtener `SN* ⟦_⟧ b'`.

Luego, se utiliza la hipótesis inductiva en `adequacy` sobre a y b para instanciar el lema

```
adequacy ⟨ a , b ⟩ ⊨σ = lemma-⟨,⟩ (adequacy a ⊨σ) (adequacy b ⊨σ)
```

Caso proyección

El caso de la proyección también requiere como hipótesis que se cumpla el predicado para el término que se está proyectando. Los pasos de reducción $\beta\text{-}\pi_1$ y $\beta\text{-}\pi_2$ implican que t tiene forma $\langle a, b\rangle$, estos requieren obtener `SN* ⟦_⟧ a` y `SN* ⟦_⟧ b` respectivamente. El problema es que no se puede concluir nada sobre a y b a partir de `SN t`. Aquí se vuelve evidente por qué es necesario generalizar la propiedad `SN` y añadir la interpretación del término. La interpretación de un par

$\langle a, b\rangle$ tiene forma `SN* ⟦_⟧ a ⊗ SN* ⟦_⟧ b` y esto es precisamente lo que se necesita para probar ambos casos.

```
lemma-π : ∀ {Γ A B C p} → {t : Γ ⊢ A × B} → SN* ⟦_⟧ t → SN* ⟦_⟧ (π C {p} t)
lemma-π SN*t = sn* tt (aux SN*t)
  where aux : ∀ {Γ A B C p t'} → {t : Γ ⊢ A × B} →
          SN* ⟦_⟧ t → (π C {p} t) ↪ t' → SN* ⟦_⟧ t'
        aux (sn* _ SNt) (ξ-π step) = lemma-π (SNt step)
        aux (sn* ( SN*a , _ ) _) β-π₁ = SN*a
        aux (sn* ( _ , SN*b ) _) β-π₂ = SN*b
```

El paso de reducción tiene tres constructores posibles:

- El caso de la congruencia se resuelve de la misma forma que para los pares.
- El caso de la proyección izquierda implica que t tiene forma $\langle a, b\rangle$ y se debe mostrar que el reducto a es `SN* ⟦_⟧` , para ello se utiliza la hipótesis `SN*a` de la interpretación del par.
- El caso de la proyección derecha se resuelve de forma análoga usando la hipótesis `SN*b` .

```
adequacy (π _ x) ⊨σ = lemma-π (adequacy x ⊨σ)
```

Caso aplicación

Este caso es similar al producto, ya que la hipótesis del lema requiere que el predicado se cumpla para ambos subtérminos. Por otro lado, la solución para el caso de la β-reducción es similar al caso de la proyección, puesto que se utiliza la interpretación de la abstracción.

```
lemma-· : ∀ {Γ A B} → {a : Γ ⊢ A ⇒ B} {b : Γ ⊢ A} →
  SN* ⟦_⟧ a → SN* ⟦_⟧ b → SN* ⟦_⟧ (a · b)
lemma-· SN*a SN*b = sn* tt λ step → aux SN*a SN*b step
  where aux : ∀ {Γ A B} {a : Γ ⊢ B ⇒ A} {b : Γ ⊢ B} {t' : Γ ⊢ A} →
          SN* ⟦_⟧ a → SN* ⟦_⟧ b → a · b ↪ t' → SN* ⟦_⟧ t'
        aux (sn* _ SNa) SN*b (ξ-·₁ step) = lemma-· (SNa step) SN*b
        aux SN*a (sn* _ SNb) (ξ-·₂ step) = lemma-· SN*a (SNb step)
        aux (sn* La SNa) SN*b β-λ        = La SN*b
```

El paso de reducción tiene tres posibles constructores, congruencia a izquierda, congruencia a derecha y la β-reducción:

- Las congruencias se resuelve del mismo modo que las presentadas anteriormente, usando las hipótesis `SNa` y `SNb` respectivamente.
- El caso de la β-reducción implica que el término a tiene forma $\lambda\ t$ y se debe mostrar que $t[b]$ es `SN* ⟦_⟧` . La interpretación `La` será de la forma `SN* ⟦_⟧ u → SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ t )` es decir que basta con aplicar dicha función a `SN*b` .

```
adequacy (a · b) ⊨σ = lemma-· (adequacy a ⊨σ) (adequacy b ⊨σ)
```

Notar cómo las interpretaciones de los términos permiten resolver los casos de eliminación (proyección y aplicación), pero, por otro lado, los casos de introducción (producto y abstracción) se vuelven más complejos, ya que son estos donde se construyen dichas interpretaciones. Esta construcción es simple para los productos, puesto que se puede deducir de forma directa aplicando la hipótesis inductiva en `adequacy`. Sin embargo, como se verá más adelante, obtener la interpretación correspondiente al caso de las abstracciones, resulta particularmente complejo.

Caso abstracción

La prueba del lema para el caso de la abstracción requiere obtener la interpretación $[\![\ \lambda\ t'\ ]\!]$ en la llamada inductiva. Para ello es necesario definir un nuevo lema que permite aplicar una substitución a ambos lados de un paso de reducción.

```
↪[] : ∀ {Γ Δ A}{t t' : Γ ⊢ A}{σ : Subst Γ Δ} → t ↪ t' → ⟪ σ ⟫ t ↪ ⟪ σ ⟫ t'
```

A su vez, la prueba de dicho lema requiere una propiedad de la substitución para poder probar el caso correspondiente a la β-reducción. La propiedad `subst-commute` dice que la substitución conmuta consigo misma:

```
subst-commute : ∀{Γ Δ A B}{t : Γ , B ⊢ A}{u : Γ ⊢ B}{σ : Subst Γ Δ}
  → (⟪ exts σ ⟫ t) [ ⟪ σ ⟫ u ] ≡ ⟪ σ ⟫ (t [ u ])
```

Luego se utiliza la función `↪[]` para definir el lema correspondiente a las abstracciones.

```
↪SN* : ∀ {Γ A}{t t' : Γ ⊢ A} → t ↪ t' → SN* [_] t → SN* [_] t'
↪SN* step (sn* _ SNt) = SNt step

lemma-ƛ : ∀ {Γ A B} → {t : Γ , B ⊢ A} → [ ƛ t ] → SN* [_] t → SN* [_] (ƛ t)
lemma-ƛ Lƛ (sn* _ SNt) =
  sn* Lƛ (λ { (ζ t↪t') →
               lemma-ƛ (λ SNu → ↪SN* (↪[] t↪t') (Lƛ SNu)) (SNt t↪t') })
```

La instanciación de las hipótesis necesarias para utilizar el lema de la abstracción en la prueba de `adequacy` requiere especial atención.

Por un lado, para obtener la hipótesis `SN* [_] t`, primero se debe probar que la extensión de una substitución adecuada sigue siendo adecuada.

```
⊨exts : ∀{Γ Δ A} {σ : Subst Γ Δ} → Γ ⊨ σ → (Γ , A) ⊨ (exts σ)
⊨exts σ Z     = lemma-var
⊨exts σ (S v) = SN*-rename S_ (σ v)
```

A su vez, para esta prueba es necesario demostrar el lema `SN*-rename` el cual dice que si se aplica un renombre de variables a un término `SN* [_]` este no perderá dicha propiedad, esto tiene sentido, ya que los renombres no alteran el significado de los términos y, por lo tanto, preservan la propiedad `SN* [_]`

```
SN*-rename : ∀{Γ Δ A} {t : Γ ⊢ A} →
  (ρ : Rename Γ Δ) → SN* [_] t → SN* [_] (rename ρ t)
```

Lógicamente, la prueba de dicho lema requiere probar que dados un renombre de variables ρ y un término $[\![t]\!]$, se puede concluir $[\![$ `rename` $\rho\ t]\!]$.

```
[]-rename : ∀{Γ Δ A} {t : Γ ⊢ A} → (ρ : Rename Γ Δ) → [ t ] → [ rename ρ t ]
[]-rename {A = A ⇒ B} {t = ƛ n} ρ Ln {_}{ρ₁}{u}
  rewrite rename-subst {Σ = ∅} {M = n} {N = u} {ρ = ρ} {σ = (ids ∘ ρ₁)}
    = λ SNu → Ln {_} {ρ₁ ∘ ρ} SNu
[]-rename {t = ⟨ a , b ⟩} ρ ( SN*a , SN*b )
  = ( SN*-rename ρ SN*a , SN*-rename ρ SN*b )
[]-rename {t = ‘ v} ρ tt   = tt
```

```
⟦⟧-rename {t = a · b} ρ tt = tt
⟦⟧-rename {t = ⋆} ρ tt     = tt
⟦⟧-rename {t = π _ t} ρ tt = tt
⟦⟧-rename {t = [ _ ]≡ t} ρ tt = tt
```

Aquí se debe realizar una observación importante. El caso de la abstracción en la prueba de `⟦⟧-rename` es el único lugar donde es necesario usar el argumento ρ de la interpretación. De hecho, este es precisamente el motivo por el cual es necesario generalizar la interpretación de las lambdas añadiendo un renombre en su definición.

Por otro lado, para obtener la hipotiposis `⟦ ƛ t' ⟧` en el lema de la abstracción se necesita demostrar dos propiedades más sobre las substituciones adecuadas.

Una de ella dice que una substitución adecuada compuesta con un renombre, es también una substitución adecuada. La prueba utiliza el hecho de que `rename` ρ puede ser escrito como `⟪ ids ∘ ρ ⟫`

```
⊨rename : ∀{Γ Δ Δ₁} {σ : Subst Γ Δ} →
  Γ ⊨ σ → (ρ : Rename Δ Δ₁) → Γ ⊨ (⟪ ids ∘ ρ ⟫ ∘ σ)
```

La otra propiedad, dice que el cons entre un término `SN* ⟦_⟧` y una substitución adecuada, es también una substitución adecuada.

```
⊨_•_ : ∀{Γ Δ A} {σ : Subst Γ Δ} {t : Δ ⊢ A} →
  SN* ⟦_⟧ t → Γ ⊨ σ → (Γ , A) ⊨ (t • σ)
(⊨ t • σ) Z     = t
(⊨ t • σ) (S v) = σ v
```

Un último detalle a destacar es que `adequacy` toma como argumento solo una substitución, por lo que se debe combinar σ y u `•` `(ids ∘ ρ)` en una sola substitución y demostrar que aplicar dicha combinación es equivalente a aplicarlas por separado.

```
subst-split :
  ∀ {Γ Δ Δ₁ A B}{t : Γ , A ⊢ B}{u : Δ₁ ⊢ A}{σ : Subst Γ Δ}{ρ : Rename Δ Δ₁}
    → ⟪ u • (⟪ ids ∘ ρ ⟫ ∘ σ) ⟫ t ≡ ⟪ u • (ids ∘ ρ) ⟫ (⟪ exts σ ⟫ t)
```

Usando los teoremas presentados ahora es posible definir el caso de `adequacy` para la abstracción.

```
adequacy {σ = σ} (ƛ t) ⊨σ =
  lemma-ƛ
    (λ { {ρ = ρ}{u = u} SNu →
      transport (SN* ⟦_⟧)
        (subst-split {t = t})
        (adequacy t (⊨ SNu • (⊨rename ⊨σ ρ)))})
    (adequacy t (⊨exts ⊨σ))
```

La función `transport` permite obtener $P\ y$ a partir de una prueba de $x \cong y$ y una instancia de $P\ x$. Esto es útil cuando se tiene como hipótesis `SN* ⟦_⟧` x y se debe concluir `SN* ⟦_⟧` y, en dicho caso simplemente se utiliza `transport` con una prueba de que x y y son equivalentes. Esta función está definida en el módulo `Relation.Binary.PropositionalEquality` con el nombre `subst`, aquí se la renombra a `transport` para evitar una colisión de nombres con la substitución.

Cierre de la prueba

Finalmente, se prueba la propiedad de normalización fuerte instanciando `adequacy` con la substitución identidad.

Como detalle adicional, se elimina la substitución identidad aplicada al término utilizando el lema `sub-id : ∀{Γ A} {t : Γ ⊢ A} → ⟪ ids ⟫ t ≡ t`

```
strong-norm : ∀ {Γ A} (t : Γ ⊢ A) → SN t
strong-norm t = transport SN sub-id (SN*-SN (adequacy t ⊨ids))
```

3.5.2. Prueba para Sistema I con tipo Top

El primer paso para extender la prueba, es agregar en la definición de `SN`, la relación de equivalencia entre términos.

```
_⇝_ : ∀ {Γ A} → (t t' : Γ ⊢ A) → Set
t ⇝ t' = t ↪ t' ⊎ t ⇄ t'

data SN {Γ A} (t : Γ ⊢ A) : Set where
  sn : (∀ {t'} → t ⇝ t' → SN t') → SN t


data SN* {Γ A} (P : Γ ⊢ A → Set) (t : Γ ⊢ A) : Set where
  sn* : P t → (∀ {t'} → t ⇝ t' → SN* P t') → SN* P t
```

Este cambio tiene como consecuencia que todas las partes de la prueba que construían un término `SN* ⟦_⟧` haciendo *pattern matching* sobre la relación ↪, ahora deberán tener también en cuenta los constructores de la relación ⇄ que sean aplicables en cada caso.

Por ejemplo, en la prueba del lema de la abstracción ahora aparecen los dos constructores $\texttt{inj}_1$ e $\texttt{inj}_2$ del tipo suma.

```
↪SN* : ∀ {Γ A}{t t' : Γ ⊢ A} → t ↪ t' → SN* ⟦_⟧ t → SN* ⟦_⟧ t'
↪SN* step (sn* _ SNt) = SNt (inj₁ step)
⇄SN* : ∀ {Γ A}{t t' : Γ ⊢ A} → t ⇄ t' → SN* ⟦_⟧ t → SN* ⟦_⟧ t'
⇄SN* step (sn* _ SNt) = SNt (inj₂ step)

lemma-ƛ : ∀ {Γ A B} → {t : Γ , B ⊢ A} → ⟦ ƛ t ⟧ → SN* ⟦_⟧ t → SN* ⟦_⟧ (ƛ t)
lemma-ƛ {t = t} Lƛ (sn* _ SNt) =
  sn* Lƛ (λ { (inj₁ (ζ t↪t')) → lemma-ƛ
                                  (λ SNu → ↪SN* (↪[] t↪t') (Lƛ SNu))
                                  (SNt (inj₁ t↪t'))
            ; (inj₂ (ζ t⇄t')) → lemma-ƛ
                                  (λ SNu → ⇄SN* (⇄[] t⇄t') (Lƛ SNu))
                                  (SNt (inj₂ t⇄t')) })
```

Este caso también requiere la definición de un nuevo lema análogo al definido para la relación ↪ anteriormente.

```
⇄[] : ∀ {Γ Δ A}{t t' : Γ ⊢ A}{σ : Subst Γ Δ} → t ⇄ t' → ⟪ σ ⟫ t ⇄ ⟪ σ ⟫ t'
```

La extensión del resto de los lemas no presentan mayores dificultades. A modo de ejemplo se presenta el lema correspondiente al producto, y se omiten los demás.

```
lemma-⟨,⟩ : ∀ {Γ A B} → {a : Γ ⊢ A} {b : Γ ⊢ B} →
  SN* ⟦_⟧ a → SN* ⟦_⟧ b → SN* ⟦_⟧ (⟨ a , b ⟩)
lemma-⟨,⟩ SN*a SN*b = sn* ( SN*a , SN*b ) λ step → aux SN*a SN*b step
  where aux : ∀ {Γ A B} {a : Γ ⊢ A} {b : Γ ⊢ B} {t' : Γ ⊢ A × B} →
          SN* ⟦_⟧ a → SN* ⟦_⟧ b → ⟨ a , b ⟩ ⇝ t' → SN* ⟦_⟧ t'
        aux (sn* La SNa) SN*b (inj₁ (ξ-⟨,⟩₁ step))
                                          = lemma-⟨,⟩ (SNa (inj₁ step)) SN*b
        aux SN*a (sn* Lb SNb) (inj₁ (ξ-⟨,⟩₂ step))
                                          = lemma-⟨,⟩ SN*a (SNb (inj₁ step))
        aux (sn* La SNa) SN*b (inj₂ (ξ-⟨,⟩₁ step))
                                          = lemma-⟨,⟩ (SNa (inj₂ step)) SN*b
        aux SN*a (sn* Lb SNb) (inj₂ (ξ-⟨,⟩₂ step))
                                          = lemma-⟨,⟩ SN*a (SNb (inj₂ step))
```

Notar que ahora el paso de reducción tiene cuatro constructores posibles, dos congruencias correspondientes a la relación $\hookrightarrow$ y dos correspondientes a $\rightleftarrows$.

El lema para el constructor de isomorfismos es donde se concentra la mayor complejidad de la prueba, ya que se deben resolver los casos de cada una de las equivalencias de términos. Para empezar, las congruencias se resuelven de forma análoga a las presentadas anteriormente, la dificultad está en que cuando se realiza *pattern matching* sobre el paso de reducción en la función `aux`, se deben resolver todos los casos correspondientes a cada uno de los constructores del tipo $\rightleftarrows$.

Debido a la extensión de este lema, se presentará la función `aux` dividida en secciones y solo se mostrarán los casos más relevantes.

```
lemma-≡ : ∀ {Γ A B iso} → {t : Γ ⊢ A} → SN* ⟦_⟧ t → SN* {A = B} ⟦_⟧ ([ iso ]≡ t)
lemma-≡ SN*t = sn* tt (aux SN*t)
  where aux : ∀ {Γ A iso t'} → {t : Γ ⊢ A} →
          SN* ⟦_⟧ t → ([ iso ]≡ t) ⇝ t' → SN* ⟦_⟧ t'
        aux (sn* _ SNt) (inj₁ (ξ-≡ step)) = lemma-≡ (SNt (inj₁ step))
        aux (sn* _ SNt) (inj₂ (ξ-≡ step)) = lemma-≡ (SNt (inj₂ step))
```

Luego, el objetivo para cada constructor será obtener `SN* ⟦_⟧ t'` a partir de `SN* ⟦_⟧ t`, donde `t` y `t'` son los términos relacionados por el isomorfismo correspondiente. La idea de la prueba es utilizar los lemas definidos anteriormente para los distintos constructores de términos.

Caso comm

Por ejemplo, el isomorfismo COMM $\langle r, s\rangle \rightleftarrows \langle s, r\rangle$ es uno de los casos más simples que ilustra la técnica utilizada. Aquí simplemente basta con instanciar el lema del producto utilizando las hipótesis provistas por la interpretación del par.

```
aux (sn* ( SNr , SNs ) _) (inj₂ comm)     = lemma-⟨,⟩ SNs SNr
aux (sn* ( SNr , SNs ) _) (inj₂ sym-comm) = lemma-⟨,⟩ SNs SNr
```

Caso asso

El caso de ASSO $\langle r, \langle s, t\rangle\rangle \rightleftarrows \langle\langle r, s\rangle, t\rangle$ no tiene mayor dificultad. Notar cómo se aprovecha la naturaleza constructiva de las pruebas en Agda, combinando los lemas ya definidos hasta obtener un término del tipo esperado.

```
aux (sn* ( SNr , sn* ( SNs , SNt ) _ ) _) (inj₂ asso)
                    = lemma-⟨,⟩ (lemma-⟨,⟩ SNr SNs) SNt
aux (sn* ( sn* ( SNr , SNs ) _ , SNt ) _) (inj₂ sym-asso)
                    = lemma-⟨,⟩ SNr (lemma-⟨,⟩ SNs SNt)
aux (sn* ( SNr , SNs ) _) (inj₂ asso-split)
                    = lemma-⟨,⟩ (lemma-⟨,⟩ SNr (lemma-π SNs)) (lemma-π SNs)
aux (sn* ( SNr , SNs ) _) (inj₂ sym-asso-split)
                    = lemma-⟨,⟩ (lemma-π SNr) (lemma-⟨,⟩ (lemma-π SNr) SNs)
```

Caso dist

Este caso presenta la dificultad de instanciar y construir la interpretación de la abstracción. Además, se presentan dos nuevos lemas que se emplearán en múltiples isomorfismos. En primer lugar, se define un lema que permite concluir `SN* ⟦_⟧ t` a partir de `SN* ⟦_⟧ (λ t)`. La idea del lema es substituir la primera variable del término por el índice cero, obteniendo de esa forma exactamente el mismo término.

```
lemma-sub : ∀ {Γ A B} → {t : Γ , B ⊢ A} → SN* ⟦_⟧ (λ t) → SN* ⟦_⟧ t
```

Este lema resulta de utilidad en los casos en los que se debe construir algo a partir del cuerpo de una abstracción, en particular, permite resolver el caso de `dist-λ` $\langle \lambda x^A.r, \lambda x^A.s\rangle \rightleftarrows \lambda x^A.\langle r, s\rangle$.

```
aux (sn* ( (sn* Lr SNr) , (sn* Ls SNs) ) _) (inj₂ dist-λ) =
  lemma-λ
    (λ SNu → lemma-⟨,⟩ (Lr SNu) (Ls SNu))
    (lemma-⟨,⟩ (lemma-sub (sn* Lr SNr)) (lemma-sub (sn* Ls SNs)))
```

El lema de la abstracción tiene dos argumentos:

- El primero es la interpretación `⟦ λ ⟨ r , s ⟩ ⟧`, es decir, se debe mostrar que para todo término `SN* ⟦_⟧ u`, se cumple `SN* ⟦_⟧ (⟨ ⟪ u • (ids ∘ ρ) ⟫ r , ⟪ u • (ids ∘ ρ) ⟫ s ⟩)`. Las hipótesis `Lr` y `Ls` corresponden a las interpretaciones `⟦ λ r ⟧` y `⟦ λ s ⟧` respectivamente, por lo que pueden ser utilizadas para obtener `SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ r)` y `SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ s)`.
- El segundo argumento requiere mostrar `SN* ⟦_⟧ ⟨ r , s ⟩`. Esto puede obtenerse simplemente aplicando el lema `lemma-sub` sobre las hipótesis.

En segundo lugar, se define un lema que se empleará para probar la propiedad de la interpretación de la abstracción, en los casos donde se deba η-expandir un término.

```
lemma-S : ∀ {Γ Δ A B} → {t : Γ ⊢ A} {u : Δ ⊢ B} →
  SN* ⟦_⟧ t → (ρ : Rename Γ Δ) → SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ (rename S_ t))
```

Utilizando este lema se pueden resolver los casos correspondientes a los constructores que combinan la η-expansión con el isomorfismo DIST. Se presenta a modo de ejemplo el caso del constructor `dist-λη`$_{lr}$ $\langle r, s\rangle \rightleftarrows \lambda x^A.\langle r\, x, s\, x\rangle$.

```
aux (sn* ( SNr , SNs ) _) (inj₂ dist-λη_lr) =
  lemma-λ
    (λ { {ρ = ρ} SNu → lemma-⟨,⟩
```

```
                    (lemma-· (lemma-S SNr ρ) SNu)
                    (lemma-· (lemma-S SNs ρ) SNu) })
  (lemma-⟨,⟩
    (lemma-· (SN*-rename S_ SNr) lemma-var)
    (lemma-· (SN*-rename S_ SNs) lemma-var))
```

Para obtener los argumentos del lema de la abstracción se sigue un razonamiento similar al caso anterior. Los puntos a destacar son:

- En el primer argumento se utiliza el nuevo lema `lemma-S` para obtener `SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ (rename S_ r))` a partir de *SNr*, y lo mismo aplica para *SNs*.
- En el segundo argumento se utiliza el lema `SN*-rename` para obtener `SN* ⟦_⟧ (rename S_ r)` a partir de *SNr*, y lo mismo aplica para *SNs*.

Caso congruencia abstracción

El siguiente caso presenta un isomorfismo donde aparece una substitución en el término de la derecha. Coloquialmente, la substitución σ-cong$\Rightarrow_1$ *iso* = `[ iso ]≡ (` Z) • (ids ∘ S_) reemplaza el primer índice del término por `[ iso ]≡ (` Z) y deja el resto de los índices intactos. La hipótesis *Lt* permite obtener un término de la forma `SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ t)`, entonces, la estrategia será hallar un renombre ρ y un término u que den como resultado el término esperado en cada caso.

```
aux (sn* Lt _) (inj₂ (cong⇒₁ {t = t}{iso = iso})) =
  lemma-λ
    (λ { {ρ = ρ} {u = u} SNu →
      transport (SN* ⟦_⟧)
        (sym (subst-cong⇒₁-split {t = t}))
        (Lt {ρ = ρ} {u = [ sym iso ]≡ (u)} (lemma-≡ SNu)) })
    (Lt {ρ = S_} {u = [ sym iso ]≡ (` Z)} (lemma-≡ lemma-var))
```

Para instanciar el lema de la abstracción, las dos hipótesis requeridas en este caso son:

- Una prueba de que para todo término `SN* ⟦_⟧ u` se cumple `SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ (⟪ [ iso ]≡ (` Z) • (ids ∘ S_) ⟫ t))`. Se debe notar que es posible combinar las dos substituciones en una sola `SN* ⟦_⟧ (⟪ [iso]≡ u • (ids ∘ ρ) ⟫ t)`, y esta última expresión es posible obtenerla a partir de *Lt*, tomando $\rho = \rho$ y $u =$ `[iso]≡ u`.
- Una prueba de `SN* ⟦_⟧ ⟪ [ iso ]≡ (` Z) • (ids ∘ S_) ⟫ t`, que puede obtenerse directamente a partir de *Lt*, tomando $\rho =$ `S_` y $u =$ `[iso]≡ (` Z).

Es importante entender cómo se pueden combinar múltiples substituciones en una sola, que pueda ser obtenida a través de la interpretación de la abstracción. Esta técnica será la clave para resolver los constructores correspondientes al isomorfismo CURRY, los cuales también presentan substituciones.

Caso curry

La estrategia para los dos constructores correspondientes a este isomorfismo será, en primer lugar, definir un lema que pruebe el primer argumento requerido por `lemma-λ`, y luego aprovechar el hecho de que `⟪ ` Z • ids ∘ S_ ⟫ t ≡ t` para concluir el segundo argumento.

El constructor curry $\lambda x^A.\lambda y^B.t \rightleftarrows \lambda z^{A\times B}.t[\pi_A(z)/x, \pi_B(z)/y]$ tiene la particularidad de que la interpretación de su hipótesis tiene la forma ⟦ ƛ ƛ t ⟧ , por lo tanto, a partir de un término SN* ⟦_⟧ u_1 se puede concluir SN* ⟦_⟧ (ƛ (⟪ exts (u_1 • (ids ∘ ρ_1)) ⟫ t)) , a su vez, este término contiene una segunda interpretación de la forma ⟦ ƛ (⟪ exts (u_1 • (ids ∘ ρ_1)) ⟫ t) ⟧, la cual permite concluir a partir de otro término SN* ⟦_⟧ u_2:

$$\texttt{SN*}\ [\![\_]\!]\ (\langle\!\langle\ u_2 \bullet (\texttt{ids} \circ \rho_2)\ \rangle\!\rangle\ (\langle\!\langle\ \texttt{exts}\ (u_1 \bullet (\texttt{ids} \circ \rho_1))\ \rangle\!\rangle\ t))$$

Por otro lado, el objetivo será utilizar el lemma-ƛ para probar SN* ⟦_⟧ (ƛ ⟪ σ-curry ⟫ t). En el primer argumento del lema de la abstracción se debe mostrar que para todo término SN* ⟦_⟧ u se cumple SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ (⟪ σ-curry ⟫ t). Aquí se debe notar que es posible reescribir esta substitución de una forma equivalente:

$$\begin{aligned}
&\langle\!\langle\ u \bullet (\texttt{ids} \circ \rho)\ \rangle\!\rangle\ (\langle\!\langle\ \sigma\text{-curry}\ \rangle\!\rangle\ t) \\
=\ &\langle\!\langle\ u \bullet (\texttt{ids} \circ \rho)\ \rangle\!\rangle\ (\langle\!\langle\ \pi\ B\ (\texttt{`}\ \texttt{Z}) \bullet \pi\ A\ (\texttt{`}\ \texttt{Z}) \bullet (\texttt{ids} \circ \texttt{S_})\ \rangle\!\rangle\ t) \\
=\ &\langle\!\langle\ \pi\ B\ u \bullet \texttt{ids}\ \rangle\!\rangle\ (\langle\!\langle\ \texttt{exts}\ (\pi\ A\ u \bullet \texttt{ids} \circ \rho)\ \rangle\!\rangle\ t)
\end{aligned}$$

Es posible obtener esta última expresión utilizando las dos interpretaciones anidadas presentadas en el párrafo anterior.

En simples palabras, las hipótesis dicen que dados dos términos u_1 y u_2 con "buen comportamiento", el resultado de reemplazar las dos variables libres de t por dichos términos $t[u_1/x, u_2/y]$, también tendrá "buen comportamiento". Luego, el objetivo del primer argumento del lema de la abstracción es probar que para todo término u con "buen comportamiento", el resultado de reemplazar la primera variable libre del término $t[\pi_A(z)/x, \pi_B(z)/y]$ por u, es decir, $(t[\pi_A(z)/x, \pi_B(z)/y])[u/z] = t[\pi_A(u)/x, \pi_B(u)/y]$, también tendrá "buen comportamiento". Por lo que basta con aplicar las hipótesis tomando $u_1 = \pi_A(u)$ y $u_2 = \pi_B(u)$.

```
lemma-curry : ∀ {Γ Δ A B C} → {t : Γ , A , B ⊢ C} {u : Δ ⊢ A × B} →
  ⟦ ƛ ƛ t ⟧ → SN* ⟦_⟧ u → (ρ : Rename Γ Δ) →
  SN* ⟦_⟧ (⟪ u • (ids ∘ ρ) ⟫ (⟪ σ-curry ⟫ t))
lemma-curry {A = A} {B = B} {t = t} {u = u} Lt SNu ρ =
  case Lt {ρ = ρ} {u = π A {inj₁ refl} u} (lemma-π SNu) of λ {(sn* Lr _) →
    transport (SN* ⟦_⟧)
      (subst-curry-split {t = t})
      (Lr {ρ = λ x → x} {u = π B {inj₂ refl} u} (lemma-π SNu))}
```

Notar que en lemma-curry primero se instancia la hipótesis Lt con $u = \pi\ A\ (\texttt{`}\ \texttt{Z})$ para obtener Lr = ⟦ ƛ (⟪ exts ($\pi\ A\ u$ • ids ∘ $\rho$) ⟫ t) ⟧, y luego se instancia Lr con $u = \pi\ B\ (\texttt{`}\ \texttt{Z})$.

```
aux (sn* Lt _) (inj₂ (curry {A = A}{B = B}{t = t})) =
  lemma-ƛ
    (λ {_}{ρ}{u} SNu → lemma-curry {t = t} Lt SNu ρ)
    (transport (SN* ⟦_⟧)
      (Z-weaken)
      (lemma-curry {t = t} {u = ` Z} Lt lemma-var S_))
```

Para el caso uncurry $\lambda x^{A\times B}.t \rightleftarrows \lambda y^A.\lambda z^B.t[\langle y, z\rangle/x]$ se debe realizar, en cierta forma, un razonamiento opuesto. Si en el caso anterior se utilizaron dos hipótesis anidadas para instanciar un solo lemma-ƛ, en este caso se utilizará una sola hipótesis para instanciar dos lemma-ƛ anidados.

El primer lema requiere obtener:

```
SN* ⟦_⟧ (⟪ exts (u₁ • (ids ∘ ρ₁)) ⟫ (⟪ σ-uncurry ⟫ t))
```

mientras que el lema anidado requiere:

```
SN* ⟦_⟧ (⟪ u₂ • (ids ∘ ρ₂) ⟫ (⟪ exts (u₁ • (ids ∘ ρ₁)) ⟫ (⟪ σ-uncurry ⟫ t)))
```

Aquí es donde se debe hallar una substitución equivalente que pueda ser obtenida a través de la interpretación ⟦ ƛ t ⟧. Notar que la primera substitución aplicada es extendida, por lo que primero se reemplaza el segundo índice por el término u_1. Luego, se substituye el primer índice por el término u_2 y se aplica el renombre ρ_2, por lo que u_1 será renombrado. Por lo tanto, la substitución equivalente es `SN* ⟦_⟧ (⟪ ⟨ rename ρ₂ u₁ , u₂ ⟩ • ids ∘ ρ₂ ∘ ρ₁ ⟫ t)`, esta expresión es posible obtenerla a través de la única interpretación dada por hipótesis.

Coloquialmente, la hipótesis dice que dado un término u de tipo par con "buen comportamiento", el resultado de reemplazar la primera variable libre de t por dicho término $t[u/x]$, también tendrá "buen comportamiento". Luego, el objetivo del primer lema de la abstracción es probar que para todo término u_1 con "buen comportamiento", el resultado de reemplazar la primera variable libre del término $\lambda z^B.t[\langle y, z\rangle/x]$ por u_1, es decir, $(\lambda z^B.t[\langle y, z\rangle/x])[u_1/y] = \lambda z^B.t[\langle u_1, z\rangle/x]$, también tendrá "buen comportamiento". Mientras que el objetivo del segundo lema de la abstracción será mostrar que para todo término u_2 con "buen comportamiento", el resultado de reemplazar la primera variable libre de $t[\langle u_1, z\rangle/x]$ por u_2, es decir, $(t[\langle u_1, z\rangle/x])[u_2/z] = t[\langle u_1, u_2\rangle/x]$, también tendrá "buen comportamiento". Por lo tanto, basta con aplicar la hipótesis tomando $u = \langle u_1, u_2\rangle$.

```
lemma-uncurry : ∀ {Γ Δ A B C} → {t : Γ , A × B ⊢ C} {u₁ : Δ ⊢ A} →
  ⟦ ƛ t ⟧ → SN* ⟦_⟧ u₁ → (ρ₁ : Rename Γ Δ) →
  SN* ⟦_⟧ (⟪ u₁ • (ids ∘ ρ₁) ⟫ (ƛ ⟪ σ-uncurry ⟫ t))
lemma-uncurry {Δ = Δ} {A = A} {B = B} {t = t} {u₁ = u₁} Lt SNu₁ ρ₁ =
  lemma-ƛ
    (λ {_}{ρ₂}{u₂} SNu₂ → lemma-uncurry₂ SNu₂ ρ₂)
    (transport (SN* ⟦_⟧)
      (Z-weaken)
      (lemma-uncurry₂ (lemma-var {v = Z}) S_))
  where
    lemma-uncurry₂ : ∀ {Δ₁} → {u₂ : Δ₁ ⊢ B} →
      SN* ⟦_⟧ u₂ → (ρ₂ : Rename Δ Δ₁) →
      SN* ⟦_⟧ (⟪ u₂ • ids ∘ ρ₂ ⟫ (⟪ exts (u₁ • ids ∘ ρ₁) ⟫ (⟪ σ-uncurry ⟫ t)))
    lemma-uncurry₂ {u₂ = u₂} SNu₂ ρ₂ =
      transport (SN* ⟦_⟧)
        (sym (subst-uncurry-split {t = t}))
        (Lt {ρ = ρ₂ ∘ ρ₁} {u = ⟨ rename ρ₂ u₁ , u₂ ⟩}
          (lemma-⟨,⟩ (SN*-rename ρ₂ SNu₁) SNu₂))
```

Notar que en `lemma-uncurry₂` se instancia a una única hipótesis `Lt` con $u = \langle$ `rename` $\rho_2\ u_1$, $u_2 \rangle$ y $\rho = \rho_2 \circ \rho_1$.

```
aux (sn* Lt _) (inj₂ (uncurry {t = t})) =
  lemma-ƛ
    (λ {_}{ρ}{u} SNu → lemma-uncurry {t = t} Lt SNu ρ)
    (transport (SN* ⟦_⟧)
      (Z-weaken)
      (lemma-uncurry {t = t} {u = ‘ Z} Lt lemma-var S_))
```

El resto de isomorfismos se omiten, ya que se resuelven aplicando las mismas técnicas que en los casos anteriores.

3.5.3. Relación bien fundada

Una vez demostrada la propiedad de normalización fuerte, se puede definir la función de evaluación sin necesidad de utilizar un argumento extra para pasar el chequeo de terminación de Agda.

Código 16. Evaluación de un término

```
data Steps {A} : ∅ ⊢ A → Set where
  steps : {t t' : ∅ ⊢ A}
    → t ⟶* t'
    → Value t'
      ----------
    → Steps t

eval´ : ∀ {A} → (t : ∅ ⊢ A) → SN t → Steps t
eval´ t _ with progress t
eval´ t _       | done ⇑t              = steps (t ∎) (closed⇑→Value ⇑t)
eval´ t (sn f) | step⇄ {r} t⇄r with eval´ r (f (inj₂ t⇄r))
...              | steps r⟶t' fin = steps (t ⇄⟨ t⇄r ⟩ r⟶t') fin
eval´ t (sn f) | step↪ {r} t↪r with eval´ r (f (inj₁ t↪r))
...              | steps r⟶t' fin = steps (t ↪⟨ t↪r ⟩ r⟶t') fin

eval : ∀ {A} → (t : ∅ ⊢ A) → Steps t
eval t = eval´ t (strong-norm t)
```

Notar que ya no aparece `Maybe` en el tipo de retorno, puesto que se ha probado que toda reducción eventualmente concluye. Aquí la llamada recursiva elimina un constructor `sn` del segundo argumento, por lo tanto, dicho argumento se hace estructuralmente más pequeño en cada llamada. De esta forma, Agda puede garantizar que la función alcanzará un caso base y la recursión concluirá eventualmente.

Ejemplo 15. El siguiente ejemplo muestra una de las formas de construir el término Ω y su reducción. Si bien existen varias alternativas para tipar este término, todas se basan en usar los isomorfismos `abs` o `id-⇒` para lograr la autoaplicación de un término de tipo $\top$.

Esto no supone ningún problema para la preservación de la propiedad de normalización, ya que, por un lado, para poder aplicar la regla β-λ el término de la izquierda debe ser una abstracción, es decir, un término de la forma $\lambda\ t$. Por lo tanto, antes de poder aplicar un término `([ iso ]≡ r) · s` se debe eliminar el constructor `[]≡` del lado izquierdo aplicando una equivalencia, es decir, necesariamente se debe transformar el término de la izquierda a otro equivalente.

Por otro lado, los términos producidos por las equivalencias `sym-abs` y `sym-id-⇒` añaden una nueva lambda que no captura ninguna variable dentro del cuerpo de la abstracción, esto no tiene ninguna utilidad real, ya que al ser aplicadas simplemente descartan su argumento.

```
-- Ω = (λx:T.xx)(λx:T.xx) : T
Ω : ∅ ⊢ ⊤
Ω = (ƛ [ sym abs ]≡ (` Z) · ` Z) · ([ abs ]≡ (ƛ [ sym abs ]≡ (` Z) · ` Z))
```

```
Ω⇝⋆ : Ω ⇝* ⋆
Ω⇝⋆ =
      Ω
        ⇄⟨ ξ-·₁ (ζ (ξ-·₁ sym-abs)) ⟩
      (ƛ (ƛ ‘ (S Z)) · ‘ Z) · ([ abs ]≡ (ƛ ([ sym abs ]≡ (‘ Z)) · ‘ Z))
      -- (λx. (λy. x) · x) · (λx. x · x)
        ↪⟨ ξ-·₁ (ζ β-ƛ) ⟩
      (ƛ ‘ Z) · ([ abs ]≡ (ƛ ([ sym abs ]≡ (‘ Z)) · ‘ Z))
      -- (λx. x) · (λx. x · x)
        ⇄⟨ ξ-·₂ (ξ-≡ (ζ (ξ-·₁ sym-abs))) ⟩
      (ƛ ‘ Z) · ([ abs ]≡ (ƛ (ƛ ‘ (S Z)) · ‘ Z))
      -- (λx. x) · (λx. (λy. x) · x)
        ↪⟨ ξ-·₂ (ξ-≡ (ζ β-ƛ)) ⟩
      (ƛ ‘ Z) · ([ abs ]≡ (ƛ ‘ Z))
      -- (λx. x) · (λx. x)
        ⇄⟨ ξ-·₂ abs ⟩
      (ƛ ‘ Z) · ⋆
      -- (λx. x) · ⋆
        ↪⟨ β-ƛ ⟩
      ⋆
  ∎

_ : eval Ω ≅ steps Ω⇝⋆ V-⋆
_ = refl
```

La forma en la que se codifica la propiedad de normalización fuerte, y el hecho de que sea justamente esta propiedad la que permite realizar inducción sobre la relación $\rightsquigarrow$, no son casualidad. Para entender los fundamentos que se esconden detrás de estas observaciones se presentan las siguientes definiciones.

En teoría del orden, se dice que una relación está bien fundada cuando todas sus cadenas, de la forma $x_1 < x_2 < \ldots < x_n$, tienen un largo acotado. Esta propiedad se denomina accesibilidad de un elemento.

Código 17. Accesibilidad de un término x en la relación $\_ < \_$

```
data Acc {A : Set a} (_<_ : Rel A ℓ) (x : A) : Set (a ⊔ ℓ) where
  acc : (∀ y → y < x → Acc _<_ y) → Acc _<_ x
```

Un elemento $x : A$ es accesible, si todo elemento $y < x$ es también accesible. Luego, una relación está bien fundada si todo elemento es accesible.

Código 18. Relación bien fundada

```
WellFounded : Rel A ℓ → Set _
WellFounded _<_ = ∀ x → Acc _<_ x
```

Una relación bien fundada soporta inducción, y en el contexto de los sistemas de reescritura, cumple con la propiedad de terminación. De hecho, la propiedad de normalización fuerte implica que la relación de reducción está bien fundada, y por lo tanto, soporta inducción. Se puede demostrar de forma sencilla esta implicancia.

Código 19. Normalización fuerte permite concluir que la relación $\rightsquigarrow$ está bien fundada

```
_↜_ : ∀ {Γ A} → (t' t : Γ ⊢ A) → Set
t' ↜ t = t ⇝ t'

WF-↜ : ∀ {Γ A} → WellFounded (_↜_ {Γ} {A})
WF-↜ t = SN→Acc (strong-norm t)
  where
    SN→Acc : ∀ {Γ A} {t : Γ ⊢ A} → SN t → Acc _↜_ t
    SN→Acc (sn f) = acc (λ _ t'↜t → SN→Acc (f t'↜t))
```

Capítulo 4

Conclusiones

En este trabajo se analizaron los aportes previos realizados en torno a la familia de sistemas módulo isomorfismos, en particular Sistema I. Luego, se realizó una formalización en Agda de dicho sistema, donde además se incorporó el tipo $\top$ y los tres isomorfismos correspondientes a este nuevo tipo. La formalización presenta algunas características interesantes, siendo una de ellas la representación de términos intrínsecamente tipados. En esta representación, la propiedad de preservación de tipos es inherente al sistema, y tiene como consecuencia que la aplicación de los isomorfismos de términos resulta dirigida por sintaxis. En cierta forma, la relación $\rightleftarrows$ elimina los isomorfismos de tipo del término, del mismo modo que la relación $\hookrightarrow$ elimina las aplicaciones y proyecciones, este punto es fundamental para la terminación del cálculo, ya que evita que algunos isomorfismos de término, como por ejemplo $\langle r, s\rangle \rightleftarrows \langle s, r\rangle$, sean aplicados un número indefinido de veces.

Otra característica interesante es que la prueba de normalización fuerte utiliza una técnica distinta a la empleada en la clásica prueba de candidatos de reducibilidad. La formalización de esta prueba se presenta en primer lugar para el cálculo lambda simplemente tipado con pares, lo cual puede ser considerado como un aporte secundario. Además, es importante destacar que debido a que este trabajo posee el formato de una tesina de grado, la exposición es detallada, esto resulta un aporte a la comprensión de una prueba con una estructura compleja, como es el caso de la normalización fuerte.

4.1. Trabajo futuro

4.1.1. Inferencia de tipos

La construcción de un término en Sistema I, no solo implica dar su derivación de tipo, sino también construir isomorfismos a través de las reglas de congruencia. Este proceso puede volverse engorroso cuando se trabaja con términos de tamaño considerable.

En la práctica, por ejemplo cuando se programa en Haskell, solo se dan anotaciones de tipo a los términos de nivel superior y luego el *type checker* del lenguaje puede inferir el resto de los tipos.

En el libro *Programming Language Foundations in Agda* [WKS22] se presenta una formalización de inferencia bidireccional de tipos, allí se implementa un algoritmo que a partir de simples anotaciones de tipo, retorna la derivación de tipos completa para un término dado, o un error en caso de que el término esté mal formado. Un trabajo futuro podría ser la definición de un algoritmo similar al presentado en este libro que permita inferir los tipos de los términos definidos en

esta formalización. Debido al paradigma de Proposiciones como Tipos, la formalización de este algoritmo para Sistema I funcionaría también como una prueba de que el tipado de los términos de este cálculo es decidible.

4.1.2. Formalización SIP

Una posible línea de trabajo futuro derivado de la presente tesina sería extender la formalización a Sistema I Polimórfico.

Esto implicaría, en primer lugar, añadir tipos polimórficos al cálculo de base, para ello se podría seguir una estrategia similar a la empleada en este trabajo, es decir, tomar como punto de partida una formalización de Sistema F preexistente, y luego añadir las construcciones particulares de SIP.

En segundo lugar, implicaría extender la prueba de normalización fuerte a SIP, la técnica empleada en este trabajo fue presentada originalmente para Sistema F [Sch19], por lo que se espera que extenderla a SIP sea posible.

Bibliografía

[Aba+91] M. Abadi et al. "Explicit substitutions". En: *Journal of Functional Programming* 1.4 (1991), págs. 375-416. DOI: 10.1017/S0956796800000186.

[AD20] Beniamino Accattoli y Alejandro Díaz-Caro. "Functional Pearl: The Distributive λ-Calculus". En: *CoRR* abs/2002.07944 (2020). arXiv: 2002.07944. URL: https://arxiv.org/abs/2002.07944.

[Bar91] Henk Barendregt. "Introduction to generalized type systems". En: *Journal of Functional Programming* 1.2 (1991), págs. 125-154. DOI: 10.1017/S0956796800020025.

[Bru72] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". En: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), págs. 381-392. DOI: 10.1016/1385-7258(72)90034-0.

[CF58] Haskell B. Curry y Robert M. Feys. *Combinatory Logic Vol. 1*. Amsterdam, Netherlands: North-Holland Publishing Company, 1958.

[DD19] Alejandro Díaz-Caro y Gilles Dowek. "Proof Normalisation in a Logic Identifying Isomorphic Propositions". En: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. por Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 14:1-14:23.

[DD20] Alejandro Díaz-Caro y Gilles Dowek. "Extensional proofs in a propositional logic modulo isomorphisms". En: *CoRR* abs/2002.03762 (2020). arXiv: 2002.03762. URL: https://arxiv.org/abs/2002.03762.

[Di 05] Roberto Di Cosmo. "A short survey of Isomorphisms of Types". En: *Mathematical Structures in Computer Science* 15 (2005). DOI: http://dx.doi.org/10.1017/S0960129505004871.

[DL15] Alejandro Díaz-Caro y Pablo E. Martínez López. "Isomorphisms considered as equalities: Projecting functions and enhancing partial application through and implementation of lambda+". En: *CoRR* abs/1511.09324 (2015). arXiv: 1511.09324. URL: http://arxiv.org/abs/1511.09324.

[GLT89] Jean-Yves Girard, Yves Lafont y Paul Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1989.

[How69] William A. Howard. "The formulae-as-types notion of construction". En: 1969. URL: https://api.semanticscholar.org/CorpusID:118720122.

[Mar75] Per Martin-Löf. "An intuitionistic theory of types: predicative part". En: *Logic Colloquium '73, Proceedings of the Logic Colloquium*. Ed. por H.E. Rose y J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. North-Holland, 1975, págs. 73-118.

[Mar82] Per Martin-Löf. "Constructive mathematics and computer programming". En: *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*. Ed. por L. Jonathan Cohen et al. Vol. 104. Studies in Logic and the Foundations of Mathematics. North-Holland, 1982, págs. 153-175. DOI: 10.1016/S0049-237X(09)70189-2. URL: http://dx.doi.org/10.1016/S0049-237X(09)70189-2.

[Mar84] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984, págs. iv+91. ISBN: 88-7088-105-9.

[Mar98] Per Martin-Löf. "An intuitionistic theory of types". En: *Twenty-five years of constructive type theory (Venice, 1995)*. Ed. por Giovanni Sambin y Jan M. Smith. Vol. 36. Oxford Logic Guides. Oxford University Press, 1998, págs. 127-172.

[McB05] Conor McBride. "Type-Preserving Renaming and Substitution". 2005.

[Rey98] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. DOI: 10.1017/CBO9780511626364.

[Sch19] Steven Schäfer. "Engineering formal systems in constructive type theory". Tesis doct. 2019. DOI: http://dx.doi.org/10.22028/D291-29909.

[SDL21] Cristian F. Sottile, Alejandro Díaz-Caro y Pablo E. Martínez López. "Polymorphic System I". En: *CoRR* abs/2101.03215 (2021). arXiv: 2101.03215. URL: https://arxiv.org/abs/2101.03215.

[Sot20] Cristian Sottile. "Agregando polimorfismo a una lógica que identifica proposiciones isomorfas". Tesis. Universidad Nacional de La Plata, 10 de mar. de 2020. URL: http://sedici.unlp.edu.ar/handle/10915/118544.

[SU06] Morten Heine Sørensen y Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Amsterdam; Oxford: Elsevier, 2006. ISBN: 0444520775 9780444520777. URL: https://www.worldcat.org/title/lectures-on-the-curry-howard-isomorphism/oclc/1171193011&referer=brief_results.

[Wad15] Philip Wadler. "Propositions as types". En: *Communications of the ACM* 58 (nov. de 2015), págs. 75-84. DOI: 10.1145/2699407.

[WKS22] Philip Wadler, Wen Kokke y Jeremy G. Siek. *Programming Language Foundations in Agda*. Ago. de 2022. URL: https://plfa.inf.ed.ac.uk/22.08/.