Universidad Nacional Autónoma
de México

Facultad de Ciencias

Verificación formal del zipper relacional simétrico

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Licenciado en Ciencias de la Computación

PRESENTA:

Fernando Abigail Galicia Mendoza

TUTOR

Dr. Favio Ezequiel Miranda Perea

1.Datos del alumno	1. Datos del alumno
Apellido paterno	Galicia
Apellido materno	Mendoza
Nombre(s)	Fernando Abigail
Teléfono	13 12 62 88
Universidad Nacional Autónoma de México	Universidad Nacional Autónoma de México
Facultad de Ciencias	Facultad de Ciencias
Carrera	Ciencias de la Computación
Número de cuenta	308080777
2. Datos del tutor	2. Datos del tutor
Grado	Dr.
Nombre(s)	Favio Ezequiel
Apellido paterno	Miranda
Apellido materno	Perea
3. Datos del sinodal 1	3. Datos del sinodal 1
Grado	Dra.
Nombre(s)	Lourdes del Carmen
Apellido paterno	González
Apellido materno	Huesca
4. Datos del sinodal 2	4. Datos del sinodal 2
Grado	M. en C.
Nombre(s)	Pilar Selene
Apellido paterno	Linares
Apellido materno	Arévalo
5. Datos del sinodal 3	5. Datos del sinodal 3
Grado	M. en C.
Nombre(s)	Noé Salomón
Apellido paterno	Hernández
Apellido materno	Sánchez
6. Datos del sinodal 4	6. Datos del sinodal 4
Grado	L. en C.C.
Nombre(s)	Cenobio Moisés
Apellido paterno	Vázquez
Apellido materno	Reyes
7.Datos del trabajo escrito.	7. Datos del trabajo escrito
Título	Verificación formal del zipper relacional simétrico
Subtitulo	
Número de páginas	179 p.
Año	2018

A mis padres Ramón Galicia y Leticia Mendoza por el apoyo incondicional que me han brindado, siempre tendré presente las enseñanzas que me han dado.
A el M. en C. Rubén Antonio Molina Hernández (Q.E.D.) por haberme mostrado la belleza de las matemáticas y como de forma indirecta crean las relaciones humanas más hermosas.
Sin ustedes, este trabajo no hubiese sido posible.
Los llevo siempre en mi corazón:
Fernando Galicia

Agradecimiento

El desarrollo del presente trabajo de tesis tuvo lugar en el marco del proyecto:

“Tópicos en computación teórica”
(PAPIME, PE102117)

Otorgado por la Dirección General de Asuntos del Personal Académico de la Universidad Nacional Autónoma de México.

Agradecimientos

En primer lugar quiero agradecer a mis padres Guadalupe Leticia Mendoza Alvarado y Ramón Galicia Ruiz Esparza, por darme las bases para ser el hombre que actualmente soy. Ustedes me enseñaron que debo ser formal, recto y siempre tener las metas claras, pero sin pasar por aspectos esenciales como el amor al prójimo y a uno mismo. Fueron los primeros profesores que tuve y sus enseñanzas siempre las tendré en cuenta. Espero que este trabajo sea de su agrado, es un verdadero honor ser su hijo.

Cuando el Dr. Favio Ezequiel Miranda Perea me aceptó como tesista me indicó que le interesaba la estructura del *zipper*, mientras realizaba dicha investigación me encontré con el artículo de Yuta Ikeda y Susumu Nishimura (2011) [16]. Doctor, le agradezco por haberse aventurado conmigo a verificar formalmente dicho artículo. Sin embargo, también le agradezco aquellas clases donde me contagió el amor hacia el área de la formalidad en computación, una vez siendo su ayudante me enseñó la necesidad de ser claro y finalmente como su tesista me mostró que la claridad de un trabajo es necesaria para evitar el *pecado original*.

Bajo distintas circunstacias pude observar e inclusive trabajar con mis sinodales: la Dra. Lourdes del Carmen González Huesca, la M. en C. Pilar Selene Linares Arévalo, M. en C. Noé Salomón Hernández Sánchez y el L. en C.C. Cenobio Moisés Vázquez Reyes. Quiero agradecerles por brindarme parte de su tiempo para la revisión de este trabajo, es un honor haber contado con su apoyo.

A mi familia que por diversas circunstancias nos hemos disntanciado, pero quiero que sepan que siempre estarán en mi corazón y haré el mejor esfuerzo en beneficio de ustedes. Este trabajo no hubiese sido posible, sin su apoyo constante en distintas circunstancias. Les agradezco de corazón. Menciono de manera especial a mis abuelos (Q.E.D.): Juana Ruiz Esparza, Delfina Alvarado, Ramón Galicia y Miguel Mendoza ya que, los recuerdos que tengo con ustedes son inolvidables y las enseñanzas brindadas yacen en lo profundo de mis pensamientos.

La vida me presentó de una forma curiosa a la M. en C. Montserrat Vite, lo cual siempre agradeceré. Ya que, eres más que mi amiga, has estado conmigo en las peleas más duras que brinda la vida, pero hemos tenido el placer de viajar en varias dimensiones, pelear contra los mejores reyes y navegar en los más bastos océanos. Con gran orgullo y cariño es que digo: ¡Muchas gracias sensei!

Muchas gracias L. en C.C. Antonio Galván por ser un amigo único. Dado que este es un trabajo formal, no puedo hablar con el léxico que usualmente utilizamos, por lo que dichas palabras te per-

mito imaginarlas. Por otra parte, has estado conmigo hombro a hombro desde aspectos académicos hasta unos más personales, todas esas experiencias son muy especiales para mi.

Estefanía Prieto fuiste mi compañera de viaje en los mundos que da la lógica, pero sobre todo has estado conmigo apoyándome en varias situaciones que he vivido. Muchas gracias por esto y por aceptar mis locuras desde el primer día de clases.

A mis compañeros y resto de amistades de la Facultad de Ciencias, tuve el placer de compartir con ustedes distintas experiencias en nuestra querida casa de estudios. Deseo de corazón que cada uno haya cumplido o esté en vías de cumplir sus metas. A mis exalumnos, aseguro que con ustedes logramos cursos interesantes y deseo que disfruten los beneficios académicos y personales que da la facultad. Y a los profesores de la Facultad de Ciencias, que tuve el placer de presenciar sus excelentes cursos, espero que los alumnos disfruten estos últimos, como yo lo hice. A todos ustedes les agradezco de corazón.

Por otra parte, agradezco al grupo Espacios y Sistemas Interactivos para la Educación, dicho grupo establecido en el Insituto de Ciencias Aplicadas y Tecnología, UNAM. Primero por haberme dado la oportunidad de participar como estudiante de Servicio Social y finalmente, poderlos apoyar en distintos proyectos. Agradezco a los académicos el Dr. Fernando Gamboa que tuvo la confianza de invitarme a ciertos proyectos bajo su dirección, la Dra. Clara Alvarado por brindarme parte de su tiempo al explicarme ciertas aspectos de la docencia que resultan esenciales, a la M. en A. M. Libia Eslava por brindarme espacio para corregir ciertos aspectos estéticos que cualquier trabajo merece, al M. en A. Ricardo Castañeda porque usted fue mi compañero en los descansos necesarios mientras realizabamos nuestros respectivos trabajos de titulación. Finalmente, quiero agradecer de manera especial al Dr. Gustavo de la Cruz por haberme apoyado desde que fui su alumno en su curso de Inteligencia Artificial, pasando a ser ayudante de la misma y finalmente haberme dado las herramientas para poder liderar un proyecto. Por esto y más, muchas gracias doctor.

También quiero agradecer a la M. en C. Liliana Reyes por permitirme trabajar en el proyecto *Sistemas de monitoreo y análisis para la toma de decisiones en materia de congestión vehicular y movilidad para la CDMX*. Fue un proyecto retador comenzando por la necesidad de comunicarse con otros profesionistas, adentrarme al área del Análisis de Datos y algo que considero fue lo más complicado: liderar un equipo. En este último aspecto, agradezco a los compañeros que estuvieron apoyándome en distintas fases del proyecto: Karla Jiménez, Rafael Robles, Gilberto García y el L. en C.C. Ricardo Rosas, gracias por el excelente trabajo realizado.

Al M. en C. Rubén Molina (Q.E.D.) te dedico un gran agradecimiento ya que, fuiste un espléndido profesor dentro y fuera del aula. Llevo conmigo recuerdos que nunca desaparecerán y en el lugar que te encuentres, espero que este trabajo refleje una de tus principales enseñanzas: Las matemáticas son la herramienta esencial para trabajar con un equipo de cómputo. Muchas gracias maestro.

Finalmente, a la gente de México quiero decirles que su esfuerzo resulta en una de las bases principales de mi querida casa de estudios, la UNAM. Prometo hacer valer cada una de las enseñanzas adquiridas por esta institución y siempre tratar de brindar lo mejor a México. Muchas gracias.

Resumen

En el proceso de programación, una tarea esencial es realizar cambios locales en una estructura de datos. Por ejemplo, actualizar el valor de un nodo en un árbol. En lenguajes de programación funcional, hacer estos cambios locales en una estructura de datos funcional, puede resultar ineficiente. Ya que, la solución usual consiste en la destrucción total de la estructura para llegar al punto donde se realizará el cambio (al cual llamamos foco), para finalmente reconstruir la estructura de datos en su totalidad [30].

Gérard Huet [12] en respuesta a esta problemática define una estructura de datos, llamada *zipper*, cuyo fin es dividir una estructura de datos en dos elementos: el elemento uno es la subestructura correspondiente a dicho foco y el segundo elemento es la descripción de dicha estructura hasta donde se encuentra el foco en cuestión (a la cual llamamos contexto). Por lo que, utilizando ciertas funciones aplicadas al zipper, *navegaremos* a través de este último. Resultando en un proceso más rápido de modificaciones locales, por la exactitud que brindan estas funciones.

Por otra parte, Yuta Ikeda y Susumu Nishimura [16] mencionan que las funciones anteriores son funciones parciales no inyectivas. Ocasionando que la composición de éstas resulte en errores de cómputo indeseables, esto lo solucionan empleando un lenguaje relacional. Como ejemplo de este enfoque, los autores definen una interpretación relacional del lenguaje xPath[1] para la verificación de equivalencias de expresiones de dicho lenguaje. Esto se realiza verificando que las relaciones binarias, obtenidas de la interpretación de estas expresiones, sean iguales.

El objetivo de este trabajo es mostrar el proceso de verificación formal de las aseveraciones hechas por Yuta Ikeda y Susumu Nishimura en [16], utilizando el asistente de pruebas `COQ` [2]. Para lograr esto, es necesario considerar lo siguiente respecto al artículo de estos autores:

1. Utilizan la siguiente forma de razonamiento para sus demostraciones: *Sean R, S, T y U relaciones binarias y φ una propiedad acerca de la contención $T \subseteq U$, para demostrar que $R \subseteq S$ basta aplicar la propiedad φ a la contención $T \subseteq U$.*

[1] Utilizado en el análisis de archivos `XML` [31].
[2] https://coq.inria.fr/

El argumento anterior lo denotan de la siguiente forma:

$$\begin{array}{c} R \subseteq S \\ \Leftarrow \varphi \\ T \subseteq U \end{array}$$

Esta clase de razonamientos son parecidos a los que se utilizan en COQ, con la notable diferencia de que en el asistente algunas construcciones deben ser inductivas, por lo que la forma de demostrar cambia radicalmente en este sentido. Lo que lleva al siguiente punto.

2. Yuta Ikeda y Susumu Nishimura definen las cerraduras de relaciones binarias utilizando los puntos fijos creados a partir del teorema de Knaster-Tarski, cuya implementación directa en el asistente resultaría complicada e ineficiente. En su lugar, ya que COQ cuenta con mecanismos de demostración para definiciones inductivas y co-inductivas, se utiliza esta característica, lo cual hace más naturales las demostraciones que involucran a estas cerraduras.

3. Finalmente, Yuta Ikeda y Susumu Nishimura definen una cerradura llamada *cerradura simétrica*[1]. Se define nuevamente utilizando el teorema de Knaster-Tarski, pero en el presente trabajo hacemos una traducción de esta definición a un sistema de juicios inductivos. Lo cual facilita la implementación y las demostraciones. Esta idea es una de las aportaciones más importantes de este trabajo.

Es importante aclarar que las demostraciones exhibidas en este trabajo pueden resultar mecánicas, es decir, parece que se repite el mismo razonamiento varias veces. Pero esta forma de razonamiento es adecuada y necesaria para trabajar con el asistente de pruebas, ya que este solamente realiza cálculos simbólicos.

El contenido de este trabajo está dividido de la siguiente forma:

- Preliminares: Se define de manera amplia lo que son las estructuras de datos funcionales y el manejo algebraico de sus constructores. Después se presenta el estudio de un caso mediante el problema de la sustitución en expresiones aritméticas y se da un procedimiento para definir los zipper en estructuras de datos funcionales.

- Capítulo 1: Se da una introducción a la estructura de los documentos XML, se introduce la sintaxis del lenguaje xPath y se explica el significado de las expresiones generadas por este lenguaje, mediante el empleo de una implementación usual de dicho lenguaje y utilizando una representación arbórea de documentos XML.

- Capítulo 2: Se desarrolla el tema de ecuaciones relacionales comenzando con la estructura del álgebra relacional, analizando las relaciones correflexivas y las cerraduras reflexiva-transitiva y simétrica. Se define lo que son las relaciones negativas y se discuten sus propiedades. Se finaliza con definiciones de cerraduras y predicados utilizando juicios co-inductivos y se analiza su comportamiento para determinar su relación con las cerraduras creadas inductivamente.

[1] No confundir con la cerradura simétrica clásica [29].

- Capítulo 3: Se define el zipper relacional y se analizan distintas propiedades utilizando ecuaciones relacionales.
- Capítulo 4: Se le brinda al lenguaje xPath un modelo relacional para analizarlo mediante los zipper relacionales y se verifica la equivalencia de expresiones utilizando la simetría de las relaciones de navegación.
- Capítulo 5: Se ofrecen conclusiones de los temas vistos y se mencionan algunas líneas de investigación que pueden explorarse a futuro.

Esta tesis está pensada para trabajarse en paralelo. Es decir, una vez conocida de manera amplia el uso de `COQ`, el escrito servirá como guía de las pruebas realizadas.

Para consultar la verificación formal de los resultados mostrados en el trabajo, se deben utilizar los archivos creados con `COQ`. Estos archivos están disponibles en el siguiente repositorio:

https://bitbucket.org/FerGaMen/tesis_fgm/src/master/

Índice general

Índice de figuras

Lista de códigos

Capítulo 0

Preliminares

> «*(. . .) El programador funcional suena como un monje medieval, negándose los placeres de la vida con la esperanza de hacerse virtuoso. (. . .)* » (p. 2)
>
> Hughes, John (1989) [13]

Cuando se le presenta un problema a un programador, muy probablemente optará por el paradigma imperativo para su solución. Esto se debe principalmente a dos factores: históricamente los lenguajes de programación funcionales son más lentos que los imperativos [1] y existe una considerable cantidad de literatura donde se pueden consultar estructuras de datos, funciones y/o métodos eficientes, etc. para lenguajes imperativos [30].

Sin embargo, entre las ventajas que tienen los lenguajes de programación funcional [13] destacan:

- El seguimiento de cómputo usualmente es más sencillo, por la noción de función matemática.
- El código puede resultar corto y elegante.

Suponiendo que se opta por el paradigma funcional, uno de los primeros pasos que debe hacer el programador es elegir una (o varias) estructuras de datos para lograr la abstracción del problema. En este capítulo analizaremos las estructuras de datos funcionales desde una perspectiva algebraica, analizamos el problema de sustitución en expresiones aritméticas, utilizando la estructura de datos llamada *zipper* y finalizamos con la descripción de un procedimiento para construir el *zipper* de cualquier estructura de datos funcional.

[1] Esto es parcialmente cierto, debido a los grandes avances que se han hecho en el desarrollo de compiladores.

0.1. Estructuras de datos funcionales

Chris Okasaki en [30] brinda una breve explicación de lo complicado que resulta diseñar una estructura de datos funcional comparada con una imperativa, ambas para resolver un mismo problema.

Todo comienza desde que las estructuras de datos se dividen en dos categorías: *persistente* y *efímera*.

Definición 0.1 (Persistente)**.** *Una estructura de datos es llamada persistente si soporta múltiples versiones, es decir,* ***persiste*** *la última versión previa a una actualización.*

Definición 0.2 (Efímera)**.** *Una estructura de datos es llamada efímera si permite una sola versión posterior a la actualización.*

Toda estructura de datos funcional es persistente, esto puede resultar en un arma de doble filo: tenemos acceso a las versiones anteriores pero resulta ineficiente mantenerlas. Veamos el caso particular de las listas, las cuales son una estructura de datos persistente bajo una implementación funcional. Sin embargo, en una implementación imperativa esta propiedad no se cumple.

Ejemplo 0.1. *La especificación de la función* `tail` *es: dada una lista no vacía ℓ, devuelve la lista resultante de eliminar el primer elemento. La versión funcional de su implementación, por ejemplo en lenguaje de programación* `HASKELL`*, es:*

Código 0.1 Versión funcional de `tail`.

```
tail :: [a] -> [a]
tail [] = error "Lista vacía."
tail (x:xs) = xs
```

Por ejemplo si $\ell = [1, 2, 3]$, al ejecutar la instrucción `tail` ℓ *se obtiene el valor $\ell_1 = [2, 3]$. Y obsérvese que la lista ℓ no ha sido modificada, es decir, ℓ y ℓ_1 coexisten en la memoria.*

Por otra parte, la implementación de `tail` *en pseudocódigo imperativo es:*

```
def tail_i return list(a) (l:list(a)){
  if(length(l) == 0){
    return error;
  }else{
```

```
        l <- l->head();
        return l;
      }
    }
```

Código 0.2: Versión imperativa de `tail`.

En este caso, al ejecutar la instrucción `tail_i` ℓ *el valor de* ℓ *se modifica a* `[2,3]` *siendo imposible recuperar el valor original* `[1,2,3]`.

A simple vista las estructuras de datos funcionales resultan inconvenientes con respecto a las imperativas, pero es precisamente esta característica de persistencia la causante de que dichas estructuras se puedan representar mediante una expresión matemática lo cual como se verá más adelante proporciona ventajas en un análisis formal.

0.2. Constructores de tipos

Construir una estructura de datos no es una tarea sencilla, existen procedimientos que permiten construir una estructura de datos mediante una especificación dada. Por ejemplo, Daniela Lauro en [20] estudia un procedimiento que permite construir una estructura de datos funcional a partir de una especificación particular utilizando una traducción puramente sintáctica.

Dicho procedimiento se realiza mediante un sistema de ecuaciones polinomiales generadas por un conjunto de constructores[1], los cuales son operadores que reciben tipos (estructura de datos) y las transforman en un nuevo tipo. Estas operaciones se pueden clasificar en tres categorías [27]:

1. Tipos producto. Son tuplas de valores donde cada componente es de un tipo específico.
2. Tipos suma. Son valores de alguno de n tipos específicos con una etiqueta explícita indicando cual de estos n tipos se trata.
3. Tipos recursivos. Son tipos auto-referenciables en el sentido de que sus valores pueden definirse mediante otros valores del mismo tipo.

Formalmente los operadores son expresiones generadas por la siguiente gramática:

$$F \ ::= \ Empty \mid Void \mid Id \mid (F \mid F) \mid (F \times F) \mid (F \circ F)$$

Estos operadores son aplicados a un tipo específico y su interpretación es la siguiente:

[1] Daniela Lauro en [20] y Connor McBride en [25], llaman a estas transformaciones *funtores aplicativos* ya que, tienen una acción funtorial llamada *fmap*. Sin embargo, no es necesario este concepto para los fines de este trabajo.

- *Empty*: Constructor que transforma un tipo `a` al tipo vacío, es decir, $Empty\ \texttt{a} = Empty$.
- $Void$: Constructor que transforma un tipo `a` en el tipo $Void$, este último contiene un solo elemento. Es decir, $Void\ \texttt{a} = e$ con e elemento de tipo $Void$.
- Id: Constructor que deja intacto al tipo dado como parámetro, es decir, dado un tipo `a` se tiene que $Id\ \texttt{a} = \texttt{a}$.
- $F_1 \mid F_2$: Constructor que dado un tipo `a` devuelve el tipo suma de los constructores F_1 y F_2 aplicados al tipo `a`, es decir, $(F_1 \mid F_2)\ \texttt{a} = F_1\ \texttt{a} \mid F_2\ \texttt{a}$.
- $F_1 \times F_2$: Constructor que dado un tipo `a` devuelve el tipo producto de los constructores F_1 y F_2 aplicados al tipo `a`, es decir, $(F_1 \times F_2)\ \texttt{a} = F_1\ \texttt{a} \times F_2\ \texttt{a}$.
- $F_1 \circ F_2$: Constructor que dado un tipo `a` devuelve el tipo composición de los constructores F_1 y F_2 aplicados al tipo `a`, es decir, $(F_1 \circ F_2)\ \texttt{a} = F_1(F_2\ \texttt{a})$.

Obsérvese que los operadores están recibiendo un tipo `a` pero este tipo es una variable que representa un tipo particular. Por ejemplo: Si $\texttt{a} = Void \mid Nat$, entonces el tipo $Bool \times \texttt{a}$ es el tipo cuyos valores son expresiones de la forma (b, e) con b un valor Booleano y e valor de tipo $Void$ ó Nat, este último es el tipo que representa los números naturales.

A continuación algunos ejemplos de estos constructores y sus implementación en el lenguaje `HASKELL`.

- Constructor suma. El tipo `Nat | Bool` representa a la unión ajena de números naturales y valores booleanos.

Código 0.3 Ejemplo del funtor suma.

```
data NB = Left Nat | Right Bool
```

- Constructor producto. El constructor `Double × Double` permite representar pares de puntos de $\mathbb{R} \times \mathbb{R}$.

Código 0.4 Ejemplo del funtor producto.

```
type R2 = (Double,Double)
```

- Tipo composición. Una forma general de representar matrices $M_{n\times m}$ de números reales, con $n, m \in \mathbb{N}$, es utilizando las listas que contengan listas que almacenan números reales. El tipo `List(List Double)` cumple esta especificación.

Código 0.5 Ejemplo del funtor composición.

```
-- [a] es azúcar sintáctica de List a.

type Matrices = [[Double]]
```

El interés que tenemos del procedimiento dado por Daniela Lauro(2015) en [20], es el último paso para generar una estructura de datos:

> «(...) 4. A partir del sistema de ecuaciones T_D se obtiene un tipo de dato en HASKELL, introduciendo nombres adecuados de constructores en dicha ecuación y siguiendo algunas convenciones propias del lenguaje.(...) » (p. 42)

Es decir, a partir de una representación en ecuaciones con constructores es posible obtener un tipo de datos para el lenguaje HASKELL.

Se adoptarán las siguientes convenciones:

- El tipo suma será representado por el símbolo usual, es decir, $F_1 + F_2$.
- Se utilizará la notación de funciones, es decir, en lugar de escribir F a se escribirá $F(\mathtt{a})$.

Ejemplo 0.2. *Considerando la ecuación con tipo arbitrario* a*:*

$$F(\mathtt{a}) = Void + \mathtt{a} \times F(\mathtt{a})$$

Se obtiene la estructura de datos lista.

Código 0.6 Estructura de datos: List.

```
data List a = Nil | Cons a (List a)
```

Ejemplo 0.3. *Considerando la ecuación con tipo arbitrario* a*:*

$$F(\mathtt{a}) = Void + \mathtt{a} \times F(\mathtt{a}) \times F(\mathtt{a})$$

Se obtiene la estructura de datos árbol binario.

Código 0.7 Estructura de datos: AB.

```
data AB a = Empty | Node a (AB a) (AB a)
```

Obsérvese que el procedimiento puede ser invertido, es decir, dada una estructura de datos se puede obtener su representación en un sistema de ecuaciones con constructores [25].

Ejemplo 0.4. *Considerando el siguiente tipo de datos:*

```
data AB a = Empty | Node a (AB a) (AB a)
```

Se obtiene la siguiente ecuación:

$$F(\mathtt{a}) = Void + \mathtt{a} \times F(\mathtt{a}) \times F(\mathtt{a})$$

Para nuestros propósitos es más simple utilizar una ecuación no recursiva con un parámetro extra y hacer referencia al punto fijo de dicha ecuación, como se ve en los siguientes ejemplos:

Ejemplo 0.5. *La definición de tipos de datos lista se traduce a la siguiente ecuación:*

$$F(\mathtt{a}) = Void + \mathtt{a} \times F(\mathtt{a})$$

Renombremos a $F(a)$ por la variable R, para evitar la recursión:

$$F(\mathtt{a}, R) = Void + \mathtt{a} \times R$$

De esta manera el tipo de datos lista denotado L es el punto fijo de dicha ecuación, es decir, se cumple que $L = F(a, L)$.

Ejemplo 0.6. *La definición de tipos de datos árboles binarios se traduce a la siguiente ecuación:*

$$F(\mathtt{a}) = Void + \mathtt{a} \times F(\mathtt{a}) \times F(\mathtt{a})$$

Renombremos a $F(a)$ por la variable R, para evitar la recursión:

$$F(\mathtt{a}, R) = Void + \mathtt{a} \times R \times R$$

De esta manera el tipo de datos lista denotado AB es el punto fijo de dicha ecuación, es decir, se cumple que $AB = F(a, AB)$.

Con este procedimiento es posible definir de manera algebraica las estructuras de datos funcionales, de tal forma que se pueda manipular y atacar un problema derivado de la persistencia: la ineficiencia de tener que destruir y reconstruir la estructura de datos.

En la siguiente sección presentamos un análisis sobre la sustitución en expresiones aritméticas, haciendo más eficiente la solución intuitiva al problema anterior mediante un análisis sintáctico.

0.3. Estudio de un caso: Sustitución en expresiones aritméticas

Los editores de texto estructurados son utilizados para la creación y/o modificación de diagramas, fórmulas, programas de computadora y otros tipos de documentos. Su característica principal es proporcionar una interfaz agradable para el usuario con la posibilidad de buscar, sustituir, deshacer y rehacer código.

A continuación, se discute la solución al problema de sustituir texto en el caso particular de un lenguaje de expresiones aritméticas, dicha solución es implementada en el lenguaje de programación `HASKELL`.

0.3.1. Problema

Se considera el siguiente lenguaje de expresiones aritméticas:

$$e \ ::= \ x \mid n \mid e + e \mid e * e \mid suc(e)$$

con $x \in Var$ y $n \in \mathbb{N}$, donde Var denota al conjunto (infinito) de nombres de variables.

Imaginemos que un usuario creó el programa `1+suc(2+x)` y después de trabajar un rato en el programa quiere modificarlo, por ejemplo, sustituyendo la subexpresión `2+x` por la expresión `suc(12*x)`, obteniendo la expresión `1+suc(suc(12*x))`. En ese momento el usuario selecciona la expresión y el problema se resuelve mediante las acciones de cortar y pegar a cargo del editor, formalmente este es un proceso de sustitución en expresiones discutido a continuación.

0.3.2. Primera solución

Suponiendo que el programa ya está transformado en su sintaxis abstracta, la solución al problema lo realiza el Código 0.8.

Se observan dos inconvenientes de esta solución:

1. Llamadas recursivas inútiles.
2. Necesidad de recorrer toda la estructura de datos para realizar varios cambios en esta.

Una forma de atacar estos inconvenientes sería considerar las direcciones exactas de las subexpresiones a sustituir, las cuales corresponden a rutas en el árbol de sintaxis abstracta.

```haskell
-- | EA. Tipo de dato que representa una expresión aritmética.
data EA = V String | N Int | Suma EA EA | Prod EA EA | Suc EA


-- | Sust. Tipo que representa una sustitución.
type Sust = (EA,EA)


-- | sustituye. Función que dada una expresión e y una sustitución (e1,e2)
-- si e1 figura en e, entonces devuelve la sustitución de e1 por e2 en e.
sustituye :: EA -> Sust -> EA
sustituye (V x) (V y,e) = if x == y then e else (V x)
sustituye (N n) (N m,e) = if n == m then e else (N n)
sustituye (Suc e) (Suc t,et) =
```

```
  if e == t then et else Suc (sustituye e (Suc t,et))
sustituye (Suc e) (t,et) = Suc (sustituye e (t,et))
sustituye (Suma e1 e2) (Suma t1 t2,et) =
  if (e1 == t1) && (e2 == t2) then et
  else Suma (sustituye e1 (Suma t1 t2,et)) (sustituye e2 (Suma t1 t2,et))
sustituye (Suma e1 e2) (e,et) = Suma (sustituye e1 (e,et))
                                     (sustituye e2 (e,et))
sustituye (Prod e1 e2) (Prod t1 t2,et) =
  if (e1 == t1) && (e2 == t2) then et
  else Prod (sustituye e1 (Prod t1 t2,et)) (sustituye e2 (Prod t1 t2,et))
sustituye (Prod e1 e2) (e,et) = Prod (sustituye e1 (e,et))
                                     (sustituye e2 (e,et))
sustituye x s = x
```

Código 0.8: Sustitución en expresiones aritméticas.

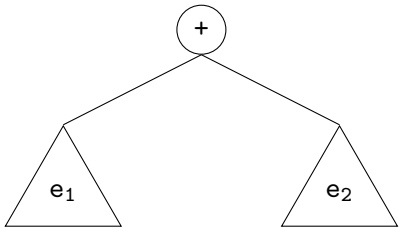
0.3.3. Direcciones

En el momento en que el usuario solicite sustituir una subexpresión e_1 por una expresión e_2, la acción que está realizando el editor es cortar a e_1 y pegar a e_2 en su lugar. Es decir, desde un punto de vista abstracto la acción que realiza el editor es *enfocarse* en un subárbol del árbol de sintaxis abstracta y sustituir dicho árbol por el árbol de sintaxis abstracta que representa la nueva expresión.

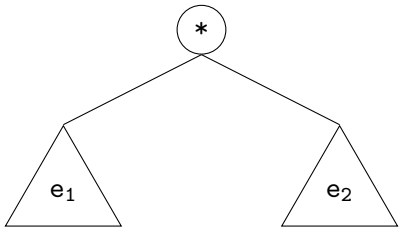
En los árboles de sintaxis abstracta de la Figura 1 se observa que el usuario únicamente puede enfocarse en:

- Un subárbol izquierdo o derecho si es la expresión suma.
- Un subárbol izquierdo o derecho si es la expresión producto.
- Un subárbol si es la expresión sucesor.
- Una hoja que contenga una variable.
- Una hoja que contenga un número.

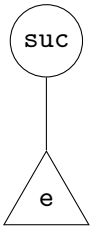
Esto indica que existen cinco formas de destruir la estructura de las expresiones aritméticas. Visualmente esto se traduce a que hay tres posibles direcciones, las cuales podemos tomar en el árbol de sintaxis abstracta: izquierda, derecha o abajo.



(a) Estructura de la suma.



(b) Estructura del producto.



(c) Estructura del sucesor.

(d) Hoja que contiene una variable.

(e) Hoja que contiene un número.

Figura 1: Árboles de sintaxis abstracta para las expresiones aritméticas.

El tipo de datos que representa a estas direcciones es:

Código 0.9 Tipo de dato: `Direccion`.

```haskell
-- | Direccion. Tipo de dato que representa una dirección.

data Direccion = Izquierda | Derecha | Abajo
```

En la expresión `1+suc(2+x)` el usuario se está enfocando en la subexpresión `2+x`, cuya raíz es el operador suma, la siguiente figura muestra la subexpresión enfocada, lo cual llamamos *foco*.

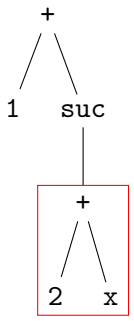


Figura 2: Subexpresión a analizar.

Para poder analizar la subexpresión `2+x`, se supondrá que un analizador sintáctico devolverá una lista con las direcciones para llegar al foco, en la Figura 2 la lista de direcciones es `[Derecha,Abajo]`.

Estando en el punto de enfoque, el proceso se reduce a una sustitución textual cuya implementación se muestra en el Código 0.10.

```haskell
-- | Camino. Tipo que representa un conjunto de direcciones.
type Camino = [Direccion]

-- | sustCamino. Función que dada una expresión e, una expresión e1
-- que reemplazara una subexpresión de e y un camino c,
-- hace el reemplazo de e1 en e de acuerdo al camino c.
sustCamino :: EA -> EA -> Camino -> EA
sustCamino _ e [] = e
sustCamino (Suc e) s (Abajo:ds) = Suc (sustCamino e s ds)
sustCamino (Suma e1 e2) s (Izquierda:ds) = Suma (sustCamino e1 s ds) e2
sustCamino (Suma e1 e2) s (Derecha:ds) = Suma e1 (sustCamino e2 s ds)
sustCamino (Prod e1 e2) s (Izquierda:ds) = Prod (sustCamino e1 s ds) e2
sustCamino (Prod e1 e2) s (Derecha:ds) = Prod e1 (sustCamino e2 s ds)
sustCamino _ _ _ = error "Dirección incorrecta."
```

Código 0.10: Sustitución utilizando caminos.

Para sustituir la subexpresión `2+x` por la expresión `suc(12+x)` se logra ejecutando la siguiente instrucción:

Código 0.11 Ejemplo de `sustCamino`.

```haskell
ejemSustCam = sustCamino (1+suc(2+x)) (12+x) [Derecha,Abajo]
```

Por lo que las direcciones solucionan el problema de tener que destruir toda la expresión, sin embargo, si el usuario dio una serie de instrucciones de manera *local*, como se ve en la Figura 4, se tendría que repetir el proceso de sustitución tantas veces como el número de instrucciones brindadas.

0.3.4. Un beneficio parcial de las direcciones

El usuario le solicita al editor una sustitución sobre una expresión e utilizando un camino c_1 resultando en una nueva expresión e_1 y acto seguido le solicita una sustitución sobre e_1 utilizando un

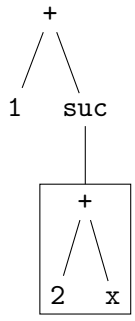
camino c_2, realizar estas sustituciones se traduce a ejecutar dos veces el Código 0.10. Por ejemplo:

Código 0.12 Intersección de caminos.

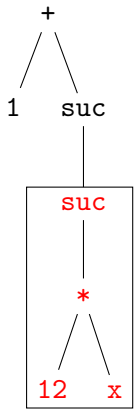
```
-- Primera sustitución

inst1 = sustCamino (1+suc(2+x)) (suc(12*x)) [Derecha,Abajo]


-- Segunda sustitución

inst2 = sustCamino inst1 (3+y) [Derecha,Abajo,Abajo]
```

Se observa que la intersección de ambos caminos es la lista `[Derecha,Abajo]`, es decir, todas las sustituciones se están realizando de manera local en el nivel que se encuentra originalmente la subexpresión `2+x`. Los árboles de sintaxis abstracta mostrados en las figuras 3 y 4 ejemplifican la ejecución simbólica de esta sustitución local.

Recordemos que requerimos una estructura capaz de representar estos cambios locales de manera precisa, pero sin perder la información almacenada en la estructura original. Dicha estructura se construye en la siguiente sección.



(a) Llegar el punto de enfoque.



(b) Primera sustitución.

Figura 3: Ejecución simbólica. Los símbolos en rojo corresponden a la primera sustitución.

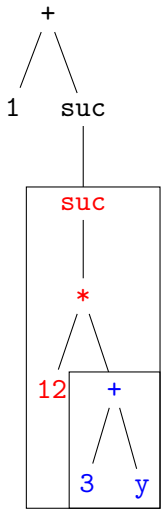


Figura 4: Ejecución simbólica. Los símbolos en azul a la segunda sustitución.

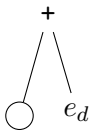
0.3.5. Migajas

Resolver el inconveniente anterior consiste en que cada vez que el cursor se mueva a una nueva subexpresión debe existir un medio de almacenamiento que encapsule al operador y la subexpresión que no serán modificados, a la cual llamaremos *migaja* [23].

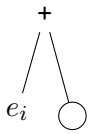
Por ejemplo en la Figura 3, cuando el editor va a realizar la segunda sustitución deberá almacenar el operador de producto y el número 12, sustituir la variable x por la expresión $3+y$ y devolver la expresión resultante; en este caso la migaja consiste en el operador de producto y el número 12.

Para el caso de la suma al haber dos posibles subexpresiones, la migaja almacena el operador y la subexpresión que no está siendo analizada (análogamente para el producto); para el sucesor al haber una única subexpresión, la migaja solamente almacena el operador. Esto se observa en los árboles mostrados en la Figura 5. Formalmente la estructura de la migaja está constituida de la siguiente forma:

1. El operador principal que es la raíz del árbol.
2. La subexpresión almacenada. (Solo para el caso de la suma y producto.)
3. El hueco, que es la subexpresión que será analizada.

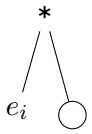


(a) Migaja suma izquierda.



(b) Migaja suma derecha.

(c) Migaja producto izquierdo.

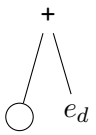


(d) Migaja producto derecho.

(e) Migaja sucesor.

Figura 5: Migajas de los operadores.

A esta estructura se le nombra migaja ya que que es un *rastro* de las subexpresiones que no serán utilizadas, por ejemplo, en la Figura 6 la migaja *suma izquierda* (inciso (a)) indica que el rastro almacenado es la expresión derecha e_d de la suma $e_i + e_d$, dicho rastro es causado por un movimiento del cursor hacia la expresión izquierda e_i (inciso (b)).



(a) Rastro almacenado.

e_i

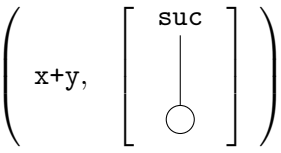
(b) Localización actual del cursor.

Figura 6: Migaja *suma izquierda* y su expresión que está siendo analizada.

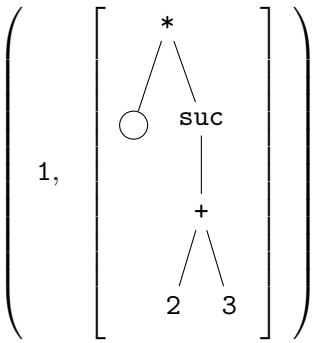
A continuación se presentan unos ejemplos del recorrido de un cursor sobre expresiones aritméticas y las migajas resultantes de este recorrido:

Ejemplo 0.7. *Dada la expresión* `suc(x+y)` *se muestra la migaja de sucesor indicando el camino recorrido del cursor hasta la subexpresión* `x+y`*. Es decir, se extrae el contenido del sucesor el cual queda como foco de análisis y el rastro dejado por este movimiento: La migaja que almacena el*

operador `suc`*, es almacenada en una lista.*



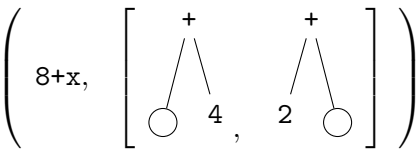
Ejemplo 0.8. *Dada la expresión* `1*suc(2+3)` *se muestra la migaja de producto izquierdo indicando el foco de análisis. Es decir, se extrae el contenido izquierdo del producto como foco de análisis y el rastro dejado por este movimiento: La migaja que almacena el operador* `*` *y la subexpresión* `suc(2+3)`*, es almacenada en una lista.*



Ejemplo 0.9. *Dada la expresión* `2+((8+x)+4)` *se muestran las migajas de la suma, izquierda y derecha, indicando el camino recorrido del cursor hasta la subexpresión* `8+x`*.*
Es decir, se hicieron los siguientes movimientos:

1. *Se extrae el contenido de la suma derecha (*`(8+x)+4`*) como foco de análisis y el rastro dejado por este movimiento: La migaja que almacena el operador* `+` *y la subexpresión* `2`*, es almacenada en una lista* ℓ*.*

2. *Se extrae el contenido de la suma izquierda (*`8+x`*) como foco de análisis y el rastro dejado por este movimiento: La migaja que almacena el operador* `+` *y el valor* `4`*, es almacenada al inicio de la lista* ℓ*.*



Para lograr este tipo de almacenamiento, las migajas de las expresiones aritméticas deben ser implementadas de la siguiente forma:

Código 0.13 Tipo de dato: `Migaja`.

```
-- | Migaja. Tipo que representa una migaja de las expresiones aritméticas.
data Migaja  = SumaIzq EA | SumaDer EA | ProdIzq EA | ProdDer EA | Succ
```

Donde `SumaIzq` y `SumaDer` representan las migajas izquierda y derecha de la suma, respectivamente; análogamente para el caso del producto, mientras que `Succ` representa la migaja del operador sucesor. Recalcamos que Gérard Huet en [12] no incluye el hueco en la definición de migaja, por lo que se decidió seguir esta convención.

Los ejemplos mostrados son la idea de la estructura de datos zipper que encapsulan expresiones aritméticas, en la siguiente sección se explica de forma detallada la razón porque el zipper está definida de esta forma.

0.3.6. El zipper y sus funciones

Dado que el movimiento del cursor es parecido a una búsqueda en profundidad, la estructura de datos que almacena las migajas de acuerdo al orden del movimiento del cursor es una lista, esta lista es llamada *contexto* y se implementa de la siguiente forma:

Código 0.14 Tipo de dato: `Contexto`.

```
-- | Contexto. Tipo que representa los rastros que va dejando el cursor.
type Contexto = [Migaja]
```

Observemos que la principal característica de los ejemplos 0.7, 0.8 y 0.9 es un par cuya primera entrada es el foco de análisis, mientras que la segunda es el contexto.

Código 0.15 Tipo de dato: `Zipper`.

```
-- | Zipper. Tipo que representa el foco de análisis (una expresión aritmética)
-- y el contexto de una expresión aritmética.
type Zipper = (EA,Contexto)
```

Por lo que los ejemplos mostrados al final de la sección anterior, quedan implementados con las siguientes expresiones en `HASKELL`:

```
-- Ejemplo 0.7
ej1_7 = (Suma (V "x") (V "y"),[Succ])


-- Ejemplo 0.8
```

```haskell
ej1_8 = (N 1,[ProdIzq (Suc (Suma (N 2) (N 3)))])


-- Ejemplo 0.9
ej1_9 = (Suma (N 8) (V "x"),[SumaIzq (N 4),SumaDer (N 2)])
```

Código 0.16: Ejemplos de zipper.

Para generar expresiones de esta forma realizamos el siguiente procedimiento:

1. Se encapsula la expresión aritmética inicial. Es decir, dada una expresión aritmética `e` crear el par `(e,[])`, donde `[]` es el contexto vacío.
2. El cursor se mueve hasta el foco de análisis.
3. Se realizan los cambios locales, los cuales son movimientos cercanos al foco de análisis.
4. Se reconstruye la expresión hasta que el contexto sea vacío.

Por lo que la ejecución simbólica presentada en la Figura 3 se formaliza, utilizando esta nueva estructura, como se muestra a continuación:

Ejemplo 0.10. *Dada la expresión* `1+suc(2+x)` *se sustituirá la subexpresión* `2+x` *por la expresión* `suc(12*x)`*, una vez hecho esto se debe sustituir la variable* `x` *por la expresión* `3+y`*. Este es el procedimiento:*

1. *Encapsulamiento de la expresión aritmética inicial.*

   ```
   (1+suc(2+x), [])
   ```

2. *El cursor llega al primer foco de análisis.*

   ```
   (2+x, [Succ,SumaDer 1])
   ```

3. *Se hacen los cambios locales.*

   a) *Primer cambio local, es decir, la primera sustitución.*

   ```
   (suc(12*x), [Succ,SumaDer 1])
   ```

   b) *El cursor llega al segundo foco.*

   ```
   (x,[ProdDer 12,Succ,Succ,SumaDer 1])
   ```

c) *Segundo cambio local, es decir, la segunda sustitución.*

```
(3+y,[ProdDer 12,Succ,Succ,SumaDer 1])
```

4. *Reconstrucción de la expresión.*

```
(1+suc(suc(12*(3+y))),[])
```

obsérvesese que el ejemplo anterior es un caso ideal ya que, la segunda sustitución fue realizada en un nivel inferior al foco de análisis. Sin embargo, si las sustituciones deben ser realizadas lejanas al foco de análisis esto se vuelve ineficiente.

Las operaciones anteriores son composiciones de las llamadas *funciones de navegación* [12]:

```haskell
-- | toZip. Función que encapsula una expresión aritmética en un zipper.
toZip :: EA -> Zipper
toZip e = (e,[])

-- | sumaIzq. Función que dada una suma enfocada mueve el foco
-- a la subexpresión izquierda.
sumaIzq :: Zipper -> Zipper
sumaIzq (Suma e1 e2,c) = (e1,SumIzq e2:c)
sumaIzq _ = error "Movimiento inválido."

-- | sumaDer. Función que dada una suma enfocada mueve el foco
-- a la subexpresión derecha.
sumaDer :: Zipper -> Zipper
sumaDer (Suma e1 e2,c) = (e2,SumDer e1:c)
sumaDer _ = error "Movimiento inválido."

-- | prodIzq. Función que dado un producto enfocado mueve el foco
-- a la subexpresión izquierda.
prodIzq :: Zipper -> Zipper
prodIzq (Prod e1 e2,c) = (e1,ProdIzq e2:c)
prodIzq _ = error "Movimiento inválido."
```

```haskell
-- | prodDer. Función que dado un producto enfocado mueve el foco
-- a la subexpresión derecha.
prodDer :: Zipper -> Zipper
prodDer (Prod e1 e2,c) = (e2,ProdDer e1:c)
prodDer _ = error "Movimiento inválido."

-- | succ. Función que dado un sucesor enfocado mueve el foco a la
-- subexpresión del operador.
succ :: Zipper -> Zipper
succ (Sucesor e,c) = (e,Succ:c)
succ _ = error "Movimiento inválido."

-- | migajaSumaIzq. Función que dado un contexto cuya cabeza sea
-- el rastro de una suma izquierda, construye la suma con la
-- expresión enfocada.
migajaSumaIzq :: Zipper -> Zipper
migajaSumaIzq (e,SumaIzq e':c) = (Suma e e',c)
migajaSumaIzq _ = error "Movimiento inválido."

-- | migajaSumaDer. Función que dado un contexto cuya cabeza sea
-- el rastro de una suma derecha, construye la suma con la
-- expresión enfocada.
migajaSumaDer :: Zipper -> Zipper
migajaSumaDer (e,SumaDer e':c) = (Suma e' e,c)
migajaSumaDer _ = error "Movimiento inválido."

-- | migajaProdIzq. Función que dado un contexto cuya cabeza sea
-- el rastro de un producto izquierdo, construye el producto con la
-- expresión enfocada.
```

```haskell
migajaProdIzq :: Zipper -> Zipper
migajaProdIzq (e,ProdIzq e':c) = (Prod e e',c)
migajaProdIzq _ = error "Movimiento inválido."

-- | migajaProdDer. Función que dado un contexto cuya cabeza sea
-- el rastro de un producto derecho, construye el producto con la
-- expresión enfocada.
migajaProdDer :: Zipper -> Zipper
migajaProdDer (e,ProdDer e':c) = (Prod e' e,c)
migajaProdDer _ = error "Movimiento inválido."

-- | migajaAbajo. Función que dado un contexto cuya cabeza sea
-- el rastro de un sucesor, construye el sucesor con la
-- expresión enfocada.
migajaAbajo :: Zipper -> Zipper
migajaAbajo (e,Succ:c) = (Suc e,c)
migajaAbajo _ = error "Movimiento inválido."

-- | reconstruye. Función que dado un contexto y una expresión
-- enfocada, devuelve la construcción del rastro dejado.
reconstruye :: Zipper -> EA
reconstruye (e,[]) = e
reconstruye (e,SumaIzq e':c) = reconstruye (Suma e e',c)
reconstruye (e,SumaDer e':c) = reconstruye (Suma e' e,c)
reconstruye (e,ProdIzq e':c) = reconstruye (Prod e e',c)
reconstruye (e,ProdDer e':c) = reconstruye (Prod e' e,c)
reconstruye (e,Succ e':c) = reconstruye (Suc e',c)
```

Código 0.17: Sustitución de expresiones aritméticas con el zipper.

Entonces el Ejemplo 0.10 muestra que primero hay que realizar un análisis sobre los árboles de sintaxis abstracta de la estructura de datos a estudiar, después se definen las migajas, los contextos y finalmente el zipper. Una vez construido el zipper, solamente falta definir las funciones de navegación de acuerdo a la destrucción y reconstrucción de la estructura de datos.

El proceso de construir el zipper resulta tedioso por el análisis sobre los árboles de sintaxis abstracta. En la siguiente sección se describe un procedimiento puramente algebraico para construir el zipper de cualquier estructura de datos recursiva.

0.4. Procedimiento para construir el zipper

En la sección anterior se hizo el análisis de la estructura de datos que representa a las expresiones aritméticas para generar su zipper y se definieron las funciones de navegación sobre el mismo, esto resulta en un procedimiento largo en caso de crear el zipper de otra estructura de datos funcional recursiva.

Por otra parte, en la sección 0.2 se da la traducción de una estructura de datos a un sistema de ecuaciones con constructores. Estas ecuaciones fueron usadas por Conor McBride en [25] para generar el zipper de cualquier estructura, mediante el siguiente procedimiento:

1. Se obtiene el sistema de ecuaciones con constructores de la estructura `E`.
2. Se transforman las ecuaciones a una donde se incluya el punto fijo de recursión.
3. Se deriva la ecuación respecto al punto fijo de recursión.
4. Se traduce la ecuación anterior a un tipo de datos, este nuevo tipo de datos define a las migajas.
5. Se define el contexto del zipper (`C`), es decir, una lista que almacene las migajas del paso 4.
6. Se genera el par `(E,C)`, resultando en el zipper que encapsula a la estructura de datos del paso 1.

Dado que las ecuaciones con constructores tienen forma de polinomio, la operación de derivar a la que nos referimos en el paso 3, es análoga a la definida en el análisis matemático. Es decir, dado un polinomio $p(x) = a_0 + a_1x + \cdots + a_nx^n$, con a_i constante tal que $1 \leq i \leq n$, su derivada se define de la siguiente forma:

$$\frac{\partial p(x)}{x} = \sum_{i=1}^{n} i a_i x^{i-1}$$

Ejemplo 0.11. *La estructura que representa las expresiones aritméticas está definida de la siguiente forma:*

```
data EA = V String | N Int | Suma EA EA | Prod EA EA | Suc EA
```

La traducción a un sistema de ecuaciones es la siguiente:

$$\mathcal{EA}(R) = C_1 + C_2 + R \times R + R \times R + R \tag{1}$$

Se calcula la derivada respecto a la variable R:

$$\frac{\partial \mathcal{EA}(R)}{R} = 2R + 2R + 1 \tag{2}$$

$$= R + R + R + R + 1 \tag{3}$$

Se observa que la derivada genera cuatro expresiones de llamada recursiva y una constante, las primeras cuatro resultan ser la suma izquierda, suma derecha, producto izquierdo, producto derecho y el operador de sucesor. Entonces la implementación a `HASKELL` *es:*

```
data Migaja  = SumaIzq EA | SumaDer EA | ProdIzq EA | ProdDer EA | Succ
```

Como ya se dijo, el contexto es una lista que almacena las migajas obtenidas:

```
type Contexto = [Migaja]
```

Finalmente, se define el zipper como:

```
type Zipper = (EA,Contexto)
```

A diferencia de otras estructuras de datos recursivas como listas o árboles binarios, las expresiones aritméticas son una estructura de datos que no reciben un tipo como parámetro. Sin embargo, los árboles de sintaxis abstracta de cada expresión aritmética son árboles binarios cuyos nodos almacenan los operadores aritméticos y las hojas almacenan variables o números. Esto permite generalizar el zipper de expresiones aritméticas a árboles binarios. A continuación se aplica el procedimiento previo, para obtener el zipper que encapsula árboles binarios.

Ejemplo 0.12. *La estructura de árboles binarios está definida de la siguiente forma:*

```
data AB a = Empty | Node (AB a) a (AB a)
```

La traducción a un sistema de ecuaciones es la siguiente:

$$\mathcal{AB}(x) = C_1 + \mathcal{AB}(x) \times x \times \mathcal{AB}(x) \tag{4}$$

Fijando la variable x como una constante y considerando a $\mathcal{AB}(x)$ como el punto fijo R se tiene que:

$$\mathcal{AB}(R) = C_1 + R \times x \times R \tag{5}$$

$$= C_1 + xR^2 \tag{6}$$

Dado que x es una constante respecto a la variable recursiva R, la derivada del polinomio respecto a la variable R es:

$$\frac{\partial \mathcal{AB}(R)}{R} = 2xR \tag{7}$$

$$= x \times R + x \times R \tag{8}$$

El polinomio resultante de la derivada define a las migajas, obsérvese que el tipo almacenado por los árboles binarios se debe pasar como parámetro a las migajas:

```
data Migaja a = L a (AB a) | R a (AB a)
```

De manera análoga a las migajas de las expresiones aritméticas, las migajas de árboles binarios son los rastros almacenados al momento de destruir la estructura. Es decir, la migaja `L x r` *es el rastro de haber destruido un árbol binario por la izquierda y tener almacenados la raíz* `x` *y el subárbol derecho* `r`*, de manera similar es el caso para la migaja* `R x l`*.*
El contexto es la lista que almacena las migajas:

```
type Contexto a = [Migaja a]
```

Finalmente, se define el zipper:

```
type Zipper a = (AB a,Contexto a)
```

Teniendo el procedimiento para generar el zipper de una estructura de datos funcional, es necesario definir las funciones de destrucción y reconstrucción de la estructura que encapsula el zipper. Gérard Huet en [12] estableció que tales funciones se les llame *funciones de navegación* ya que, desde una perspectiva puramente visual, las funciones están indicando el movimiento del foco a través del árbol de sintaxis abstracta.

Las funciones de navegación no son complicadas de obtener, únicamente hay que brindar una función que realice movimientos hacia arriba o abajo tales que correspondan a las ramas del árbol de sintaxis abstracta [16]. Por ejemplo las funciones de navegación para árboles binarios mueven el foco de análisis entre los subárboles izquierdos y derechos del árbol inicial encapsulado en el zipper.

Código 0.18 Funciones de destrucción sobre árboles binarios utilizando su zipper.

```
-- | dnL. Función que destruye el árbol binario
-- con movimiento hacia la izquierda y almacena el subárbol
-- restante en el contexto.
dnL :: Zipper a -> Zipper a
dnL (Node l i r,c) = (l,L i r:c)
dnL _ = error "Movimiento no válido."

-- | dnR. Función que destruye el árbol binario
-- con movimiento hacia la derecha y almacena el subárbol
-- restante en el contexto.
dnR :: Zipper a -> Zipper a
dnR (Node l i r,c) = (r,R i l:c)
dnR _ = error "Movimiento no válido."
```

Código 0.19 Funciones de construcción sobre árboles binarios utilizando su zipper.

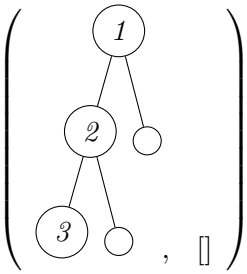
```
-- | upR. Función que reconstruye el árbol binario
-- con contexto almacenando el subárbol derecho.
upR :: Zipper a -> Zipper a
upR (t,L i r:c) = (Node t i r,c)
upR _ = error "Movimiento no válido."

-- | upL. Función que reconstruye el árbol binario
-- con contexto almacenando el subárbol izquierdo.
upL :: Zipper a -> Zipper a
upL (t,R i l:c) = (Node l i t,c)
upL _ = error "Movimiento no válido."
```

Si estas funciones de navegación se componen de manera incorrecta o bien al momento de realizar cambios sobre el foco, el contexto no cumple las características necesarias para aplicarlas, entonces el programa terminará con un mensaje de error.

0.4.1. Navegación incorrecta

En esta sección se describen posibles escenarios donde la aplicación de las funciones de navegación causen que el programa termine en error. Por ejemplo si un árbol binario, almacenado en un zipper, no cuenta con los subárboles necesarios para la aplicación de las funciones de navegación que destruyen al árbol, entonces la aplicación de estas funciones termina en error.

Ejemplo 0.13. *Considerando el siguiente zipper* **`z`** *que contiene un árbol binario que almacena números naturales:*



Al ejecutar la instrucción **`dnL (dnR z)`** *devolverá error. En otras palabras la función* `dnL` ∘ `dnR` *no está definida para el zipper* **`z`**.

El ejemplo anterior establece que las funciones de navegación para árboles binarios son parciales, ya que no están definidas para todos los posibles zipper que almacenen árboles binarios. De este ejemplo se sigue que las funciones navegación `dn_L`,`dn_R`, `up_L`, `up_R` y las composiciones `dn_L` ∘ `up_L` y `dn_R` ∘ `up_R`, son funciones parciales.

Por otra parte, si se considera la función que reconstruye un árbol binario dado un contexto cualquiera:

Código 0.20 Función que navega un nivel arriba en el zipper de árboles binarios.

```
go_up :: Zipper a -> Zipper a

go_up (l,L i r:c) = (Node l i r,c)

go_up (r,R i l:c) = (Node l i r,c)

go_up _ = error "Movimiento inválido."
```

La función anterior no es inyectiva, impidiendo definir una función que revierta dicha acción.

Estos resultados indican que las funciones de navegación pueden causar la terminación abrupta de un programa y funciones como `go_up`, al no tener inversa, impiden una navegación libre en el árbol. Se podrían analizar estas funciones trabajando con el error[1], al precio de complicar ampliamente los razonamientos. En respuesta a este inconveniente, Yuta Ikeda y Susumu Nishimura en [16] proponen estudiar a las funciones de navegación como relaciones binarias, llamadas *relaciones*

[1] Es decir, utilizando los mecanismos de manejo de excepciones de cada lenguaje de programación.

de navegación, permitiendo un análisis algebraico más dócil.

Dado que los documentos `XML` tienen una representación en árboles binarios, se pueden utilizar las *relaciones de navegación* para la interpretación de las expresiones generadas por el lenguaje *xPath*, dicho lenguaje se especifica en el siguiente capítulo.

Capítulo 1

El lenguaje xPath

> «*XML deriva de la filosofía que los datos pertenecen a sus creadores y que los proveedores del contenido se benefician mejor con un formato de datos que no los vincule a lenguajes interpretados(...)*» (p. 129)
>
> Bosak, Jon (1997) [35]

Uno de los principales intereses de este trabajo no es analizar un lenguaje de expresiones aritméticas, más bien es utilizar la estructura del zipper en una estructura más general, la cual permita el almacenamiento jerárquico de datos.

Actualmente, existen distintos lenguajes que permiten el almacenamiento de información. Por ejemplo:

1. `XML`: Metalenguaje que estructura los datos de manera jerárquica e inmutable.
2. `JSON`: Expresiones que estructuran los datos en objetos almacenados en un arreglo.
3. `CSV`: Formato que almacena los datos en forma de tablas.

Este tipo de lenguajes son utilizados para el intercambio de información entre distintos sistemas, en el caso de los documentos `XML` tienen la estructura jerárquica solicitada y están definidos en dos partes [4]:

- Parte Física. El documento está compuesto de unidades llamadas entidades (etiquetas con atributos específicos).
- Parte Lógica. El documento está compuesto de declaraciones, elementos, referencias de caracteres y procesos de instrucción, cada uno indicado por una etiqueta explícita.

Uno de los intereses primordiales que se tiene respecto a los documentos `XML` es poder obtener la mayor cantidad de información que cumplan ciertas propiedades, las expresiones generadas por el lenguaje *xPath* son instrucciones que permiten obtener esta información. En las siguientes secciones se construye el lenguaje a través de ejemplos, con apoyo de una representación arbórea de los documentos `XML`.

1.1. Representación arbórea de archivos `XML`

El Código 1.1 es una factura electrónica apócrifa[1] basada en indicaciones del SAT (Sistema de Administración Tributaria), dependencia del gobierno de los Estados Unidos Mexicanos.[2] Dicha factura es emitida por *Don Gato* al receptor *Benito Bodoque* por una serie de productos que le vendió. Este tipo de documentos son parte de ciertos análisis de datos.

```
<Comprobante>
    <Emisor>
        <Nombre nombre="Don Gato"/>
        <Domicilio direccion="Bote de basura 1"/>
        <ExpedidoEn ciudad="CDMX"/>
    </Emisor>
    <Receptor>
        <Nombre nombre="Benito Bodoque"/>
        <Domicilio direccion="Bote de basura 2"/>
    </Receptor>
    <Conceptos>
        <Concepto valor=13>Lata de atún</Concepto>
        <Parte/>
        <Concepto valor=200>Cachiporra de matute</Concepto>
        <Parte/>
    </Conceptos>
    <Impuestos>
```

[1] Carece de varias etiquetas establecidas por la *W3C* (*World Wide Web Consortium*) [4]. Enlace oficial: https://www.w3.org/.

[2] Consultado en http://www.sat.gob.mx/informacion_fiscal/factura_electronica/Documents/cfdi/ejemplocfdv3.xml el 6 de Noviembre del 2017.

```
            <Traslados>
                <Traslado iva=.16/>
                <Traslado ietu=.32/>
            </Traslados>
        </Impuestos>
    </Comprobante>
```

Código 1.1: Ejemplo documento XML.

Supongamos que *Don Gato* requiere un programa que obtenga todos datos de *Benito Bodoque* y los productos que éste adquirió. A parte, *Don Gato* quiere que este procedimiento se haga para cualquier factura emitida por él, de manera que no son necesarios los valores almacenados en las etiquetas y solamente se debe conocer la estructura base de cualquier factura extendida por él. Dicha estructura base se llama esqueleto y la mostramos en el Código 1.2.

```
<Comprobante>
    <Emisor>
        <Nombre/>
        <Domicilio/>
        <ExpedidoEn/>
    </Emisor>
    <Receptor>
        <Nombre/>
        <Domicilio/>
    </Receptor>
    <Conceptos>
        <Concepto></Concepto>
        <Parte/>
        <Concepto></Concepto>
        <Parte/>
    </Conceptos>
    <Impuestos>
```

```
            <Traslados>
                <Traslado/>
                <Traslado/>
            </Traslados>
        </Impuestos>
    </Comprobante>
```

Código 1.2: Esqueleto de un documento `XML`.

Por la naturaleza jerárquica de los documentos `XML`, un árbol n-ario es la mejor opción de representación. Se consideran las siguientes abreviaciones:

- `Comprobante` como C.
- `Conceptos` como Cs.
- `Concepto` como Ct.
- `Domicilio` como D.
- `Emisor` como E.
- `ExpedidoEn` como Ex.
- `Impuestos` como I.
- `Nombre` como N.
- `Parte` como Pt.
- `Receptor` como R.
- `Traslados` como T.
- `Traslado` como Tr.

El árbol que representa el Código 1.2 es la siguiente:

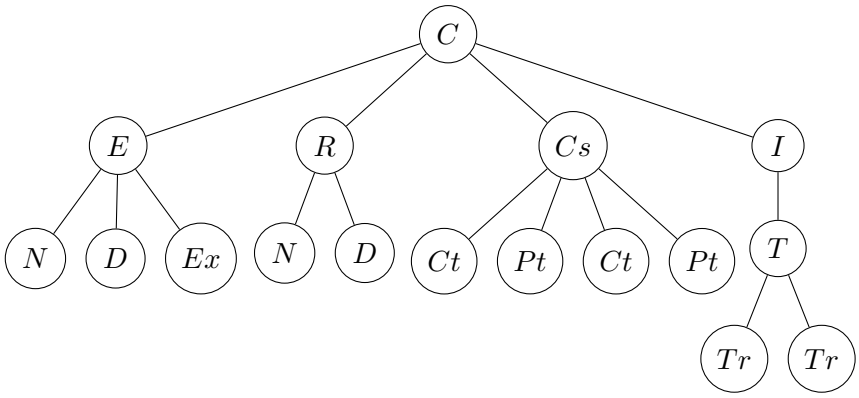


Figura 1.1: Árbol representante del Código 1.2.

Con esto, la solución al problema de *Don Gato* consiste en: un programa que devuelva listas de nodos. Por ejemplo, la lista que contiene todos los datos de *Benito Bodoque* y los productos que adquirió es la siguiente:

$$[R, N, D, Cs, Ct, Ct, Pt, Pt]$$

Sin embargo, este conjunto puede causar conflictos, ya que los nodos N y D tienen como padre el nodo R pero también el nodo E. La solución nos la da la naturaleza de las etiquetas, ya que una etiqueta está conformada por los siguientes elementos básicos:

1. La etiqueta de apertura, es decir etiquetas de la forma `<id>`.
2. Contenido de la etiqueta, o bien propiedades del objeto que representa.
3. La etiqueta que cierra, es decir etiquetas de la forma `</id>`.

Esta descripción establece que la información es acotada por las etiquetas `<id>` y `</id>`. De manera que para obtener la información solicitada por *Don Gato*, bastaría extraer la información almacenada por las etiquetas *Receptor* y *Conceptos*.

En otras palabras, se deberán extraer los siguientes fragmentos del Código 1.2:

```
<!-- Información del Receptor -->
<Receptor>
    <Nombre/>
    <Domicilio/>
</Receptor>

<!-- Información de los Conceptos -->
```

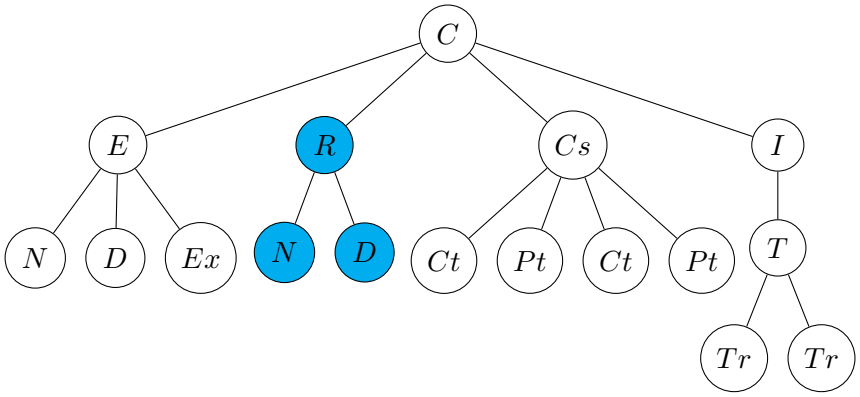
```
<Conceptos>
    <Concepto></Concepto>
    <Parte/>
    <Concepto></Concepto>
    <Parte/>
</Conceptos>
```

Código 1.3: Fragmentos del Código 1.2

Esta restricción sobre las etiquetas es una propiedad de seguridad esencial, de otra forma se tendría la capacidad de eliminar y/o modificar información de los documentos `XML`, causando la corrupción de datos.

Para obtener la información respecto a una etiqueta, es necesario tener una estructura que encapsule la información representada por un conjunto de nodos de la representación arbórea. El *eje del nodo* [1] define un conjunto de nodos relativos al nodo actual (llamado *nodo contexto*), es decir, si se considera un nodo n de una representación arbórea, el eje del nodo n consiste en todos los nodos que están conectados a n de acuerdo a una propiedad específica.

Ejemplo 1.1. *Por ejemplo los nodos conectados al nodo contexto R consiste en los siguientes nodos*[2]*:*



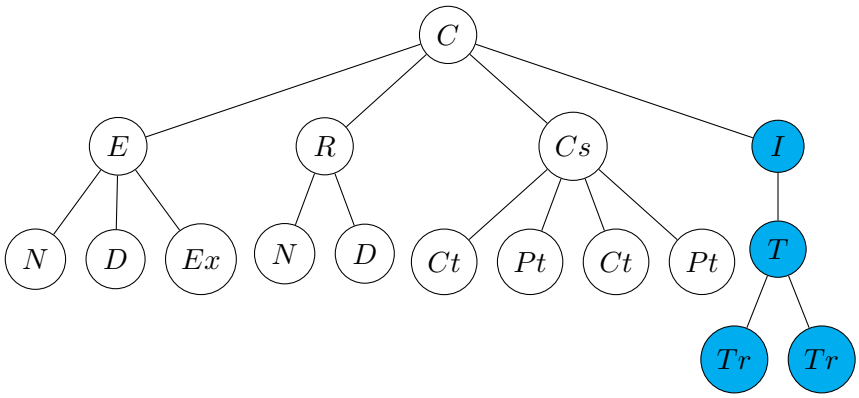
Este conjunto de nodos es formalmente llamado *eje del nodo self* del nodo contexto R. Dichos ejes son detallados en la siguiente, mientras veamos otro ejemplo.

[1] En inglés *node axis*.

[2] En el resto del Capítulo, toda la información que sea obtenida será marcada con color azul en los árboles representantes del documento.

A *Don Gato* no le basta saber qué productos le vendió a *Benito Bodoque*, también quisiera saber que cantidad de dinero deberá cobrarle para el respectivo pago de impuestos. Dicha información está almacenada en la etiqueta *Impuestos* del Código 1.2. Dado que el nodo I se encuentra al mismo nivel que el nodo Cs, la información solicitada por *Don Gato* pertenece al eje de los hermanos derechos del nodo Cs.

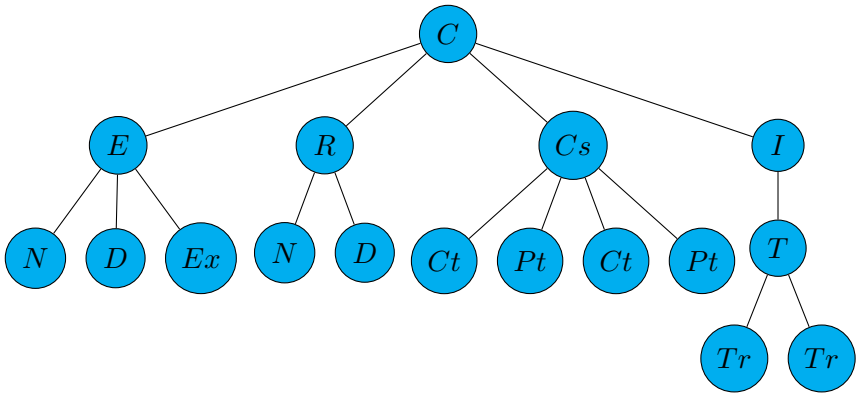
Ejemplo 1.2. *Eje de los hermanos derechos del nodo contexto Cs:*



Obsérvese que al solicitar el eje de los hermanos derechos del nodo Cs no se considera únicamente el nodo I, también se consideran los nodos que se encuentran debajo de este último. La razón es que la etiqueta *Impuestos* tiene como contenido las etiquetas *Traslados* y *Traslado*. Es decir, cuando se solicite el eje P de un nodo n, hay que devolver los nodos pertenecientes al eje P.

En el ejemplo anterior se solicita información que se encuentra debajo de un nodo contexto n, sin embargo, los ejes del nodo también capturan información que se encuentra en niveles superiores a la posición del nodo n. Por ejemplo, si se solicita el eje ancestros del nodo T, entonces se devuelve la información que almacenan los nodos superiores a T, es decir, todo el Código 1.2.

Ejemplo 1.3. *Eje de los ancestros del nodo contexto T:*



Utilizando la estructura de los ejes del nodo, la solución de *Don Gato* es un programa que devuelva la información correspondiente al conjunto de los ejes de los nodos de acuerdo a una

propiedad solicitada. Para realizar esta tarea es necesario definir un lenguaje que represente las especificaciones (propiedades) que da *Don Gato*, este lenguaje se llama *xPath* y se introduce en la siguiente sección.

1.2. Uso del lenguaje xPath

La *W3C* definió el lenguaje `XPATH` (*XML Path Language*) para extraer información de documentos `XML`, este lenguaje permite dar rutas exactas sobre la estructura jerárquica de un árbol que representa a estos documentos [4].

En este trabajo se utiliza una abstracción teórica del lenguaje `XPATH`, nombrado xPath. Éste último es propuesto por Dan Olteanu *et. al* en [31] y no contiene ciertos predicados (por ejemplo, relaciones de orden sobre números), funciones (posición de la etiqueta) u otros elementos con los que cuenta `XPATH` ya que, estos no aportan información extra en las interpretaciones teóricas que se darán más adelante.

El lenguaje xPath se genera mediante la siguiente gramática:

$$
\begin{array}{lll}
e & ::= & \backslash p \mid p \mid e \cup e \mid e \cap e \\
p & ::= & a :: * \mid a :: \beta \mid p[q] \mid p/p \\
q & ::= & q \text{ and } q \mid q \text{ or } q \mid \text{not } q \mid p \\
a & ::= & \text{self} \mid \text{child} \mid \text{foll-sibling} \mid \text{desc} \mid \text{desc-or-self} \mid \text{following} \mid \text{parent} \mid \\
 & & \text{prec-sibling} \mid \text{anc} \mid \text{anc-or-self} \mid \text{preceding}
\end{array}
$$

Cada regla de producción genera las siguientes categorías sintácticas:

- e. Expresiones xPath.
- p. Caminos.
- q. Predicados.
- a. Ejes de los nodos.

Actualmente existen distintas implementaciones para la interpretación del lenguaje `XPATH`, por ejemplo:

- Herramientas online como: https://www.freeformatter.com/xpath-tester.html.
- Programas en el sistema operativo `Linux` como `xmllint`, enlace oficial: http://xmlsoft.org/xmllint.html.
- Módulo en el lenguaje de programación `HASKELL`, enlace oficial: https://hackage.haskell.org/package/hxt-xpath.

A continuación se describe cada regla, desde la base (ejes de los nodos) hasta las expresiones xPath y se muestran ejemplos de la interpretación de estas expresiones. Estos ejemplos incluyen: la especificación en español, el resultado de la interpretación en una representación arbórea, la traducción de la especificación a una expresión `XPATH` y el código resultante de la interpretación hecha por el programa `xmllint`.

1.2.1. Eje de los nodos

Considerando un árbol que representa un documento `XML` y un nodo contexto n_i perteneciente a este árbol, se definen los ejes del nodo n_i [32]:

- self: Eje que contiene las etiquetas en el alcance de la etiqueta representada por el nodo n_i.

Ejemplo 1.4. *Considerando el árbol de la Figura 1.1, el eje del propio nodo, eje self, Emisor (E) está compuesto como se muestra a continuación:*

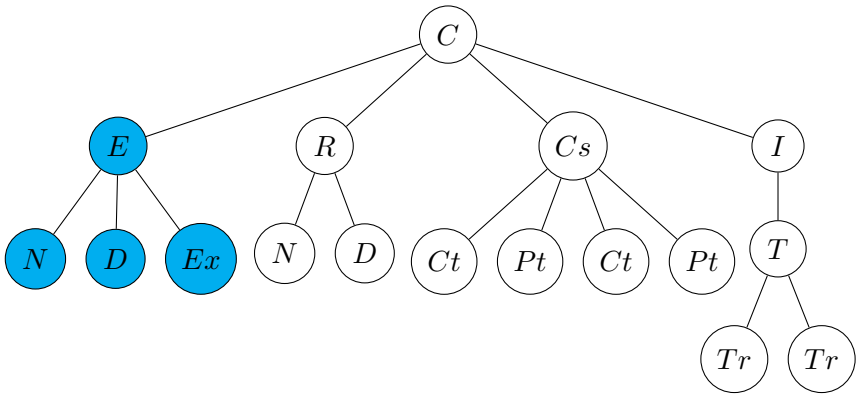


Figura 1.2: Ejemplo gráfico de self.

Código 1.4 Expresión ejemplo de self.

```
//Emisor/self::*
```

Los textos `\\` *y* `*`*, indican que el análisis no comienza desde la raíz del árbol*[1] *y que tomará cualquier etiqueta dentro de la etiqueta* `Emisor`*.*

Código 1.5 Eje del nodo self.

```
<Emisor>
    <Nombre/>
    <Domicilio/>
    <ExpedidoEn/>
</Emisor>
```

[1] Algo parecido a la cadena `\~` de sistemas `Linux`.

- child: Eje que contiene todas las etiquetas hijas de la etiqueta representada por n_i, es decir, los hijos de n_i.

  Ejemplo 1.5. *Considerando el árbol de la Figura 1.1, el eje de los nodos hijos de Impuestos (I) está compuesto como se muestra a continuación:*

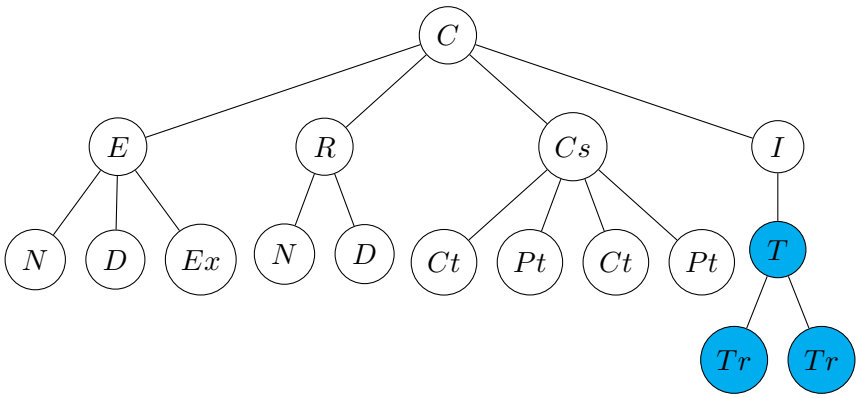


Figura 1.3: Ejemplo gráfico de child.

Código 1.6 Expresión ejemplo de child.

```
//Impuestos/child::*
```

Código 1.7 Eje del nodo child.

```
<Traslados>
    <Traslado/>
    <Traslado/>
</Traslados>
```

- desc: Eje que contiene todas las etiquetas hijas y etiquetas hijas de las hijas de la etiqueta representada por n_i. Es decir, devuelve los nodos hijos de n_i, los hijos de los hijos y así sucesivamente hasta llegar a las hojas.

  Ejemplo 1.6. *Considerando el árbol de la Figura 1.1, el eje de los nodos descendientes de Impuestos (I) está compuesto como se muestra a continuación:*

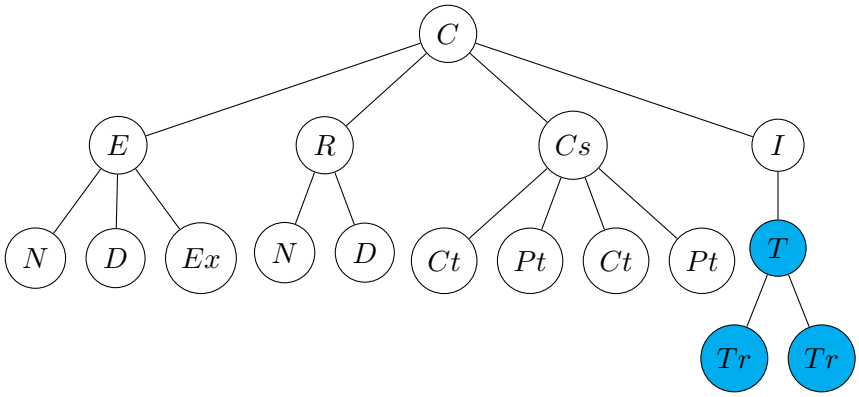


Figura 1.4: Ejemplo gráfico de descendant.

Código 1.8 Expresión ejemplo de descendant.

```
//Impuestos/descendant::*
```

Código 1.9 Eje del nodo descendant.

```
<Traslados>
    <Traslado/>
    <Traslado/>
</Traslados>

<Traslado/>

<Traslado/>
```

Obsérvese que la diferencia principal entre child y desc, es que el segundo devuelve los hijos de los hijos. Es decir, no regresa solamente las etiquetas hijas, también devuelve por separado el contenido de cada etiqueta contenida en cada etiqueta hija.

- desc-or-self: Eje que contiene las etiquetas descendant más las etiquetas self del nodo n_i.

Ejemplo 1.7. *Considerando el árbol de la Figura 1.1, el eje de los nodos descendientes o el mismo nodo Impuestos (I) está compuesto como se muestra a continuación:*

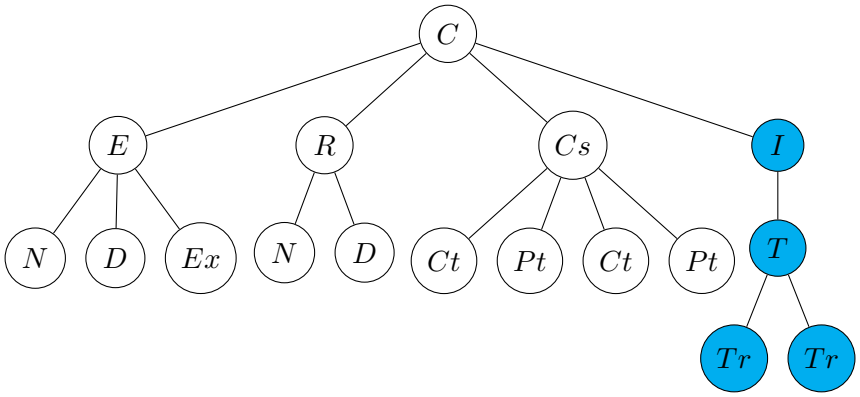


Figura 1.5: Ejemplo gráfico de descendant or self.

Código 1.10 Expresión ejemplo de descendant or self.

```
//Impuestos/descendant-or-self::*
```

Código 1.11 Eje del nodo descendant or self.

```
<Impuestos>
    <Traslados>
        <Traslado/>
        <Traslado/>
    </Traslados>
</Impuestos>

<Traslados>
    <Traslado/>
    <Traslado/>
</Traslados>

<Traslado/>

<Traslado/>
```

- following: Eje que contiene todos los nodos después del cierre de la etiqueta correspondiente al nodo n_i.

Ejemplo 1.8. *Considerando el árbol en la Figura 1.1, el eje de los nodos siguientes a Conceptos (Cs) está compuesto como se muestra a continuación:*

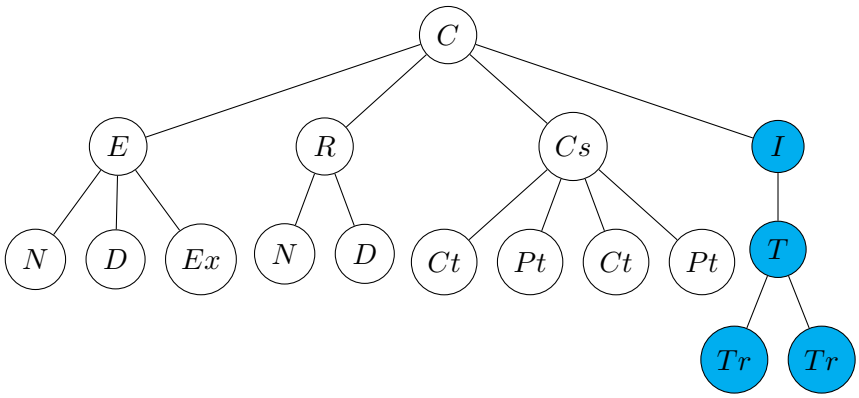


Figura 1.6: Ejemplo gráfico de following.

Código 1.12 Expresión ejemplo de following.

```
//Conceptos/following::*
```

Código 1.13 Eje del nodo following.

```
<Impuestos>
    <Traslados>
        <Traslado/>
        <Traslado/>
    </Traslados>
</Impuestos>
```

- foll-sibling: Eje que contiene las etiquetas de todos los nodos hermanos derechos del nodo n_i.

Ejemplo 1.9. *Considerando el árbol de la Figura 1.1, el eje de los nodos hermanos derechos de Receptor (R) está compuesto como se muestra a continuación:*

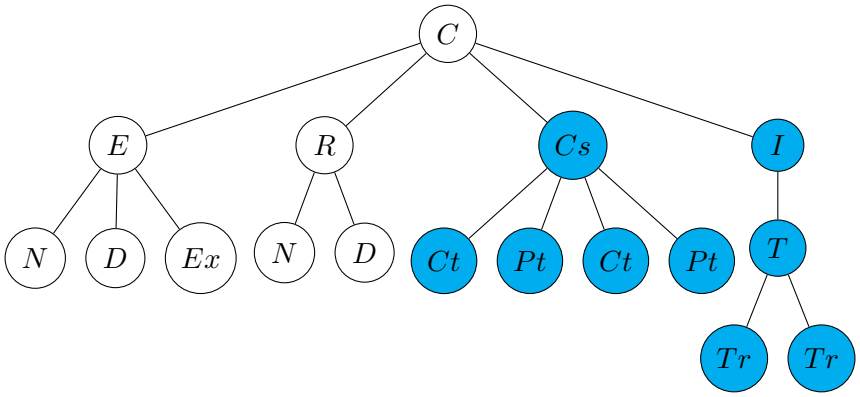


Figura 1.7: Ejemplo gráfico de following-sibling.

Código 1.14 Expresión ejemplo de following-sibling.

```
//Receptor/following-sibling::*
```

Código 1.15 Eje del nodo following-sibling.

```
<Conceptos>
    <Concepto></Concepto>
    <Parte/>
    <Concepto></Concepto>
    <Parte/>
</Conceptos>

<Impuestos>
    <Traslados>
        <Traslado/>
        <Traslado/>
    </Traslados>
</Impuestos>
```

- prec-sibling: Eje que contiene las etiquetas de todos los nodos hermanos izquierdos del nodo n_i.

  Ejemplo 1.10. *Considerando el árbol de la Figura 1.1, el eje de los nodos hermanos izquierdos de Receptor (R) está compuesto como se muestra a continuación:*

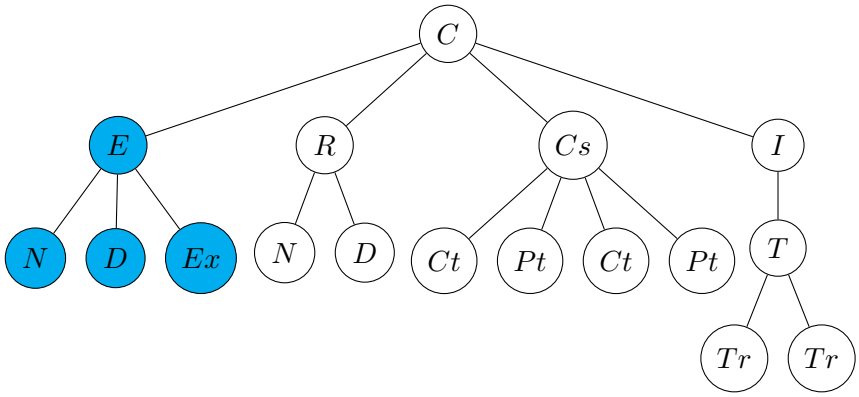


Figura 1.8: Ejemplo gráfico de preceding-sibling.

Código 1.16 Expresión ejemplo de preceding-sibling.

```
//Receptor/preceding-sibling::*
```

Código 1.17 Eje del nodo preceding-sibling.

```
<Emisor>
    <Nombre/>
    <Domicilio/>
    <ExpedidoEn/>
</Emisor>
```

- parent: Eje que contiene las etiquetas que encapsulan al nodo padre n_i o bien devuelve vacío en caso de no tener nodo padre.

  Ejemplo 1.11. *Considerando el árbol de la Figura 1.1, el eje de los nodos padres de Traslado (Tr) está compuesto como se muestra a continuación:*

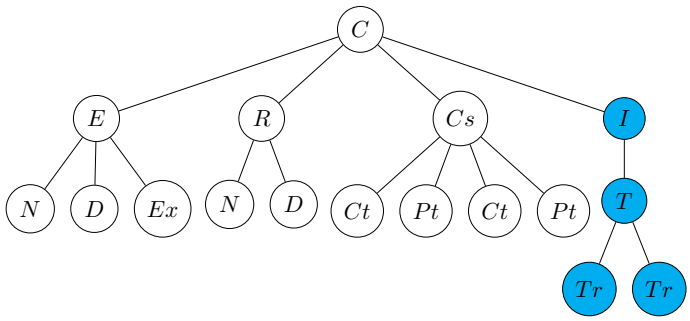


Figura 1.9: Ejemplo gráfico de parent.

Código 1.18 Expresión ejemplo de parent.

```
//Traslado/parent::*
```

Código 1.19 Eje del nodo parent.

```
<Impuestos>
    <Traslados>
        <Traslado/>
        <Traslado/>
    </Traslados>
</Impuestos>
```

- anc: Eje que contiene todas las etiquetas de padres del nodo n_i, los padres de los padres y así sucesivamente hasta llegar a la raíz del árbol. Esto es tomado de forma recursiva.

  Ejemplo 1.12. *Considerando el árbol de la Figura 1.1, el eje de los nodos ancestros de Traslado (Tr) está compuesto como se muestra a continuación:*

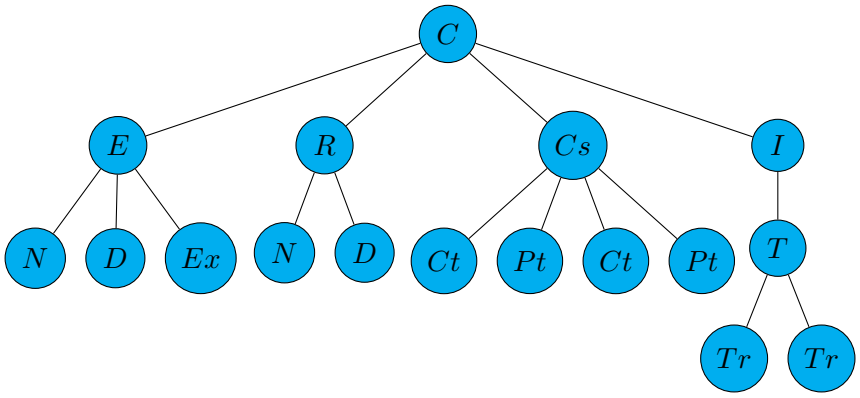


Figura 1.10: Ejemplo gráfico de ancestor.

Código 1.20 Expresión ejemplo de ancestor.

```
//Traslado/ancestor::*
```

Código 1.21 Eje del nodo ancestor.

```
<Comprobante>
<!-- Todo el documento -->
</Comprobante>

<Impuestos>
    <Traslados>
        <Traslado/>
        <Traslado/>
    </Traslados>
</Impuestos>

<Traslados>
    <Traslado/>
    <Traslado/>
</Traslados>
```

- anc-or-self: Eje que contiene las etiquetas ancestros más las etiquetas dentro del nodo n_i.

  Ejemplo 1.13. *Considerando el árbol de la Figura 1.1, el eje de los nodos ancestros o él mismo de Traslado (T) está compuesto como se muestra a continuación:*

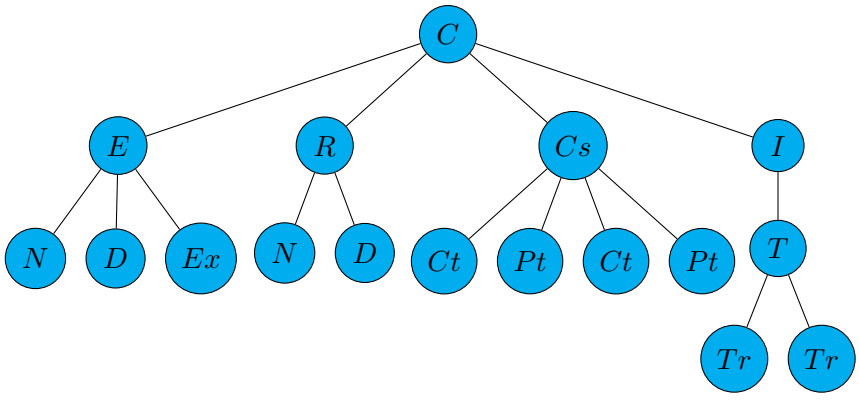


Figura 1.11: Ejemplo gráfico de ancestor-or-self.

Código 1.22 Expresión ejemplo de ancestor-or-self.

```
//Traslado/ancestor-or-self::*
```

```
<Comprobante>
<!-- Todo el documento -->
</Comprobante>

<Impuestos>
    <Traslados>
        <Traslado/>
        <Traslado/>
    </Traslados>
</Impuestos>

<Traslados>
    <Traslado/>
    <Traslado/>
</Traslados>

<Traslado/>
<Traslado/>
```

Código 1.23: Eje del nodo ancestor-or-self.

Obsérvese que los árboles mostrados en las figuras 1.10 y 1.11 son el mismo, a pesar de corresponder a distintos ejes del nodo. Esto sucede ya que la representación arbórea no tiene el nivel de expresividad que el código. El Código 1.21 contiene la información correspondiente a todo el documento y las etiquetas *Impuestos* y *Traslados*, mientras que el Código 1.23 a parte de contener la información del código anterior, contiene la información almacenada en la etiqueta *Traslados*, es decir, las etiquetas *Traslado* también son mostradas por separado.

- preceding: Eje que contiene todos los nodos inmediatos anteriores a que abra el nodo n_i, excepto sus antecesores. Obsérvese que un nodo padre no es ancestro.

Ejemplo 1.14. *Considerando el árbol de la Figura 1.1, el eje de los nodos precedentes de Nombre (N) está compuesto como se muestra a continuación:*

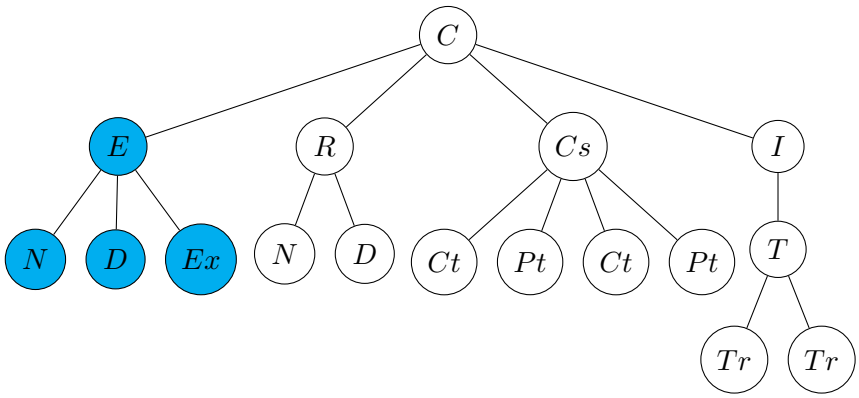


Figura 1.12: Ejemplo gráfico de preceding.

Código 1.24 Expresión ejemplo de preceding.

```
//Nombre/preceding::*
```

Código 1.25 Eje del nodo preceding.

```
<Emisor>
    <Nombre/>
    <Domicilio/>
    <ExpedidoEn/>
</Emisor>

<Nombre/>
<Domicilio/>
<ExpedidoEn/>
```

Solamente se muestra la etiqueta Emisor ya que, devuelve la primera coincidencia que contenga a la etiqueta Nombre.

1.2.2. Caminos y predicados

Las reglas p y q generan caminos y predicados necesarios para crear las rutas exactas al nodo contexto, cuyas etiquetas se desean extraer de los documentos `XML`. Los caminos y predicados tienen la siguiente definición:

Definición 1.1 (Caminos y predicados)**.** *Un camino es:*

1. *Un paso. Se muestran los ejes de los nodos del nivel actual.*
2. *Un paso con prueba en el nodo. Se muestran los ejes de los nodos* β [1].
3. *Condiciones o predicados sobre el camino, donde un predicado es:*
   a) *La conjunción. Considerando dos ejes, se verifica que los dos cumplan la solicitud.*
   b) *La disyunción. Considerando dos ejes, se verifica que al menos uno cumpla la solicitud.*
   c) *La negación. Se verifica que el eje no cumpla la solicitud.*
   d) *Un camino es un predicado.*
4. *La composición de dos caminos* p_1 *y* p_2*. Se muestra al eje del nodo* p_1 *tal que cumpla la condición* p_2*.*

Hay que notar que la definición anterior es mutuamente recursiva ya que, la exactitud de los caminos dependerá de las condiciones que solamente los predicados pueden brindar.

A continuación un ejemplo de cada constructor de caminos:

- Un paso

  Ejemplo 1.15. *Considerando el árbol de la Figura 1.1, todos los ejes de los nodos que sean padre de algún otro:*

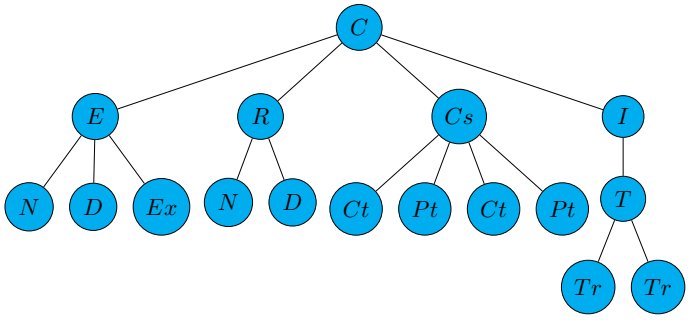


Figura 1.13: Ejemplo gráfico de un paso.

[1] Se denota a β como el identificador de la etiqueta, por ejemplo si la etiqueta es `<Nombre>` entonces $\beta =$ `Nombre`.

Código 1.26 Expresión ejemplo de un paso.

```
//parent::*
```

```
<Comprobante>
<!-- Todo el documento -->
</Comprobante>

<Emisor>
<!-- Contenido de Emisor -->
</Emisor>

<Receptor>
<!-- Contenido de Receptor -->
</Receptor>

<Conceptos>
<!-- Contenido de Conceptos -->
</Conceptos>

<Impuestos>
<!-- Contenido de Impuestos -->
</Impuestos>

<Traslados>
<!-- Contenido de Traslados -->
</Traslados>
```

Código 1.27: Un paso donde todos son padre de alguien.

Nuevamente la representación arbórea es demasiado general ya que se marcan las hojas. Sin

embargo, en el Código 1.27 se observa que no devuelve por separado las etiquetas representadas por las hojas.

- Un paso con prueba

Ejemplo 1.16. *Considerando el árbol de la Figura 1.1, los ejes de los nodos cuyo padre sea la etiqueta Traslados (T) son:*

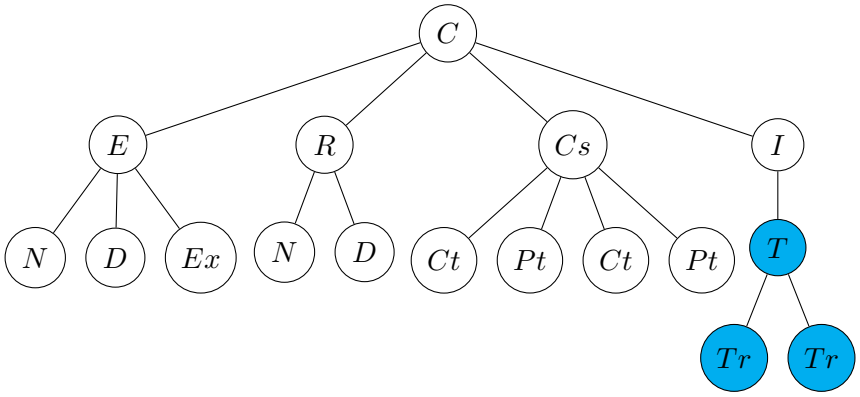


Figura 1.14: Ejemplo gráfico de un paso con prueba.

Código 1.28 Expresión ejemplo de un paso con prueba.

```
//parent::Traslados
```

Código 1.29 Un paso donde la etiqueta padre sea Traslados.

```
<Traslados>
    <Traslado/>
    <Traslado/>
</Traslados>
```

Se observa que se marca el nodo T en la Figura 1.14 ya que se devuelve la etiqueta *Traslados* y no sólo su contenido.

- Camino bajo un predicado

Ejemplo 1.17. *Considerando el árbol de la Figura 1.1, los ejes de los nodos cuyo padre sea la etiqueta Traslados (T) o Emisor (E):*

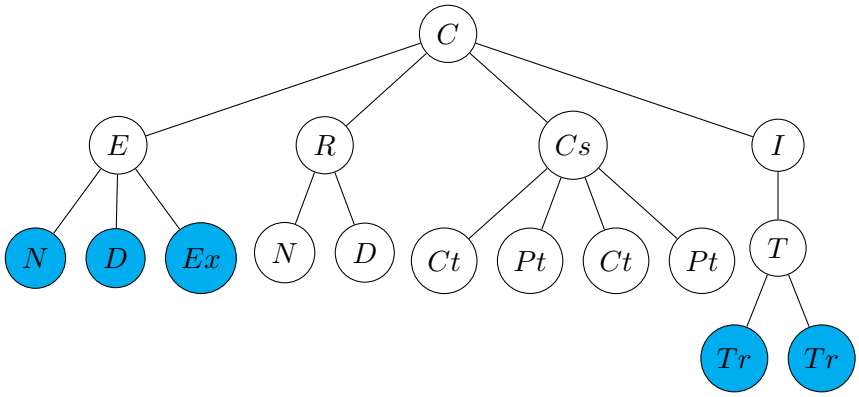


Figura 1.15: Ejemplo gráfico de un camino con predicado.

Código 1.30 Expresión ejemplo de un camino con predicado.

```
//self::*[parent::Traslados or parent::Emisor]
```

Código 1.31 Un paso donde la etiqueta padre sea Traslados o Emisor.

```
<Nombre/>

<Domicilio/>

<ExpedidoEn/>

<Traslado/>

<Traslado/>
```

- Una composición de caminos

  Ejemplo 1.18. *Considerando el árbol de la Figura 1.1, mostramos el camino exacto al nombre del emisor:*

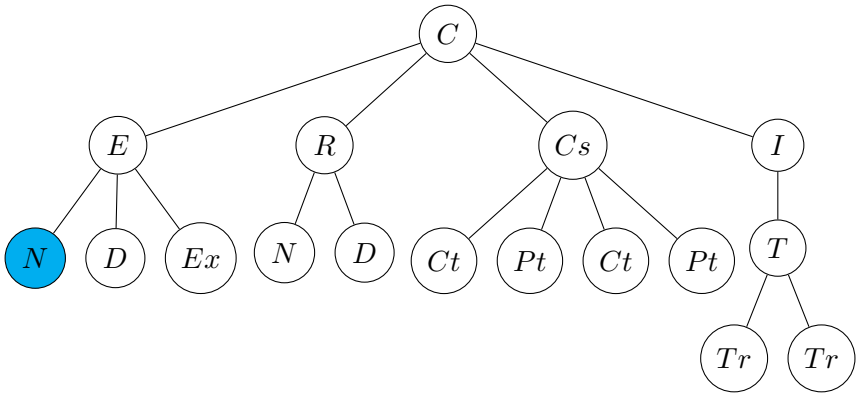


Figura 1.16: Ejemplo gráfico de un camino compuesto.

Código 1.32 Expresión ejemplo de un camino compuesto.

```
//self::Emisor/child::Nombre
```

Código 1.33 Camino exacto a la etiqueta Nombre de Emisor.

```
<Nombre/>
```

Al principio puede resultar complicado dar una expresión camino a partir de una especificación, para hacer mas dócil este proceso hay que guiarse teniendo a la mano el árbol que representa un documento `XML`. Algunos ejemplos de especificación en español y la traducción del Código 1.2:

- *Todas las etiquetas que contengan a Nombre como información.*

Código 1.34 Ejemplo 1 de especificación de caminos.

```
//self::*[child::Nombre]
```

Se obtiene cualquier etiqueta la cual tenga como etiquetas hijas la etiqueta Nombre.

- *Todas las etiquetas que sean padre, excepto la raíz.*

Código 1.35 Ejemplo 2 de especificación de caminos.

```
//parent::*[not(self::Comprobante)]
```

Se obtienen todas las etiquetas padre bajo la condición de que ninguna de ellas sea la etiqueta Comprobante.

- *Todas las etiquetas que estén en el segundo nivel de profundidad.*

Código 1.36 Ejemplo 3 de especificación de caminos.

```
//child::*/child::*/child::*
```

El primer eje child es para considerar las etiquetas n_i inmediatas a la raíz, después se compone el camino con las etiquetas hijas de cada n_i y a su vez componer el camino para tener las etiquetas exclusivas al segundo nivel.

1.2.3. Expresiones xPath

Hasta este momento se ha trabajado sobre rutas relativas. Es decir, no se ha indicado de forma explícita que un nodo contexto sea la raíz o bien un nodo intermedio en un árbol que representa un documento `XML`. Por otra parte, los predicados parecen ser buena opción para obtener etiquetas en distintas partes del documento; sin embargo, no es posible obtenerlas en distintos niveles.

Los elementos de la categoría sintáctica e produce expresiones que satisfacen estas solicitudes y su interpretación es la siguiente:

- $\backslash p$. La consulta comienza en la raíz del árbol.
- p. La consulta comienza en el nodo contexto especificado que aparece al inicio de p.
- $p \cap p$. Devuelve las etiquetas que pertenezcan a ambas consultas.
- $p \cup p$. Devuelve las etiquetas que pertenezcan a alguna de las consultas o incluso ambas.
- Ruta absoluta: expresiones de la forma $\backslash p$.

Ejemplo 1.19. *Considerando el árbol de la Figura 1.1, obtener solamente las etiquetas* `Nombre` *(N) y* `Domicilio` *(D) de la etiqueta* `Receptor` *(R):*

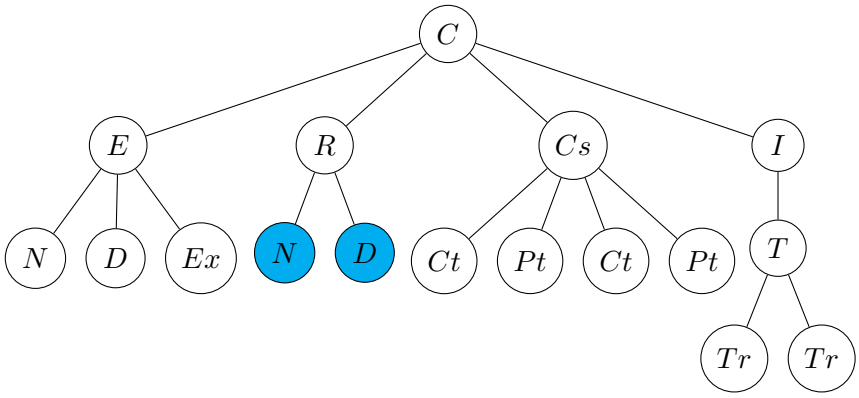


Figura 1.17: Ejemplo gráfico de una ruta absoluta.

Código 1.37 Expresión ejemplo de una ruta absoluta.

```
/child::*/child::Receptor/child::*
```

Código 1.38 Etiquetas de una ruta absoulta.

```
<Nombre/>

<Domicilio/>
```

La expresión en xPath es:

$$\backslash child :: */child :: Receptor/child :: *$$

- Ruta relativa

Ejemplo 1.20. *Considerando el árbol de la Figura 1.1, obtener solamente las etiquetas* `Nombre` *(N) y* `Domicilio` *(D) de la etiqueta* `Receptor` *(R):*

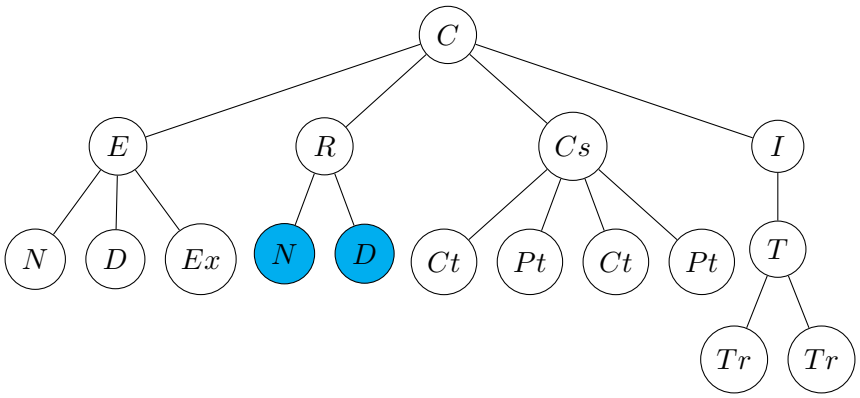


Figura 1.18: Ejemplo gráfico de una ruta relativa.

Código 1.39 Expresión ejemplo de una ruta relativa.

```
//child::Receptor/child::*
```

Código 1.40 Etiquetas de una ruta relativa.

```
<Nombre/>

<Domicilio/>
```

La expresión en xPath es:

$$child :: Receptor/child :: *$$

Obsérvese que, a diferencia de la ruta exacta, en la ruta relativa no es necesario indicar de manera explícita el nivel donde se encuentra el nodo contexto.

- Intersección de rutas

Ejemplo 1.21. *Considerando el árbol de la Figura 1.1, obtener la intersección de los hermanos izquierdos y derechos de* `Receptor` *(R):*

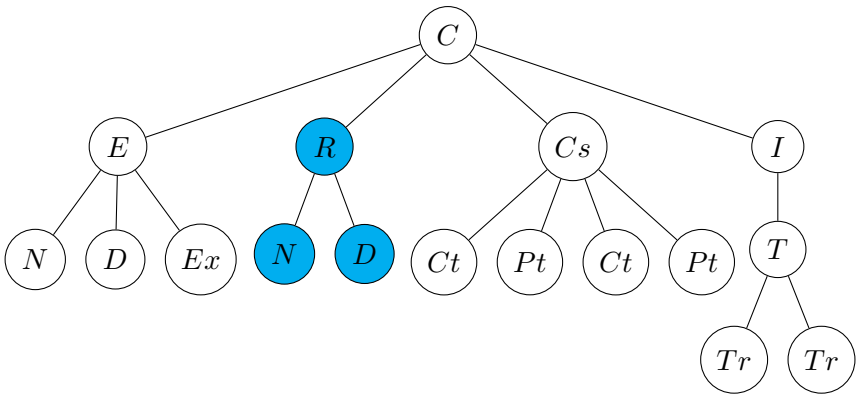


Figura 1.19: Ejemplo gráfico de una intersección de rutas.

Código 1.41 Expresión ejemplo de una intersección de rutas.

```
//following-sibling::Receptor intersect //preceding-sibling::Receptor
```

Código 1.42 Etiquetas de una intersección de rutas.

```
<Receptor>
    <Nombre/>
    <Domicilio/>
</Receptor>
```

La expresión en xPath es:

$$\textit{foll-sibling} :: \textit{Receptor} \cap \textit{prec-sibling} :: \textit{Receptor}$$

- Unión de rutas

Ejemplo 1.22. *Considerando el árbol de la Figura 1.1, obtener la unión de las etiquetas* `Emisor` *(E) y* `Receptor` *(R):*

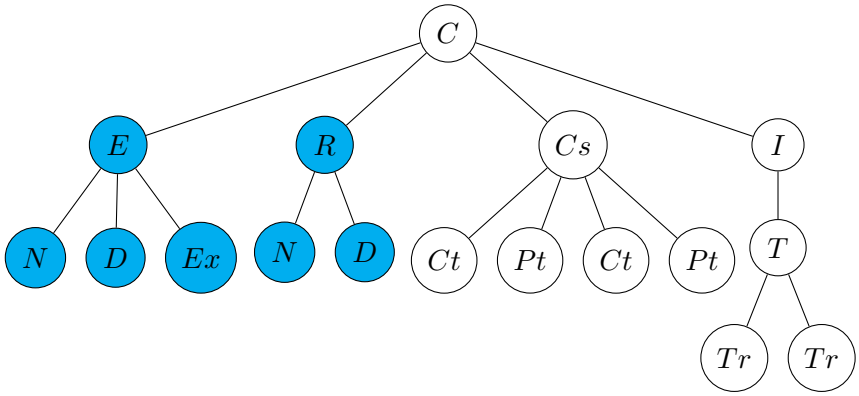


Figura 1.20: Ejemplo gráfico de una unión de rutas.

Código 1.43 Expresión ejemplo de una unión de rutas.

```
//child::Receptor union //child::Emisor
```

Código 1.44 Etiquetas de una unión de rutas.

```
<Receptor>
    <Nombre/>
    <Domicilio/>
</Receptor>

<Emisor>
    <Nombre/>
    <Domicilio/>
    <ExpedidoEn/>
</Emisor>
```

La expresión en xPath es:

$$child :: Receptor \cup child :: Emisor$$

Esto muestra que el lenguaje xPath permite moverse libremente en un documento `XML`; sin embargo, hay que recordar que *no son solamente nodos*, los nodos simplemente son una representación abstracta (y limitada) de las etiquetas.

Con esto finaliza el capítulo del lenguaje xPath, el próximo capítulo se construye el álgebra relacional, teoría necesaria para darle una traducción en relaciones binarias a las funciones de navegación de los zipper, las cuales llamamos *relaciones de navegación*. También se estudiará cómo utilizar dichas relaciones de navegación para consultas utilizando el lenguaje xPath.

Capítulo 2

Ecuaciones relacionales

«(. . .) Incluso si todo número es la suma de dos números primos, es incorrecto que diga que lo sé, al menos hasta que lo haya demostrado.»

Martin-Löf, Per (1983) [24]

En este capítulo se analiza una estructura algebraica llamada álgebra relacional que permite el razonamiento ecuacional sobre relaciones binarias[1], y utilizando juicios se prueban propiedades acerca de las cerraduras *reflexiva-transitiva* y *simétrica*, que más adelante nos sirven para analizar las interpretaciones de las expresiones xPath.

Para revisar la formalización de los resultados mostrados en este capítulo, se deben consultar los archivos siguientes:

- La definición de un álgebra relacional está en el archivo `AR.v`.
- La demostración de que el conjunto de relaciones binarias con ciertos operadores es un álgebra relacional está en el archivo `Rel_AR.v`.
- Las definiciones de ciertos operadores de relaciones binarias y cerraduras definidas inductivamente están en el archivo `Rel_Def.v`.
- Las propiedades de los elementos del punto anterior están en el archivo `Rel_Prop.v`.
- La definición y propiedades de la *relación negada* y la cerradura *reflexiva-transitiva* definida *co-inductivamente* están en el archivo `Rel_Neg.v`.

[1]Se utiliza indistintamente los términos *relaciones binarias* y *relaciones*, ya que en el capítulo únicamente se utilizan relaciones binarias.

2.1. Introducción

La teoría de relaciones binarias desarrollada por De Morgan, Pierce y Schröder [8], fue un intento de formalizar las matemáticas donde los únicos entes matemáticos primitivos son relaciones binarias (sobre un conjunto específico) y operadores sobre éstas. En otras palabras, los conceptos de conjunto y función no existen. Sin embargo existen relaciones binarias que pueden simular a estos entes matemáticos.

Una de las razones para desarrollar esta teoría es trabajar con funciones parciales. En algunas áreas de las matemáticas, trabajar con una función parcial no causa tanto revuelo al tener la posibilidad de decir: *No está definido*. Sin embargo, en el análisis formal de lenguajes y/o verificación formal de programas, esta vía de escape no existe; es necesario tener un elemento que indique dicha frase o bien tratar de utilizar otro tipo de fundamentos.

En lenguajes de programación dicho elemento puede traducirse a un valor que termina de forma abrupta el programa, es decir, el *error*. A pesar de existir un elemento que satisface la necesidad de decir *No está definido*, su presencia podría resultar en problemas bastante graves para aplicaciones delicadas. Por ejemplo, un programa que tiene que calcular la velocidad con la cual una nave espacial debe salir de la superficie terrestre. Actualmente, distintos paradigmas cuentan con mecanismos para intentar cambiar el flujo del programa de tal forma que se evite el *error* [34][1], pero el uso de tales mecanismos no resultan convenientes para los propósitos de este trabajo.

La razón anterior invita a que utilicemos estructuras más débiles, las cuales permiten trabajar sobre las asociaciones entre los elementos de los conjuntos de funciones, aunque no toda asociación esté definida (tal como exige la definición de función). Por ejemplo, consideremos la función parcial f:

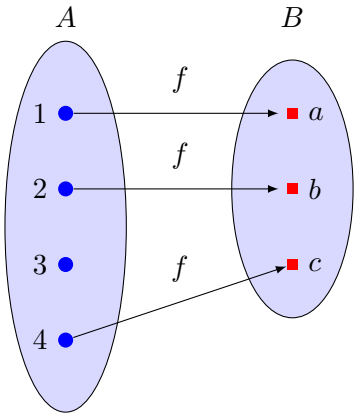


Figura 2.1: Función parcial

Se tiene que $f(3)$ no está definida, lo cual no tiene una manera formal de expresarlo.

[1] Algunos tienen las mónadas, otros las excepciones, etc.

Por otra parte, la teoría de conjuntos establece que toda función en esencia es una relación binaria [2].

Definición 2.1 (Función)**.** *Sean A y B dos conjuntos, se dice que una relación $F \subseteq A \times B$ es una función si para todo $x \in Dom(F)$ hay una única w tal que $(x, w) \in F$. Donde $Dom(F)$ es llamado dominio de la función y es el conjunto de entradas $x \in A$ para los cuales existe una $w \in B$ la cual satisface $(x, w) \in F$, es decir:*

$$Dom(F) = \{x \in A \mid \exists w \in B, (x, w) \in F\}$$

Por ejemplo, la expresión $f(1) = a$ la traducimos a la expresión en términos de relaciones binarias $(1, a) \in f$. En otras palabras, la entrada 1 que pertenece al conjunto A le corresponde el elemento a que pertenece al conjunto B.

Yuta Ikeda y Susumu Nishimura en [16], intercambian el orden de los pares, es decir, dada una función $f : A \to B$ se traduce a la relación binaria $R_f : B \leftarrow A$, lo cual indica que R_f asocia una salida, un elemento del conjunto B, a una entrada que es un elemento del conjunto A. Por ejemplo, la función f la traducimos a la relación binaria $R_f = \{(a, 1), (b, 2), (c, 4)\}$, cuyo diagrama es el siguiente:

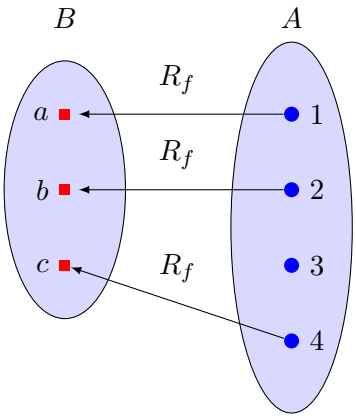


Figura 2.2: Relación representante de f.

Con este nuevo enfoque, decir que $f(3)$ no está definida lo traducimos a: no existe un elemento x en B tal que $(x, 3) \in R_f$.

Obsérvese que este intercambio de pares es importante en la interpretación de resultados, ya que la lectura al ser distinta puede causar errores en los cálculos realizados. Por ejemplo, la composición de relaciones binarias está definida de la siguiente forma:

Definición 2.2. *Dados los conjuntos A, B, C y las relaciones binarias $R : C \leftarrow B$ y $S : B \leftarrow A$, se define la composición de R y S como:*

$$R \circ S = \{(y, x) \mid \exists z \in B, (y, z) \in R \wedge (z, x) \in S\}$$

Entonces considerando un par $(y, x) \in R \circ S$ por definición: *Existe un z tal que a la entrada x le corresponde una salida z dada por R y a la entrada z le corresponde una salida y dada por S.*

Con esta idea, parece suficiente realizar un análisis de las funciones de navegación desde la perspectiva de relaciones binarias. Sin embargo, es necesario una estructura algebraica que sea la base de este análisis, y también dar traducciones a otros elementos del estudio de funciones (como restricción de funciones), en el resto del capítulo se describen y estudian dichos elementos.

2.2. Álgebra Relacional

Una estructura algebraica permite el análisis de conjuntos de elementos y funciones sobre éstos. En el caso del *álgebra relacional* es una estructura algebraica cuyo propósito es analizar ecuaciones de relaciones binarias como posible fundamento de las matemáticas [8]. Para estudiar esta estructura algebraica, es necesario conocer resultados de las estructuras algebraicas *álgebra de Boole* y *monoide*. Las definiciones y propiedades de estas últimas pueden ser consultadas en la obra de Jacobson [19] y el artículo de Huntington [15].

Steve Givant en [8] define la estructura de álgebra relacional de la siguiente forma:

Definición 2.3 (Álgebra relacional)**.** *Sean A un conjunto no vacío, $+, \times, \circ : A \times A \to A$ operadores binarios, $^-, ^\smile : A \to A$ operadores unarios, $1, 0$ y e elementos distinguidos de A. Un álgebra relacional es la 9-tupla $(A, +, \times, ^-, 1, 0, \circ, ^\smile, e)$ tal que cumple las siguientes propiedades:*

1. *$(A, +, \times, ^-, 1, 0)$ es un álgebra de Boole.*
2. *$(A, \circ, e)$ es un monoide.*
3. *Para cualquier $x \in A$ se cumple que $x^{\smile\smile} = x$.*
4. *Para cualesquiera $x, y \in A$ se cumple que $(x \circ y)^\smile = y^\smile \circ x^\smile$*
5. *Para cualesquiera $x, y, z \in A$ se cumple que $(x + y) \circ z = (x \circ z) + (y \circ z)$.*
6. *Para cualesquiera $x, y \in A$ se cumple que $(x + y)^\smile = x^\smile + y^\smile$.*
7. *(Ecuación de Tarski-De Morgan)Para cualesquiera $x, y \in A$ se cumple que $(x^\smile \circ (x \circ y)^-) + y^- = y^-$*

Obsérvese que Steve Givant en [8] utiliza la definición de álgebra de Boole dada por Huntington en [14]. Sin embargo, este último en [15] indica que la definiciones de álgebra de Boole $(A, +, \times, ^-, 1, 0)$ y $(A, +, ^-)$ son equivalentes. Utilizando este hecho se opta por utilizar la primera definición, debido que al tener más operadores y axiomas se reduce el número de sustituciones, la cuales resultan ineficientes al momento de la verificación formal.

El álgebra relacional particular que vamos a usar en este trabajo, es la siguiente:

Lema 2.1. *Sea A un conjunto no vacío, entonces $(\wp(A \times A), \cup, \cap, ^-, \Pi, \emptyset, \circ, ^\smile, Id)$ es un álgebra relacional. Donde:*

- $\cup$ *es la unión de relaciones binarias.*
- $\cap$ *es la intersección de relaciones binarias.*
- $^-$ *es el complemento de relaciones binarias.*
- Π *es la relación binaria total.*
- $\emptyset$ *es la relación binaria vacía.*
- $\circ$ *es la composición de relaciones binarias.*
- $^\smile$ *es la inversa de relaciones binarias.*
- Id *es la relación binaria identidad.*

Demostración. La demostración es directa de las definiciones de los operadores, por lo que la única propiedad que será demostrada es la ecuación de Tarski-De Morgan: Sean R, S dos relaciones binarias sobre A, por demostrar que $(R^\smile \circ (R \circ S)^-) \cup S^- = S^-$. Es decir, hay que demostrar que $(R^\smile \circ (R \circ S)^-) \cup S^- \subseteq S^-$ y que $S^- \subseteq (R^\smile \circ (R \circ S)^-) \cup S^-$.

- Demostrar $S^- \subseteq (R^\smile \circ (R \circ S)^-) \cup S^-$ es directo de la definición de la unión.
- Sea $(x, y) \in (R^\smile \circ (R \circ S)^-) \cup S^-$ por demostrar que $(x, y) \in S^-$.

  Por hipótesis y definición de la unión hay que analizar dos casos:

  - Si $(x, y) \in S^-$, entonces queda demostrado.
  - Si $(x, y) \in (R^\smile \circ (R \circ S)^-)$, entonces por definición de la composición existe un $x_1 \in A$ tal que $(x, x_1) \in R^\smile$ y $(x_1, y) \in (R \circ S)^-$.

Supongamos que $(x, y) \notin S^-$, entonces por definición del complemento se tiene que $(x, y) \in S$. Por definición del inverso se tiene que $(x_1, x) \in R$, es decir, $(x_1, y) \in R \circ S$. Lo cual contradice el hecho de que $(x_1, y) \in (R \circ S)^-$.

$\boxed{\lambda}$

Esta estructura nos da el razonamiento que necesitaremos para la interpretación relacional del lenguaje xPath, sin embargo, suponiendo que se tiene una relación R. ¿Cómo verificar qué parámetros de una función pertenecen a un conjunto A de valores? En funciones esto se traduce a restringir una función a un conjunto. Para esto se tendría que inducir una relación a partir de A, tal que su composición con R filtre las entradas a elementos de A [16]. Este tipo de relaciones inducidas son llamadas *relaciones correflexivas*, en la siguiente sección se hará una construcción y análisis sobre este tipo de relaciones.

2.2.1. Relaciones correflexivas

Restringir una función significa filtrar elementos del conjunto dominio. Es decir, si hay una función $f : A \to B$ tal que existe un conjunto $C \subseteq A$ y solamente se quiere trabajar sobre los elementos de C, entonces $f\restriction_C$ cumple el cometido. Por ejemplo, en la Figura 2.3 se considera a $C = \{1, 2\} \subseteq A$, por lo que se puede restringir a f sobre el conjunto C.

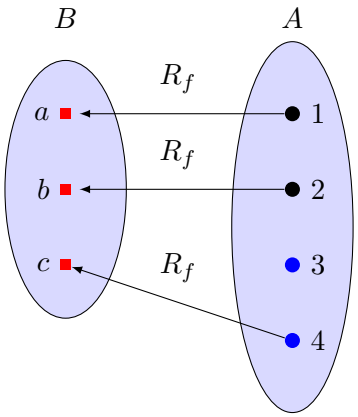


Figura 2.3: Relación representante de f con elementos distinguidos, los cuales son los elementos de C.

Con los elementos disponibles es imposible restringir una relación binaria a un conjunto de elementos ya que, no existe el concepto de conjunto como tal. Sin embargo, la idea de restringir una función a cierto subconjunto de elementos del dominio significa verificar que $(y, x) \in R_{f\restriction_C}$, esto se cumple si existe un elemento $z \in C$ tal que $x = z$ y la pertenencia $(y, z) \in R_{f\restriction_C}$ es válida.

Definición 2.4 (Relación Correflexiva)**.** *Sean A un conjunto y R una relación binaria sobre A, se dice que R es* ***correflexiva*** *si:*

$$R \subseteq Id$$

La definición indica que una relación correflexiva siempre está contenida en la relación de igualdad, es decir, si $(z, x) \in R$ con R correflexiva entonces $z = x$. Por ejemplo, la función $f\restriction_C$ mostrada en la Figura 2.3 se traduce a la relación binaria $R_f \circ C?$, donde $C? = \{(1,1),(2,2)\}$, y su representación visual está dada en la Figura 2.4.

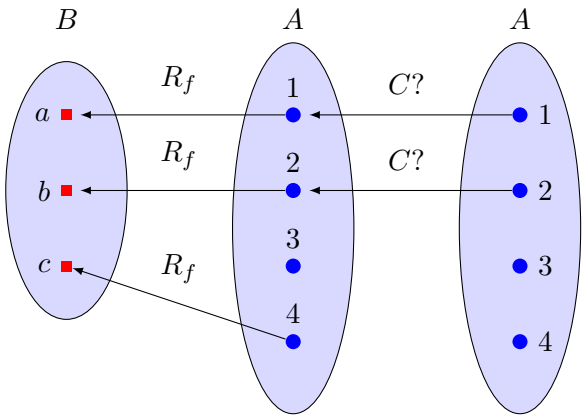


Figura 2.4: Relación representante de $f\restriction_C$.

La relación binaria $C?$ se llama *relación correflexiva inducida por C*. La definición general es la siguiente:

Definición 2.5 (Relación correflexiva inducida)**.** *Sea A el conjunto base de las relaciones binarias y $C \subseteq A$ entonces la relación correflexiva inducida por C, denotada $C?$, se define como:*

$$C? = \{(x, y) \in A \times A \mid x \in C \wedge x = y\}$$

A continuación se enuncian las propiedades de mayor importancia de las relaciones binarias correflexivas.

Proposición 2.1. *Las relaciones correflexivas son idempotentes respecto a la composición, es decir, dado un conjunto A y una relación binaria correflexiva R sobre A, se cumple que:*

$$R = R \circ R$$

Demostración. Sea R una relación correflexiva sobre A, por demostrar:

- $R \circ R \subseteq R$

  Sean $(x, y) \in R \circ R$, por definición existe un $z \in A$ tal que $(x, z) \in R$ y $(z, y) \in R$. Por

definición de correflexividad se tiene que $x = z$ y $z = y$, entonces por transitividad de la igualdad se concluye que $(x, y) \in R$.

- $R \subseteq R \circ R$

  Sean $(x, y) \in R$, por hipótesis y definición de correflexividad se tiene que $x = y$, entonces por reflexividad de la igualdad se concluye que $(x, y) \in R \circ R$.

$\boxed{\lambda}$

Lema 2.2. *Las relaciones correflexivas conmutan respecto a la composición, es decir, para cualesquiera A conjunto y R, S relaciones binarias correflexivas sobre A, se cumple que $R \circ S = S \circ R$.*

Demostración. Sean R, S relaciones correflexivas sobre A, por demostrar:

- $R \circ S \subseteq S \circ R$

  Sean $(x, y) \in R \circ S$, por definición existe un $z \in A$ tal que $(x, z) \in R$ y $(z, y) \in S$. Por definición de correflexividad se tiene que $x = z$ y $z = y$, entonces por simetría de la igualdad e hipótesis se tiene que $(x, z) \in S$. Análogamente se concluye que $(z, y) \in R$.

- $S \circ R \subseteq R \circ S$

  El razonamiento es análogo al anterior.

$\boxed{\lambda}$

Un aspecto primordial en el estudio de funciones es el conjunto dominio de una función. Por otra parte, las relaciones correflexivas son relaciones binarias para simular conjuntos. Entonces necesitamos definir una relación correflexiva que permita filtrar los elementos del dominio de una relación binaria. Yuta Ikeda y Susumu Nishimura (2011) en [16] lo indican de la siguiente forma:

> *El dominio de una relación binaria es la relación correflexiva que representa el conjunto de entradas que tienen una única salida correspondiente.* (p. 3)

Definición 2.6 (Dominio)**.** *Sean A un conjunto y R una relación sobre A, definimos el dominio de R de la siguiente forma:*

$$dom(R) = Id \cap (R^{\smile} \circ R)$$

La relación binaria $R^{\smile} \circ R$ indica que los elementos solamente pueden pertenecer al dominio si *hay un camino de ida y regreso*. Por ejemplo, considerando la siguiente función total f cuyo

dominio es $A = \{1, 2, 3, 4\}$ e imagen $B = \{a, b, c, d\}$:

$$f(1) = b$$
$$f(2) = a$$
$$f(3) = d$$
$$f(4) = c$$

La relación binaria que representa a f es:

$$R_f = \{(b, 1), (a, 2), (d, 3), (c, 4)\}$$

En la Figura 2.5 se muestra la relación $R_f^{\smile} \circ R_f$.

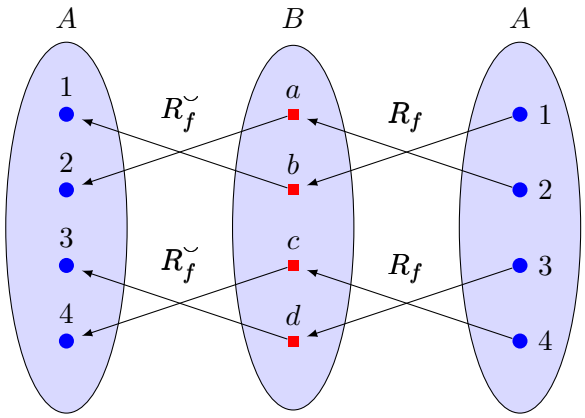


Figura 2.5: Relación $R_f^{\smile} \circ R_f$.

Originalmente el dominio de f es $\{1, 2, 3, 4\}$ y en el caso del dominio de R_f consiste en los pares $\{(1, 1), (2, 2), (3, 3), (4, 4)\}$, ya que $(1, 1) \in Id$ y existe b tal que $(1, b) \in R_f^{\smile}$ y $(b, 1) \in R_f$, análogamente se demuestra para los pares $(2, 2)$,$(3, 3)$ y $(4, 4)$.

Parece que no tiene sentido incluir la relación identidad en la definición. Sin embargo, en el caso de funciones parciales filtra aquellas entradas que no pueden pertenecer al dominio. Por ejemplo, considerando la función parcial g con dominio $A = \{1, 2, 3, 4\}$ e imagen $B = \{a, b, c\}$:

$$g(1) = b$$
$$g(2) = a$$
$$g(3) = c$$
$$g(4) = c$$

La relación binaria representante es:

$$R_g = \{(b, 1), (a, 2), (c, 3), (c, 4)\}$$

En la Figura 2.6 se representa la relación $R_g^{\smile} \circ R_g$.

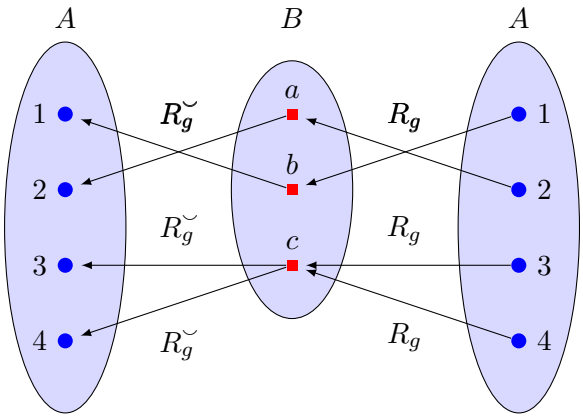


Figura 2.6: Relación $R_g^\smile \circ R_g$.

En el diagrama anterior se observa que el par $(4,3) \in R_g^\smile \circ R_g$, sin embargo esto no puede pertenecer al dominio de g ya que son valores distintos de entrada, es por ello la necesidad de intersectar con la relación identidad.

Una propiedad importante del dominio es su correflexividad.

Lema 2.3. *El dominio de una relación es una relación correflexiva.*

Demostración. Se sigue de la definición de dominio. $\boxed{\lambda}$

A continuación presentamos otras propiedades acerca del dominio de una relación binaria.

Teorema 2.1. *Sean A un conjunto y R una relación sobre A, entonces se cumple la siguiente ecuación:*

$$R = R \circ dom(R)$$

Demostración. Sea R una relación binaria sobre A, por demostrar:

- $R \subseteq R \circ dom(R)$

  Sean $(x,y) \in R$, basta demostrar que $(y,y) \in dom(R)$. Por definición del dominio se tiene que $(y,y) \in Id$, basta demostrar que $(y,y) \in R^\smile \circ R$. Por definición de la relación inversa y la composición de relaciones, existe un elemento $x \in A$, tal que $(y,x) \in R^\smile$ y $(x,y) \in R$, lo cual concluye esta contención.

- $R \circ dom(R) \subseteq R$

  Sean $(x, y) \in R \circ dom(R)$, entonces por definición existe un $z \in A$ tal que $(x, z) \in R$ y $(z, y) \in dom(R)$. Por definición de dominio se tiene que $z = y$, por lo que $(x, y) \in R$.

$\boxed{\lambda}$

Corolario 2.1. *Sean A un conjunto y R una relación sobre A. Entonces se cumple la siguiente ecuación:*

$$\Pi \circ dom(R) = \Pi \circ R$$

Demostración. Se sigue del Teorema 2.1, la definición de dominio y propiedades sobre la relación total. $\boxed{\lambda}$

Si una relación cumple ser correflexiva, entonces no se está modelando una función más bien se está modelando un conjunto.

Corolario 2.2. *Sean A un conjunto y R una relación correflexiva sobre A. Entonces se cumple la siguiente ecuación:*

$$dom(R) = R$$

Demostración. Sea R una relación correflexiva sobre A, por demostrar:

- $dom(R) \subseteq R$

  Sean $(x, y) \in dom(R)$, por definición $(x, y) \in Id$ y existe un $z \in A$ tal que, $(x, z) \in R^{\smile}$ y $(z, y) \in R$. Por definición de relación inversa y relación identidad se tiene que $z = x$, por lo que $(x, y) \in R$.

- $R \subseteq dom(R)$

  Se sigue del Teorema 2.1 y la correflexividad de R.

$\boxed{\lambda}$

Lema 2.4. *Sean A un conjunto y R, S relaciones correflexivas sobre A, entonces $R \cup S$ y $R \cap S$ son relaciones correflexivas.*

Demostración. Las demostraciones son directas de la definición de correflexividad y propiedades de la igualdad. $\boxed{\lambda}$

Lema 2.5. *Sean A un conjunto y R, S relaciones correflexivas sobre A, entonces se cumple la siguiente ecuación:*

$$R \cap S = R \circ S = S \circ R$$

Demostración. Se sigue del Lema 2.4, el Lema 2.2 y las definiciones de composición y correflexividad. $\boxed{\lambda}$

Buscamos que las funciones de navegación tengan inversa para que sea posible navegar de forma libre en la estructura almacenada en el zipper. Parte de esto se lograría si las funciones de navegación fuesen inyectivas. Hay que recordar que una función $f : A \to B$ es inyectiva si tiene inversa izquierda, es decir, si existe una función $g : B \to A$ tal que $g \circ f = Id_A$.

La Figura 2.7 muestra como la relación binaria que representa a la función parcial f, con dominio $A = \{1, 2, 3, 4\}$ e imagen $B = \{a, c, d\}$, respeta la noción de inyectividad:

$$f(1) = a$$
$$f(2) = d$$
$$f(3) = c$$

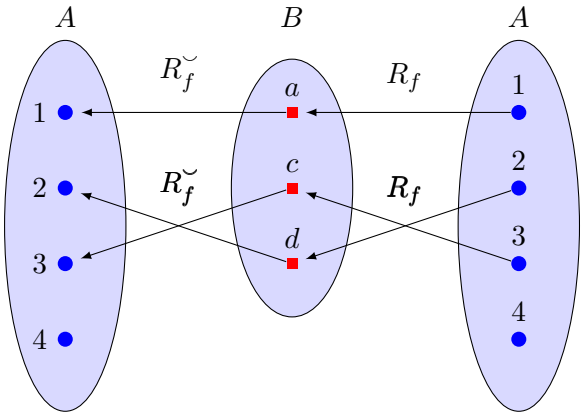


Figura 2.7: Relación representante de una función parcial que cumple inyectividad.

Entonces, la manera natural de definir una relación binaria inyectiva es: la composición resultante de la relación inversa y la original debe estar contenida en la identidad.

Definición 2.7 (Inyectiva)**.** *Sean A un conjunto y R una relación sobre A, se dice que R es inyectiva si:*

$$R^{\smile} \circ R \subseteq Id$$

En el Ejemplo 2.7 esto se cumple: Se tiene que $(3, 3) \in R_f \circ R_f^{\smile}$ pues por hipótesis el elemento $c \in B$ cumple que $(3, c) \in R_f^{\smile}$ y $(c, 3) \in R_f$ y se tiene que $c = c$, análogamente para el resto de

casos. Un ejemplo de una relación no inyectiva se muestra en la Figura 2.8, ya que por hipótesis $(3,4) \in R_f^{\smile} \circ R_f$ pero $3 \neq 4$.

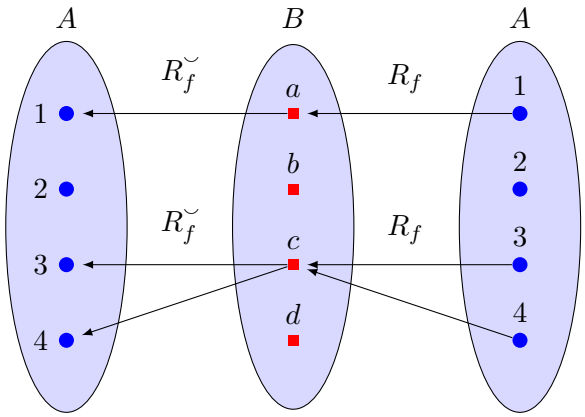


Figura 2.8: Relación representante de una función parcial que no cumple inyectividad.

Si la composición es al revés, es decir, si la composición de una relación binaria con su inversa está contenida en la relación identidad, entonces a este tipo de relaciones binarias se llaman relaciones binarias simples.

Definición 2.8 (Simple)**.** *Sean A un conjunto y R una relación sobre A, se dice que R es simple si:*

$$R \circ R^{\smile} \subseteq Id$$

Es posible obtener más resultados acerca de las traducciones de funciones a relaciones binarias, por ejemplo: Si f es una función total entonces la relación representante R_f es simple. Sin embargo, no son de interés para este trabajo.

Por otra parte, las relaciones simples e inyectivas permitirán el análisis de simetría e inyectividad necesarios para la composición de las *relaciones de navegación*.

Teorema 2.2. *Sean A un conjunto, R, S relaciones sobre A tal que S es inyectiva. Entonces se cumple la siguiente ecuación:*

$$dom(R \circ S) = S^{\smile} \circ dom(R) \circ S$$

Demostración. Se sigue de las propiedades de álgebra relacional y las definiciones de dominio e inversa de una relación. $\boxed{\lambda}$

La ecuación anterior es una herramienta auxiliar importante en la composición de relaciones inyectivas, para esto utilizamos la *cerradura reflexiva-transitiva* [16].

2.3. Cerraduras

El principal interés del estudio de las cerraduras reflexiva-transitiva y simétrica, es observar el comportamiento de la composición de varias relaciones binarias. Antes de definir las cerraduras de una relación binaria hay que recordar la definición de cerradura [29].

Definición 2.9 (Cerradura)**.** *Sean A un conjunto, R una relación sobre A y $\mathcal{P}$ una propiedad sobre relaciones, tal que R puede satisfacer $\mathcal{P}$ agregando parejas. Definimos la cerradura de R con respecto a $\mathcal{P}$ como la relación binaria más pequeña que contiene a R y además cumple $\mathcal{P}$.*

Por ejemplo la cerradura reflexiva se define de la siguiente forma:

Definición 2.10 (Cerradura reflexiva)**.** *Sea A un conjunto y R una relación sobre A, se define la cerradura reflexiva de R como:*

$$refl(R) = R \cup Id$$

Antes de proseguir con el tema de cerraduras, como se dijo antes elegimos el sistema de juicios en lugar de los puntos fijos dados por el teorema de Kanster-Tarski ya que, resulta más natural trabajar con los sistemas de juicios en el asistente de pruebas `COQ`. A pesar de que en este trabajo no usamos los puntos fijos obtenidos por el teorema de Knaster-Tarski, en la siguiente sección damos una breve introducción a este tema, con el fin de una mejor comprensión de las traducciones hechas.

2.3.1. La negativa al teorema de Knaster-Tarski

Una de las bases de `COQ` es la implementación de la teoría de tipos intuicionista propuesta por Martin Löf, dicha teoría define los tipos inductivos a través de constructores llamados *tipos producto*, *tipos suma*, etc. y reglas de inferencia para dichos constructores. En otras palabras, la teoría de tipos de Martin Löf descrita en [24] consiste en una serie de mecanismos puramente sintácticos que permiten a la computadora realizar cálculos sin necesidad de significado, el programador será el encargado de darle la interpretación a los resultados.

A diferencia de la teoría de tipos de Martin Löf, en la teoría de órdenes construimos nuevos elementos a través de una relación de orden. Para lograr esto necesitamos de la siguiente estructura algebraica [29].

Definición 2.11 (Retícula)**.** *Una retícula o latiz es un conjunto parcialmente ordenado A tal que para cualesquiera $x, y \in A$, tanto $x \sqcup y$ (supremo) como $x \sqcap y$ (ínfimo) existen, es decir, $x \sqcup y \in A$ y $x \sqcap y \in A$.*

Esta estructura asegura la existencia de elementos ínfimos y supremos, sin embargo, para la construcción de nuevos elementos, es necesario que se cumpla esta propiedad para cualquier subconjunto de elementos de una retícula A.

Definición 2.12 (Retícula Completa)**.** *Una retícula A es completa si para cualquier $B \subseteq A$, el conjunto B tiene ínfimo y supremo.*

Con esto, considerando una función monótona creciente f existen elementos los cuales quedan fijos bajo la aplicación de f y la relación de orden de la retícula con la que se trabaja [33].

Teorema 2.3 (Teorema de Knaster-Tarski)**.** *En una retícula completa si f es una función monótona creciente de relaciones, entonces existen el mínimo punto fijo, denotado $\mu X.f(X)$, y el máximo punto fijo, denotado $\nu X.f(X)$, ambos son solución de la ecuación $X = f(X)$.*

Demostración. La demostración puede consultarse en [33]. $\boxed{\lambda}$

Entonces en el caso de la retícula de las relaciones binarias sobre un conjunto A y la contención como relación de orden, el mínimo punto fijo resulta ser la relación más pequeña que indica la definición de cerradura.

Corolario 2.3. *En una retícula completa si f es una función monótona creciente de relaciones, entonces el mínimo punto fijo de f cumple las siguientes propiedades:*

- *$\mu-$cómputo:* $\mu X.f(X) = f(\mu X.f(X))$
- *$\mu-$inducción:* $f(R) \subseteq R \Rightarrow \mu X.f(X) \subseteq R$

Demostración. La demostración puede consultarse en [33]. $\boxed{\lambda}$

La forma de trabajar con estos resultados es la aplicación de $\mu-$cómputo sustituyendo el punto fijo por $f(\mu X.f(X))$ o bien asumiendo que se cumple que $f(R) \subseteq R$, utilizar $\mu-$inducción para demostrar que el mínimo punto fijo de una función monótona creciente está contenido en una relación R.

Por ejemplo Chritiene Aarts *et al.* en [1] indican que el mínimo punto fijo cumple monotonía, esta prueba es un ejemplo del razonamiento utilizando el mínimo punto fijo:

Lema 2.6 (μ-Monotonía)**.** *Para cualesquiera f,g funciones monótonas crecientes sobre relaciones binarias, si para toda relación R se cumple que $f(R) \subseteq g(R)$, denotado $f \subseteq g$, entonces:*

$$\mu X.f(X) \subseteq \mu X.g(X)$$

Demostración. Sean f,g funciones monótonas crecientes sobre relaciones binarias y se supondrá que $f \subseteq g$, por demostrar que $\mu X.f(X) \subseteq \mu X.g(X)$.

Por hipótesis se tiene que $f(\mu X.g(X)) \subseteq g(\mu X.g(X))$, entonces por la regla de μ-cómputo aplicada a g, se tiene que $f(\mu X.g(X)) \subseteq \mu X.g(X)$.
Denotando a $\mu X.g(X)$ como G entonces se tiene que $f(G) \subseteq G$ y finalmente por μ-inducción se concluye que:

$$\mu X.f(X) \subseteq \mu X.g(X)$$

$\boxed{\lambda}$

Este tipo de cálculos parecen convenientes para el razonamiento relacional. Sin embargo, traducir esta forma de razonamiento resultaría ineficiente, por lo que el sistema de juicios es más adecuado en la verificación formal hecha en `COQ`. Por otra parte, el sistema de juicios requiere que las expresiones de la forma $\mu X.f(X)$ queden solamente en sintaxis y eso es algo complicado de realizar; esto se observará en la traducción de la cerradura simétrica propuesta por Yuta Ikeda y Susumu Nishimura en [16].

2.3.2. La cerradura reflexiva-transitiva

En lenguajes de programación, la cerradura reflexiva-transitiva se usa para formalizar la evaluación de una expresión. Esta cerradura se define en distintas teorías, por ejemplo: la teoría de conjuntos [29], teoría de órdenes [16] y sistemas de juicios [10].

En el sistema de juicios permite definir estructuras de forma puramente sintáctica, lo cual es preferible al momento de realizar una verificación formal[1]. Por ejemplo utilizando el sistema de juicios, la cerradura reflexiva-transitiva está definida de la siguiente forma:

Definición 2.13. *Sean A un conjunto y R una relación sobre A. La cerradura reflexiva-transitiva de R, denotada R^*, se define con el siguiente sistema de juicios:*

$$\frac{(x,y) \in Id}{(x,y) \in R^*}\, crt_1 \qquad \frac{(x,y) \in R \qquad (y,z) \in R^*}{(x,z) \in R^*}\, crt_2$$

En este trabajo, para demostrar propiedades usando juicios hacemos lo siguiente: Si las hipótesis (propiedades arriba de la línea del juicio) se cumplen, entonces podemos ejecutar el juicio (propiedades debajo de la línea del juicio) para obtener una nueva propiedad. Por ejemplo, para demostrar que la cerradura reflexiva-transitiva de cualquier relación cumple la propiedad de reflexividad hacemos el siguiente razonamiento.

Lema 2.7. *Sean A un conjunto y R una relación binaria sobre A, entonces R^* es reflexiva.*

Demostración. Sea $x \in A$, dado que se cumple la igualdad sintáctica $x = x$, es claro que $(x,x) \in Id$. Entonces por la regla crt_1 se concluye que $(x,x) \in R^*$ para cualquier $x \in A$.

$\boxed{\lambda}$

[1] La computadora solamente trabaja con símbolos, no comprende significados.

La regla crt_2, del sistema de juicios que define a la cerradura reflexiva-transitiva, es una regla recursiva. Es decir, la expresión R^* aparece tanto en la hipótesis como en la conclusión. Al tener este tipo de reglas, es posible definir un principio de inducción sobre la estructura definida. Es importante observar que el principio de inducción que mostramos a continuación, no corresponde al principio de inducción estructural usual. Este principio de inducción corresponde a la llamada recursión primitiva, que brinda una hipótesis de inducción más fuerte, obligando a que se cumpla a la propiedad en elementos «más pequeños» que además pertenezcan a la estructura en cuestión.

Por ejemplo, el principio de inducción para la cerradura reflexiva-transitiva es el siguiente:

Definición 2.14 (Principio de inducción para la cerradura reflexiva-transitiva)**.** *Para cualesquiera A conjunto, R relación binaria sobre A y $\mathcal{P}$ un juicio binario sobre elementos de A, si se cumplen:*

1. *Para cualesquiera $x, w \in A$, si $(x, w) \in Id$, entonces $(x, w)\ \mathcal{P}$.*

2. *Para cualesquiera $x, w, z \in A$, si $(x, w) \in R$, $(w, z) \in R^*$ y $(w, z)\ \mathcal{P}$, entonces $(x, z)\ \mathcal{P}$*

Entonces para cualesquiera $y, y_0 \in A$, si $(y, y_0) \in R^*$ entonces $(y, y_0)\ \mathcal{P}$. Donde $(x, y)\ \mathcal{P}$ indica que el par (x, y) satisface el juicio $\mathcal{P}$.

La notación $(x, y)\ \mathcal{P}$ es para diferenciar entre las relaciones binarias con las que se está trabajando y una propiedad $\mathcal{P}$ que deben satisfacer dichos elementos.

Observamos que en el punto 2 del principio de inducción, además de suponer $(w, z)\ \mathcal{P}$ tenemos como hipótesis la pertenencia $(w, z) \in R^*$. A continuación, un ejemplo de cómo utilizar este principio de inducción para demostrar que la cerradura reflexiva-transitiva cumple la propiedad de transitividad.

Lema 2.8. *Sean A un conjunto y R una relación binaria sobre A, entonces R^* es transitiva.*

Demostración. Si $(x, y) \in R^*$ y $(y, z) \in R^*$ basta demostrar que $(x, z) \in R^*$ [1]. Se procede por inducción sobre $(x, y) \in R^*$:

- **Caso base:** Sea $(x, y) \in Id$ y por hipótesis se tiene que $(y, z) \in R^*$, entonces por definición de la relación Id, se cumple que $(x, z) \in R^*$.

- **Hipótesis de inducción:** Sea $z_0 \in A$ tal que si $(z_0, z) \in R^*$ entonces $(y, z) \in R^*$.

- **Paso inductivo:** Sean $(x, y) \in R$, $(y, z_0) \in R^*$ y $(z_0, z) \in R^*$, por demostrar que $(x, z) \in R^*$.

[1] En este caso $\mathcal{P}$ corresponde a la propiedad de transitividad.

Por hipótesis de inducción se tiene que $(y, z) \in R^*$, dado que $(x, y) \in R$ y por la regla crt_2 se concluye que $(x, z) \in R^*$.

$\boxed{\lambda}$

Yuta Ikeda y Susumu Nishimura en [16] construyen de manera indirecta la composición $R^* \circ S$, modificando el caso base de la cerradura reflexiva-transitiva mediante las siguientes ecuaciones con punto fijo:

$$R^* \circ S = \mu X.(S \cup R \circ X) \tag{2.1}$$
$$S \circ R^* = \mu X.(S \cup X \circ R) \tag{2.2}$$

Uno de los fines de esta tesis es traducir las expresiones dadas por el teorema de Knaster-Tarski a un sistema de juicios, a continuación el estudio de la relación $R^* \circ S$.

Definición 2.15. *Sean A un conjunto y R, S relaciones binarias sobre A, una forma de definir $R^* \circ S$ es la siguiente:*

$$\frac{(x, y) \in S}{(x, y) \in cd(R, S)}\, compDer_1 \qquad \frac{(x, y) \in R \qquad (y, z) \in cd(R, S)}{(x, z) \in cd(R, S)}\, compDer_2$$

Las relaciones $R^* \circ S$ y $cd(R, S)$ intuitivamente significan lo mismo: *El par (x, y) puede pertenecer a R^* y debe existir un z tal que el par (y, z) pertenezca a S.* Ya que como veremos en un momento, algunas interpretaciones de expresiones xPath resultan en relaciones de la forma $R^* \circ S$, mientras que el resultado de otras interpretaciones son de la forma $cd(R, S)$.

Obsérvese que el principio de inducción para la cerradura $cd(R, S)$ es muy parecido al principio de inducción para la cerradura reflexiva-transitiva.

Lema 2.9. *Sean A un conjunto, R, S dos relaciones sobre A, entonces:*

$$R^* \circ S = cd(R, S)$$

Demostración. Sean R, S dos relaciones sobre A. Por demostrar las siguientes inclusiones:

- $R^* \circ S \subseteq cd(R, S)$

  Sean $(x, y) \in R^* \circ S$, entonces por definición existe un $z \in A$ tal que $(x, z) \in R^*$ y $(z, y) \in S$.

  Se procede con inducción sobre $(x, z) \in R^*$.

  - **Caso base:** Sea $(x, z) \in Id$, entonces por hipótesis se tiene que $(x, y) \in S$. Finalmente por la regla $compDer_1$ se concluye que $(x, y) \in cd(R, S)$.
  - **Hipótesis de inducción:** Sea $y_0 \in A$ tal que si $(z, y) \in S$, entonces $(y_0, y) \in cd(R, S)$.

- **Paso inductivo:** Sean $(x, y_0) \in R$, $(y_0, z) \in R^*$ y $(z, y) \in S$, por demostrar que $(x, y) \in cd(R, S)$.

  Dado que $(x, y_0) \in R$ y por hipótesis de inducción se tiene que $(y_0, y) \in cd(R, S)$, entonces por la regla $compDer_2$ se concluye que $(x, y) \in cd(R, S)$.

- $cd(R, S) \subseteq R^* \circ S$

  Sea $(x, y) \in cd(R, S)$, por demostrar que $(x, y) \in R^* \circ S$, por definición de composición se procede por inducción sobre $(x, y) \in cd(R, S)$.

  - **Caso base:** Sea $(x, y) \in S$, por demostrar que $(x, y) \in R^* \circ S$.

    Por el Lema 2.7 existe $x \in A$, tal que $(x, x) \in R^*$, entonces por la hipótesis se concluye que $(x, y) \in R^* \circ S$.

  - **Hipótesis de inducción:** Sea $z \in A$ tal que $(y, z) \in R^* \circ S$.

  - **Paso inductivo:** Sean $(x, y) \in R$ y $(y, z) \in cd(R, S)$, por demostrar que $(x, z) \in R^* \circ S$.

    Por hipótesis de inducción y definición de composición existe un $x_0 \in A$ tal que $(y, x_0) \in R^*$ y $(x_0, z) \in S$, entonces por la regla crt_2 e hipótesis se tiene que $(x, x_0) \in R^*$. Dado que $(x_0, z) \in S$, se concluye que $(x, z) \in R^* \circ S$.

$\boxed{\lambda}$

De manera análoga a la Definición 2.15 definimos los juicios para $S \circ R^*$.

Definición 2.16. *Sean A un conjunto y R, S relaciones binarias sobre A, una forma de definir $S \circ R^*$ es la siguiente:*

$$\frac{(x, y) \in S}{(x, y) \in ci(S, R)}\, compIzq_1 \qquad \frac{(x, y) \in ci(S, R) \qquad (y, z) \in R}{(x, z) \in ci(S, R)}\, compIzq_2$$

Al igual que la Definición 2.15, la definición anterior y la expresión $S \circ R^*$ tienen un mismo significado: *El par (x, y) debe pertenecer a S y pueden existir cero o más composiciones tal que para un elemento z cause que el par (y, z) pertenezca a R^*.*

Para demostrar la Ecuación 2.2 es necesario dar una definición equivalente a la cerradura reflexiva-transitiva.

Definición 2.17. *Sean A un conjunto y R una relación sobre A. La cerradura reflexiva-transitiva de R, denotada R^{*_2}, se define de la siguiente forma:*

$$\frac{(x,y) \in Id}{(x,y) \in R^{*_2}}\, crt2_1 \qquad \frac{(x,y) \in R^{*_2} \qquad (y,z) \in R}{(x,z) \in R^{*_2}}\, crt2_2$$

El principio de inducción para esta definición es el siguiente:

Definición 2.18 (Principio de inducción para la cerradura reflexiva-transitiva-2)**.** *Para cualesquiera A conjunto, R relación binaria sobre A y $\mathcal{P}$ un juicio binario sobre elementos de A, si se cumplen:*

- *Para cualesquiera $x, y \in A$, si $(x,y) \in Id$, entonces $(x,y)\ \mathcal{P}$.*
- *Para cualesquiera $x, y, z \in A$, si $(x,y) \in R^{*_2}$, $(x,y)\ \mathcal{P}$ y $(y,z) \in R$, entonces $(x,z)\ \mathcal{P}$.*

Entonces para cualesquiera $y, y_0 \in A$, si $(y, y_0) \in R^{*_2}$ entonces $(y, y_0)\ \mathcal{P}$.

Resulta claro que la cerradura anterior es transitiva, sin embargo su demostración no es tan intuitiva como la del Lema 2.10.

Lema 2.10. *Sean A un conjunto y R una relación binaria sobre A, entonces R^{*_2} es transitiva.*

Demostración. Sean $(x,y) \in R^{*_2}$ y $(y,z) \in R^{*_2}$, por demostrar que $(x,z) \in R^{*_2}$. Se procede por inducción sobre $(y,z) \in R^{*_2}$.

- **Caso base:** Sea $(y,z) \in Id$, entonces por definición e hipótesis tenemos que $(x,z) \in R^{*_2}$.
- **Hipótesis de inducción:** Sea $x_0 \in A$ tal que si $(x, x_0) \in R^{*_2}$, entonces $(x,y) \in R^{*_2}$.
- **Paso inductivo:** Sean $(x, x_0) \in R^{*_2}$, $(x_0, y) \in R^{*_2}$ y $(y,z) \in R$, por demostrar que $(x,z) \in R^{*_2}$.

  Por hipótesis e hipótesis de inducción se tiene que $(x,y) \in R^{*_2}$ y dado que $(y,z) \in R$, entonces por la regla $crt2_2$ se concluye que $(x,z) \in R^{*_2}$.

$\boxed{\lambda}$

Con este resultado, demostrar la equivalencia de las dos definiciones de la cerradura reflexiva-transitiva y la Ecuación 2.2 (en su versión de juicios) resulta sencillo.

Teorema 2.4. *Sean A un conjunto y R una relación sobre A, entonces se cumple la siguiente igualdad:*

$$R^* = R^{*_2}$$

Demostración. En cada inclusión se utiliza la inducción sobre la cerradura que se encuentre en la hipótesis y el hecho de que ambas cerraduras son transitivas. $\boxed{\lambda}$

Corolario 2.4. *Sean A un conjunto, R, S dos relaciones sobre A, entonces:*

$$S \circ R^* = ci(S, R)$$

Demostración. Por el Teorema 2.4 basta demostrar que $S \circ R^{*2} = ci(S, R)$. Cuya demostración es análoga a la presentada en el Lema 2.9. $\boxed{\lambda}$

Otra cerradura que se utiliza más adelante es R^+, a continuación damos su definición.

Definición 2.19. *Sean A un conjunto y R una relación sobre A, se define la cerradura transitiva, denotada R^+, como:*

$$R^+ = R \circ R^*$$

Por el Lema 2.9 y el Corolario 2.4 se tiene que $R^* \circ R = R \circ R^*$, de manera que la cerradura transitiva también la definimos de la siguiente forma:

Lema 2.11. *Sean A un conjunto y R una relación sobre A, entonces se cumple la igualdad:*

$$R^+ = R^* \circ R$$

Demostración. Se sigue del Teorema 2.4 haciendo un análisis de casos sobre las reglas de ambas definiciones. $\boxed{\lambda}$

A continuación definimos una cerradura especifica para nuestros propósitos, la cual permita calcular el dominio de la composición de varias relaciones. En otras palabras, es necesario una cerradura específica para generalizar el Teorema 2.2, Yuta Ikeda y Susumu Nishimura(2011) en [16] lo explican de la siguiente manera:

> «(...) *Exista un camino a través de las composiciones de las S_i hasta llegar a la relación R, verificar que el par pertenezca a R y poder regresar por las mismas S_i.* »(p. 5)

Donde *camino* indica la composiciones de relaciones S_i y el regreso sobre este camino es la composición de las inversas de cada S_i.

2.3.3. La cerradura simétrica

Yuta Ikeda y Susumu Nishimura atacan el problema anterior utilizando los puntos fijos dados por el teorema de Knaster-Tarski, definiendo la siguiente cerradura [16].

Definición 2.20. *Sean A un conjunto, $R, S_1, \ldots, S_n$ relaciones sobre A. La cerradura simétrica de R respecto a la lista $[S_1, \ldots, S_n]$, denotada $(R)[S_1, \ldots, S_n]^*$, se define de la siguiente forma:*

$$(R)[S_1, \ldots, S_n]^* = \mu X.(R \cup (\bigcup_{i=1}^{n}(S_i^{\smile} \circ X \circ S_i)))$$

con $n \geq 1$.

El fin de la cerradura simétrica[1] es poder realizar cero o más pasos de navegación (composiciones) utilizando las S_i, transformar la entrada por medio de R y finalmente regresar a la posición original a través de la composición de las inversas [16].

Por otra parte, la definición actual de la cerradura simétrica tiene el inconveniente de depender de una unión generalizada, la expresión $(\bigcup_{i=1}^{n}(S_i^{\smile} \circ X \circ S_i))$. Dar una definición directa de esta operación es difícil desde una perspectiva puramente sintáctica. Para evitar este problema utilizamos la siguiente observación:
Si $(x, y) \in \bigcup_{i=1}^{n}(S_i^{\smile} \circ X \circ S_i)$, entonces para alguna i tal que $1 \leq i \leq n$, se cumple que $(x, y) \in S_i^{\smile} \circ X \circ S_i$. Por lo que considerando la lista $\ell = [S_1, \ldots, S_n]$, esto se traduce a que dada una relación S_i que pertenece a la lista ℓ y se cumple que $(x, y) \in S_i^{\smile} \circ X \circ S_i$, entonces $(x, y) \in \bigcup_{i=1}^{n}(S_i^{\smile} \circ X \circ S_i)$. Esta forma de parafrasear la definición permite definir la cerradura simétrica a un sistema de juicios.

Definición 2.21. *Sean A un conjunto y $R, S_1, \ldots, S_n$ relaciones sobre A. La cerradura simétrica de R respecto a la lista $[S_1, \ldots, S_n]$ la definimos con el siguiente sistema de juicios:*

$$\frac{(x, y) \in R}{(x, y) \in (R)[S_1, \ldots, S_n]^*}\, cs_1$$

$$\frac{S_i \in [S_1, \ldots, S_n] \qquad (x, y) \in S_i^{\smile} \qquad (y, z) \in (R)[S_1, \ldots, S_n]^* \qquad (z, w) \in S_i}{(x, w) \in (R)[S_1, \ldots, S_n]^*}\, cs_2$$

Análogamente que las cerraduras anteriores, esta cerradura también tiene un principio de inducción.

Definición 2.22 (Principio de inducción para la cerradura simétrica)**.** *Para cualesquiera A conjunto, $R, S_1, \ldots, S_n$ relaciones binarias sobre A y $\mathcal{P}$ un juicio binario sobre elementos de A, si se cumplen:*

- *Para cualesquiera $x, y \in A$, si $(x, y) \in R$, entonces (x, y) $\mathcal{P}$.*

[1] No hay que confundir esta cerradura simétrica con la cerradura simétrica «clásica», es decir, la cerradura $sym(R) = R \cup R^{\smile}$.

- *Para cualesquiera $x, y, z, w \in A$ y S_i relación binaria sobre A, si $S_i \in [S_1, \ldots, S_n]$, $(x, y) \in S_i^{\smile}$, $(y, z) \in (R)[S_1, \ldots, S_n]^*$, $(y, z)\ \mathcal{P}$ y $(z, w) \in S_i$, entonces $(x, w)\ \mathcal{P}$.*

Entonces para cualesquiera $y, y_0 \in A$, si $(y, y_0) \in (R)[S_1, \ldots, S_n]^*$ entonces $(y, y_0)\ \mathcal{P}$.

Anteriormente establecimos que una relación correflexiva es la forma de representar un conjunto y una relación inyectiva es aquella que al componer su inversa y la original resulta ser la relación identidad. Considerando una relación correflexiva R y una lista de relaciones inyectivas $[S_1, \ldots, S_n]$, como todo elemento de R está en Id y la cerradura simétrica compone las $S_i^{\smile}$ con S_i, entonces resulta que $(R)[S_1, \ldots, S_n]^*$ es correflexiva.

Lema 2.12. *Sean A un conjunto, $R, S_1, \ldots, S_n$ relaciones binarias sobre A. Si R es correflexiva y $S_1, \ldots, S_n$ son inyectivas, entonces $(R)[S_1, \ldots, S_n]^*$ es correflexiva.*

Demostración. Sean $R, S_1, \ldots, S_n$ relaciones binarias sobre A tales que R es correflexiva y $S_1, \ldots, S_n$ son inyectivas, por demostrar que $(R)[S_1, \ldots, S_n]^*$ es correflexiva.
Sean $(x, y) \in (R)[S_1, \ldots, S_n]^*$ por demostrar que $x = y$, se procede por inducción sobre la estructura de la cerradura simétrica:

- **Caso base:** Sea $(x, y) \in R$, dado que R es correflexiva se concluye que $x = y$.
- **Hipótesis de inducción:** Sea $z \in A$ tal que si $(y, z) \in (R)[S_1, \ldots, S_n]^*$, entonces $y = z$.
- **Paso inductivo:** Sean $w \in A$ y S_i una relación binaria tales que $S_i \in [S_1, \ldots, S_n]$, $(x, y) \in S_i^{\smile}$, $(y, z) \in (R)[S_1, \ldots, S_n]^*$ y $(z, w) \in S_i$, por demostrar que $x = w$.
  Por hipótesis de inducción se tiene que $(x, z) \in S_i^{\smile}$, entonces $(x, w) \in S_i^{\smile} \circ S_i$ finalmente por hipótesis se tiene que $S_i \in [S_1, \ldots, S_n]$ por lo que S_i es inyectiva. Concluyendo que $x = w$.

$\boxed{\lambda}$

El fin de esta cerradura es generalizar el Teorema 2.2, para obtener este resultado es necesario el siguiente resultado de la cerradura simétrica.

Lema 2.13. *Sean A un conjunto, $R, S_1, \ldots, S_n$ relaciones binarias sobre A tales que R es correflexiva y $S_1, \ldots, S_n$ son inyectivas, entonces se cumple la siguiente ecuación:*

$$\Pi \circ (R)[S_1, \ldots, S_n]^* = \Pi \circ R \circ \left(\bigcup_{i=1}^{n} S_i \right)^*$$

Demostración. Sean $R, S_1, \ldots, S_n$ relaciones binarias sobre A tales que R es correflexiva y $S_1, \ldots, S_n$ son inyectivas, por demostrar:

- $\Pi \circ (R)[S_1, \ldots, S_n]^* \subseteq \Pi \circ R \circ (\bigcup_{i=1}^{n} S_i)^*$

  Sea $(x, y) \in \Pi \circ (R)[S_1, \ldots, S_n]^*$, por definición existe un $x_0 \in A$ tal que $(x, x_0) \in \Pi$ y $(x_0, y) \in (R)[S_1, \ldots, S_n]^*$.

  Considerando a $U = \bigcup_{i=1}^{n} S_i$, veamos que $(x_0, y) \in (U^*)^{\smile} \circ R \circ U^*$; para demostrar este hecho se procede por inducción sobre $(x_0, y) \in (R)[S_1, \ldots, S_n]^*$.

  - **Caso base:** Sea $(x_0, y) \in R$, por la reflexividad de la igualdad y la regla crt_1 se tiene que $(x_0, x_0) \in U^*$ y $(y, y) \in U^*$.

    Entonces por definición de la composición, concluimos que $(x_0, y) \in (U^*)^{\smile} \circ R \circ U^*$.
  - **Hipótesis de inducción:** Sea $z \in A$ tal que $(y, z) \in (U^*)^{\smile} \circ R \circ U^*$.
  - **Paso inductivo:** Sean $w \in A$ y S_i una relación binaria sobre A tales que $S_i \in [S_1, \ldots, S_n]$,

    $(x_0, y) \in S_i^{\smile}$, $(y, z) \in (R)[S_1, \ldots, S_n]^*$ y $(z, w) \in S_i$.

    Por hipótesis de inducción y definición de composición se tiene que existen $x_1, x_2 \in A$ tales que $(y, x_2) \in (U^{\smile})^*$, $(x_2, x_1) \in R$ y $(x_1, z) \in U^*$, por demostrar que

    $(x_0, w) \in (U^*)^{\smile} \circ R \circ U^*$.

    Como $(x_0, y) \in S_i^{\smile}$ y $S_i^{\smile} \subseteq U^{\smile}$, entonces $(x_0, y) \in U^{\smile}$. Dado que $(y, x_2) \in (U^{\smile})^*$, entonces la regla crt_2 tenemos que $(x_0, x_2) \in (U^{\smile})^*$.

    Análogo al argumento anterior, dado que $(z, w) \in S_i$ por el Teorema 2.4 concluimos que $(x_1, w) \in U^*$.

  Por lo que $(x_0, y) \in (U^*)^{\smile} \circ R \circ U^*$, entonces por definición de la relación total e hipótesis se cumple que $(x_2, y) \in \Pi \circ R \circ U^*$.

- $\Pi \circ R \circ (\bigcup_{i=1}^{n} S_i)^* \subseteq \Pi \circ (R)[S_1, \ldots, S_n]^*$

  Considerando a $U = \bigcup_{i=1}^{n} S_i$, por el Corolario 2.4 se tiene que $\Pi \circ R \circ U^* = ci(\Pi \circ R, U)$.

  Sea $(x, y) \in ci(\Pi \circ R, U)$, por demostrar que $(x, y) \in \Pi \circ (R)[S_1, \ldots, S_n]^*$. Se procede por inducción sobre $(x, y) \in ci(\Pi \circ R, U)$.

- **Caso base:** Sea $(x, y) \in \Pi \circ R$, entonces por definición existe un $x_0 \in A$ tal que $(x, x_0) \in \Pi$ y $(x_0, y) \in R$. Por la regla cs_1 se tiene que $(x_0, y) \in (R)[S_1, \ldots, S_n]^*$, entonces $(x, y) \in \Pi \circ (R)[S_1, \ldots, S_n]^*$.
- **Hipótesis de inducción:** Supongamos que $(x, y) \in \Pi \circ (R)[S_1, \ldots, S_n]^*$.
- **Paso inductivo:** Sea $z \in A$ tal que $(x, y) \in ci(\Pi \circ R, U)$ y $(y, z) \in U$, por demostrar que $(x, z) \in \Pi \circ (R)[S_1, \ldots, S_n]^*$.

  Por definición existe un S_i tal que $Si \subseteq U$ y $(y, z) \in S_i$, entonces por hipótesis se concluye que $(x, z) \in \Pi \circ (R)[S_1, \ldots, S_n]^* \circ S_i$.

  Por el Corolario 2.1 se tiene que $(x, z) \in \Pi \circ dom((R)[S_1, \ldots, S_n]^* \circ S_i)$, como S_i es inyectiva entonces por el Teorema 2.2 se tiene que $(x, z) \in \Pi \circ S_i^{\smile} \circ dom((R)[S_1, \ldots, S_n]^*) \circ S_i$.

  Dado que R es correflexiva y $S_1, \ldots, S_n$ son inyectivas por el Lema 2.12 tenemos que $(R)[S_1, \ldots, S_n]^*$ es correflexiva, entonces por el Corolario 2.2 se tiene que
  $(x, z) \in \Pi \circ S_i^{\smile} \circ (R)[S_1, \ldots, S_n]^* \circ S_i$.

  Por definición existen $x_1, x_2, x_3 \in A$ tales que $(x, x_1) \in \Pi$, $(x_1, x_2) \in S_i^{\smile}$,
  $(x_2, x_3) \in (R)[S_1, \ldots, S_n]^*$ y $(x_3, z) \in S_i$; dado que $S_i \subseteq U$ es claro que $S_i \in [S_1, \ldots, S_n]$, entonces por la regla cs_2, se tiene que $(x_1, z) \in (R)[S_1, \ldots, S_n]^*$.

$\boxed{\lambda}$

El último ingrediente para obtener la generalización del Teorema 2.2 es un resultado de las conexiones de Galois. En la teoría del orden una conexión de Galois es una equivalencia de órdenes de acuerdo a dos funciones monótonas.

Definición 2.23 (Conexión de Galois)**.** *Sean $(A, \leq_A)$ y $(B, \leq_B)$ dos conjuntos parcialmente ordenados (COPO). Una conexión de Galois monótona entre estos COPO consiste en dos funciones monótonas $F : A \to B$ y $G : B \to A$, tales que para cualesquiera $a \in A$ y $b \in B$, se tiene que:*

$$F(a) \leq_B b \text{ si y sólo si } a \leq_A G(b)$$

El dominio y la composición resultan en una conexión de Galois, permitiendo hacer una «traducción» entre dichos operadores.

Teorema 2.5. *Sean A un conjunto y R, S relaciones binarias sobre A, tal que R es correflexiva. Entonces es válida la siguiente conexión de Galois:*

$$dom(R) \subseteq S \text{ si y sólo si } R \subseteq \Pi \circ S$$

Es decir, considerando a $\leq_A$ como $\subseteq$, $F(R) = dom(R)$ y $G(R) = \Pi \circ R$, se debe cumplir la conexión de Galois entre las funciones $dom(R)$ y $\Pi \circ R$.

Demostración. Se sigue de las definiciones de dominio y composición de relaciones. $\boxed{\lambda}$

Considerando a $S = dom(R)$, tenemos el siguiente resultado:

Corolario 2.5. *Sean A un conjunto y R una relación binaria sobre A, entonces se cumple lo siguiente:*

$$R \subseteq \Pi \circ (dom(R))$$

Demostración. Se sigue de la Conexión de Galois 2.5. $\boxed{\lambda}$

Finalmente, se procede al resultado tan esperado.

Teorema 2.6. *Sean A un conjunto y $R, S_1, \ldots, S_n$ relaciones binarias sobre A, tales que $S_1, \ldots, S_n$ son inyectivas. Entonces considerando a $U = \bigcup_{i=1}^{n} S_i$, se tiene la siguiente igualdad:*

$$dom(R \circ U^*) = (dom(R))[S_1, \ldots, S_n]^*$$

Demostración. Sean $R, S_1, \ldots, S_n$ relaciones binarias sobre A tales que $S_1, \ldots, S_n$ son inyectivas, por demostrar:

- $dom(R \circ U^*) \subseteq (dom(R))[S_1, \ldots, S_n]^*$

  Como el dominio es coreflexivo y dado que $S_1, \ldots, S_n$ son inyectivas, entonces se tiene que $(dom(R))[S_1, \ldots, S_n]^*$ es coreflexivo, por la Conexión de Galois 2.5 la inclusión a demostrar es equivalente a demostrar que $R \circ U^* \subseteq \Pi \circ (dom(R))[S_1, \ldots, S_n]^*$.

  Por hipótesis y el Lema 2.13, es equivalente a demostrar que $R \circ U^* \subseteq \Pi \circ dom(R) \circ U^*$.

  Entonces sea $(x, y) \in R \circ U^*$, basta ver que $(x, y) \in \Pi \circ dom(R) \circ U^*$.

  Por definición de composición existe $x_0 \in A$ tal que $(x, x_0) \in R$ y $(x_0, y) \in U^*$ y por el Corolario 2.5 concluimos que $(x, x_0) \in \Pi \circ dom(R)$.

- $(dom(R))[S_1, \ldots, S_n]^* \subseteq dom(R \circ U^*)$

  Sea $(x, y) \in (dom(R))[S_1, \ldots, S_n]^*$, por demostrar que $(x, y) \in dom(R \circ U^*)$.

  Se procede por inducción sobre $(x, y) \in (dom(R))[S_1, \ldots, S_n]^*$.

  - **Caso base:** Sea $(x, y) \in dom(R)$, por definición de dominio tenemos que $(x, y) \in Id$ y $(x, y) \in R^{\smile} \circ R$.

Por hipótesis y la inversa de la composición basta demostrar que
$(x, y) \in (U^{*\smile} \circ R^{\smile}) \circ (R \circ U^*)$.
Por definición de composición existe un $x_0 \in A$ tal que $(x, x_0) \in R^{\smile}$ y $(x_0, y) \in R$, donde por la reflexividad de la igualdad tenemos que $(x, x) \in U^{*\smile}$, por lo que $(x, x_0) \in U^{*\smile} \circ R^{\smile}$.
Análogamente se demuestra que $(x_0, y) \in R \circ U^*$.

- **Hipótesis de inducción:** Sea $z \in A$ tal que $(y, z) \in dom(R \circ U^*)$.
- **Paso inductivo:** Sean $w \in A$ y S_i una relación binaria sobre A tales que $S_i \in [S_1, \ldots, S_n]$,
  $(x, y) \in S_i^{\smile}$, $(y, z) \in (dom(R))[S_1, \ldots, S_n]^*$ y $(z, w) \in S_i$, por demostrar que
  $(x, w) \in dom(R \circ U^*)$.
  Por hipótesis de inducción y definición de dominio se tiene que $(y, z) \in Id$ y
  $(y, z) \in (U^{*\smile} \circ R^{\smile}) \circ (R \circ U^*)$.
  Como S_i es inyectiva, entonces para demostrar que $(x, w) \in Id$ basta demostrar que $(x, w) \in S_i^{\smile} \circ S_i$; lo cual es directo de las hipótesis.
  Ahora por definición de composición existe $x_1 \in A$ tal que $(y, x_1) \in U^{*\smile} \circ R^{\smile}$ y
  $(x_1, z) \in R \circ U^*$, por demostrar que $(x, x_1) \in (U^{*\smile} \circ R^{\smile})$ y que $(x_1, w) \in R \circ U^*$.
  Por definición de composición e inversa, existe un $x_2 \in A$ tal que $(x_2, y) \in U^*$ y
  $(x_1, x_2) \in R$, entonces por hipótesis se tiene que $(y, x) \in S_i$ y dado que $S_i \subseteq U$, se tiene que $(y, x) \in U$.
  Dado que $U^* = U^{*2}$, por la regla $crt2_2$ se tiene que $(x_2, x) \in U^*$, es decir, $(x, x_2) \in U^{*\smile}$.
  Concluyendo así que $(x, x_1) \in (U^{*\smile} \circ R^{\smile})$. Análogamente se concluye que $(x, x_1) \in (U^{*\smile} \circ R^{\smile})$.

$\boxed{\lambda}$

Con la cerradura simétrica y el uso de relaciones inyectivas generalizamos el caso de la cerradura reflexiva-transitiva, utilizando el Teorema 2.2. Este resultado permitirá obtener expresiones equivalentes del lenguaje xPath.

Solamente falta determinar cuándo un par de elementos no pertenecen a una relación binaria que simula un conjunto, esto servirá para la interpretación del operador de negación *not* del lenguaje xPath. Para lograr esta meta, en la siguiente sección definimos un nuevo tipo de relación binaria llamada relación negada y propiedades sobre este tipo de relaciones binarias.

2.4. Relaciones negadas

La definición del complemento de una relación binaria es [29]:

Definición 2.24 (Complemento)**.** *Sean A un conjunto y R una relación binaria sobre A se define la relación complemento, denotada R^{-}, como el siguiente conjunto:*

$$R^{-} = \{(x, y) \in A \times A | (x, y) \notin R\}$$

Esta definición indica todos los pares de elementos x, y que no pertenezcan a la relación pero sean elementos del conjunto A. Entonces considerar el complemento de relaciones correflexivas da elementos que no pertenecen a un «conjunto» pero es necesario mantener los elementos que no fueron filtrados, para esto se define la negación de una relación binaria de la siguiente forma [16]:

Definición 2.25 (Relación negada)**.** *Sean A un conjunto y R una relación binaria sobre A se define la relación negada, denotada $\neg R$, de la siguiente forma:*

$$\neg R = Id - R$$

Donde $-$ denota al operador de diferencia entre relaciones binarias.

Por definición la negación negada es correflexiva.

Proposición 2.2. *Sean A un conjunto y R una relación binaria sobre A, entonces $\neg R$ es correflexiva.*

Demostración. Se sigue de la definición de relación negada. $\boxed{\lambda}$

Una propiedad que salta a la vista es que la negación de una relación no es monótona.

Proposición 2.3. *Sean A un conjunto y R una relación binaria correflexiva sobre A, $\neg R$ es antimonótona, es decir, que para cualquier relación binaria S sobre A se satisface:*

$$R \subseteq S \text{ si y sólo si } \neg S \subseteq \neg R$$

Demostración. Sean R, S relaciones binarias sobre A, tal que R es correflexiva, por demostrar ambas implicaciones.

- Se supondrá que $R \subseteq S$, por demostrar que $\neg S \subseteq \neg R$.
  Sea $(x, y) \in \neg S$, por definición de relación negada se tiene que $(x, y) \in Id$ y $(x, y) \notin S$, con esto basta demostrar que $(x, y) \notin R$. Supongamos que $(x, y) \in R$, por hipótesis se tiene que $(x, y) \in S$ resultando en una contradicción. Por lo que $(x, y) \in \neg R$.

- Se supondrá que $\neg S \subseteq \neg R$, por demostrar que $R \subseteq S$.

  Sea $(x, y) \in R$ y supongamos que $(x, y) \notin S$. Dado que R es correflexiva, por hipótesis se tiene que $(x, y) \in \neg S$. Lo cual implica que $(x, y) \in \neg R$, cuya definición indica que $(x, y) \notin R$ causando una contradicción. Por lo que $(x, y) \in S$.

$\boxed{\lambda}$

Algunos resultados que cumple la negación de una relación tienen cierto parecido a propiedades sobre el complemento de relaciones binarias, los cuales se colectan en los siguientes lemas.

Lema 2.14. *Sean A un conjunto, considerando las relaciones vacía e identidad sobre el conjunto A, las siguientes ecuaciones son válidas:*

$$\neg\emptyset = Id \ y \ \neg Id = \emptyset$$

Demostración. Se sigue de la definición de relación negada. $\boxed{\lambda}$

Lema 2.15. *Sean A un conjunto y R una relación binaria correflexiva sobre A, se satisface:*

$$\neg\neg R = R$$

Demostración. Sea R una relación binaria correflexiva sobre A, por demostrar ambas inclusiones:

- $\neg\neg R \subseteq R$

  Sea $(x, y) \in \neg\neg R$, por definición se tiene que $(x, y) \in Id$ y $(x, y) \notin \neg R$, nuevamente por definición se tiene que $(x, y) \notin Id$ o bien $(x, y) \in R$.

  Haciendo un análisis de casos sobre las últimas dos hipótesis se concluye que $(x, y) \in R$.

- $R \subseteq \neg\neg R$

  Sea $(x, y) \in R$, por hipótesis R es correflexiva entonces $(x, y) \in Id$. Basta demostrar que $(x, y) \notin \neg R$.

  Supongamos que $(x, y) \in \neg R$, entonces por definición se tiene que $(x, y) \notin R$ lo cual contradice a la hipótesis.

$\boxed{\lambda}$

Lema 2.16. *Sean A un conjunto y R, S relaciones binarias sobre A, se satisfacen las leyes de De Morgan:*

$$\neg(R \cup S) = \neg R \cap \neg S$$
$$\neg(R \cap S) = \neg R \cup \neg S$$

Demostración. Se sigue de la definición de relación negada y las leyes de De Morgan aplicadas a los operadores lógicos. $\boxed{\lambda}$

Lema 2.17. *Sean A un conjunto y R una relación binaria sobre A, se satisfacen las leyes:*

$$\neg R \cup R = Id \text{ si } R \text{ es correflexiva.}$$
$$\neg R \cap R = \emptyset$$

Demostración. Se sigue de la definición de relación negada, utilizando ley del tercer excluido y el principio de contradicción. $\boxed{\lambda}$

2.4.1. El dominio negado

El dominio de una relación binaria R_f es simular el dominio de la función f. Por lo que, el dominio negado de R_f es la relación binaria que simula el conjunto de los elementos ajenos al dominio de la función f.

La ecuación mostrada en el Teorema 2.2, establece que el dominio de una composición de funciones tiene una expresión más acorde a la idea de navegar a través de la composición de relaciones binarias. Entonces la negación del dominio de una composición de funciones indica que un par x, y no pertenecen al dominio de una función o no pertenecen al dominio de la otra función compuesta. Para demostrar esto primero demostremos que la diferencia y unión de relaciones binarias son una conexión de Galois.

Teorema 2.7. *Sean A un conjunto y R, S, T relaciones binarias sobre A, se satisface la siguiente conexión de Galois:*

$$R - S \subseteq T \text{ si y sólo si } R \subseteq S \cup T$$

Demostración. Se sigue de las definiciones de diferencia y unión. $\boxed{\lambda}$

En el Teorema 2.2, no es suficiente la condición de que la relación con que se compone sea inyectiva, ya que, no habría forma de realizar la composición. Para esto también es necesario que la relación sea simple y permita un «regreso» en la composición.

Lema 2.18. *Sean A un conjunto y R, S relaciones binarias sobre A tal que S es inyectiva y simple, entonces se cumple la siguiente ecuación:*

$$\neg dom(R \circ S) = \neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$$

Demostración. Sean R, S relaciones binarias sobre A tal que S es inyectiva y simple, por demostrar:

- $\neg dom(R \circ S) \subseteq \neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$

  Por la Conexión de Galois 2.7 esto equivale a que se demuestre la inclusión:

  $Id \subseteq dom(R \circ S) \cup (\neg dom(S)) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$.

  Por la asociatividad y conmutatividad de la unión, utilizando nuevamente la Conexión de Galois 2.7 y la definición de relación negada, lo anterior es equivalente a demostrar:

  $\neg\neg dom(S) \subseteq dom(R \circ S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$.

  Como el dominio de cualquier relación es coreflexivo por el Lema 2.3 y dado que la doble negación de una relación es ella misma por el Lema 2.15, entonces se tiene que lo anterior es equivalente a demostrar que:

  $dom(S) \subseteq dom(R \circ S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$.

  Dado que S es inyectiva, entonces el dominio de la unión de composiciones por el Teorema 2.2 resulta en el siguiente expresión a demostrar:

  $dom(S) \subseteq (S^{\smile} \circ dom(R) \circ S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$

  Por las leyes de distribución de la unión y la intersección se tiene que demostrar:

  $dom(S) \subseteq (S^{\smile} \circ (dom(R) \cup \neg dom(R)) \circ S)$

  Dado que el dominio es correflexivo entonces por el Lema 2.17, se tiene que demostrar que:

  $dom(S) \subseteq (S^{\smile} \circ Id \circ S)$

  Finalizando en la siguiente expresión a demostrar por la neutralidad de la relación identidad respecto a la composición y desglosando la definición de dominio:

  $Id \cap (S^{\smile} \circ S) \subseteq (S^{\smile} \circ S)$

  Por lo que la inclusión queda demostrada, ya que S es inyectiva por hipótesis.

- $\neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S) \subseteq \neg dom(R \circ S)$

  Sean $(x, y) \in \neg dom(S) \cup ((\neg dom(R)) \circ S)$, se hace un análisis de casos sobre la unión.

  - Si $(x, y) \in \neg dom(S)$, por definición $(x, y) \in Id$ y $(x, y) \notin dom(S)$.

    Con esto basta demostrar que $(x, y) \notin dom(R \circ S)$, entonces se supondrá que

$(x, y) \in dom(R \circ S)$. Dado que S es inyectiva, por el Teorema 2.2, se tiene que $(x, y) \in S^{\smile} \circ dom(R) \circ S$.

Entonces por definición de composición existen $x_0, x_1 \in A$ tales que $(x, x_1) \in S^{\smile}$, $(x_1, x_0) \in dom(R)$ y $(x_0, y) \in S$, dado que el dominio es correflexivo esto implica que $(x, x_1) \in S^{\smile}$ y $(x_1, y) \in S$. Por lo que $(x, y) \in dom(R \circ S)$, teniendo una contradicción.

- Si $(x, y) \in S^{\smile} \circ (\neg dom(R)) \circ S$

  Para demostrar que $(x, y) \in \neg dom(R \circ S)$, basta ver que
  $S^{\smile} \circ (\neg dom(R)) \circ S \subseteq \neg dom(R \circ S)$.
  Por la Conexión de Galois 2.7 y la doble negación esto equivale a demostrar que
  $Id \subseteq \neg(S^{\smile} \circ (\neg dom(R)) \circ S) \cup (\neg dom(R \circ S))$.
  Como S es inyectiva entonces por el Teorema 2.2, la expresión anterior es equivalente a demostrar:
  $Id \subseteq \neg(S^{\smile} \circ (\neg dom(R)) \circ S) \cup \neg(S^{\smile} \circ dom(R) \circ S)$.
  Por leyes de De Morgan y la negación de la relación vacía, equivale a demostrar:
  $\neg\emptyset \subseteq \neg((S^{\smile} \circ (\neg dom(R)) \circ S) \cap (S^{\smile} \circ dom(R) \circ S))$.
  Es fácil de observar que la negación de la relación vacía es correflexiva, entonces por la no monotonía de la negación de relaciones lo anterior es equivalente a demostrar:
  $\neg\neg((S^{\smile} \circ (\neg dom(R)) \circ S) \cap (S^{\smile} \circ dom(R) \circ S)) \subseteq \neg\neg\emptyset$.
  Por el Teorema 2.2, el hecho de que el dominio y la negación de una relación son relaciones correflexivas, entonces por el Lema 2.15 es equivalente a demostrar:
  $(S^{\smile} \circ (\neg dom(R)) \circ S) \cap (S^{\smile} \circ dom(R) \circ S) \subseteq \emptyset$.
  Entonces sean $x, y \in A$ tales que $(x, y) \in (S^{\smile} \circ (\neg dom(R)) \circ S) \cap (S^{\smile} \circ dom(R) \circ S)$, utilizando la ley modular se tiene que $(x, y) \in ((S^{\smile} \circ (\neg dom(R))) \cap (S^{\smile} \circ dom(R) \circ S)) \circ S$.
  Por la definición de la composición existe $x_0 \in A$ tal que
  $(x, x_0) \in (S^{\smile} \circ (\neg dom(R))) \cap (S^{\smile} \circ dom(R) \circ S)$ y $(x_0, y) \in S$, entonces por propiedades de la inversa se tiene que $(x_0, x) \in ((\neg dom(R)) \circ S^{\smile\smile}) \cap ((S^{\smile} \circ dom(R) \circ S \circ S^{\smile})^{\smile})$.
  Nuevamente por la ley modular se tiene que
  $(x_0, x) \in ((\neg dom(R))^{\smile} \cap (S^{\smile} \circ dom(R) \circ S \circ S^{\smile\smile} \circ S^{\smile\smile\smile})) \circ S^{\smile\smile}$.
  Por definición de la composición existe $x_1 \in A$ tal que
  $(x_0, x_1) \in (\neg dom(R))^{\smile} \cap (S^{\smile} \circ dom(R) \circ S \circ S^{\smile\smile} \circ S^{\smile\smile\smile})$ y $(x_1, x) \in S$.

Entonces por propiedades de la inversa se tiene que

$(x_0, x_1) \in (\neg dom(R))^{\smile\smile} \cap (S^{\smile} \circ dom(R) \circ S \circ S^{\smile\smile} \circ S^{\smile\smile\smile\smile})$.

Que por definición de intersección y propiedades de la inversa se tiene que

$(x_0, x_1) \in \neg dom(R)$ y $(x_0, x_1) \in S \circ S^{\smile} \circ dom(R) \circ S \circ S^{\smile}$.

Como S es simple, se concluye que $(x_0, x_1) \in dom(R)$ lo cual es una contradicción.

Por lo que si está contenido en la relación vacía, terminando así esta parte de la demostración.

$\boxed{\lambda}$

Si una relación es correflexiva, entonces es simple e inyectiva. De manera que el Lema 2.18 lo generalizamos al siguiente resultado.

Teorema 2.8. *Sean A un conjunto y R, S relaciones binarias sobre A tal que S es correflexiva, entonces la siguiente ecuación es válida:*

$$\neg dom(R \circ S) = (\neg S) \cup ((\neg dom(R)) \circ S)$$

Demostración. Sean R, S relaciones binarias sobre A tal que S es correflexiva, por el hecho anterior se tiene que S es simple e inyectiva.
Entonces basta demostrar la ecuación $\neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S) = (\neg S) \cup ((\neg dom(R)) \circ S)$, por demostrar:

- $\neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S) \subseteq (\neg S) \cup ((\neg dom(R)) \circ S)$

  Sean $(x, y) \in \neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$, se realiza un análisis de casos sobre la unión:

  - Si $(x, y) \in \neg dom(S)$, por definición $(x, y) \in Id$ y $(x, y) \notin dom(S)$, supongamos que $(x, y) \in S$.

    Se tiene que $(x, y) \in S^{\smile} \circ S$, ya que por hipótesis $x = y$ y $(x, y) \in S$, entonces $(x, y) \in dom(S)$ lo cual es una contradicción; esto implica que $(x, y) \notin S$. Por lo tanto $(x, y) \in \neg S$.

  - Si $(x, y) \in S^{\smile} \circ (\neg dom(R)) \circ S$, por definición de composición existe un $x_0 \in A$ tal que $(x, x_0) \in S^{\smile}$ y $(x_0, y) \in (\neg dom(R)) \circ S$.

    Demostremos que $(x, y) \in (\neg dom(R)) \circ S$, por la correflexividad de S se tiene que $(x, x_0) \in Id$ entonces $(x, y) \in (\neg dom(R)) \circ S$.

- $(\neg S) \cup ((\neg dom(R)) \circ S) \subseteq \neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$

  Sean $x, y \in A$ tales que $(x, y) \in (\neg S) \cup ((\neg dom(R)) \circ S)$, se realiza un análisis de casos sobre la unión:

  - Si $(x, y) \in \neg S$, por definición de la relación negada se tiene que $(x, y) \in Id$ y $(x, y) \notin S$. Demostremos que $(x, y) \in \neg dom(S)$, para esto basta ver que $(x, y) \notin dom(S)$.
    Supongamos que $(x, y) \in dom(S)$, entonces por definición de dominio $(x, y) \in S^{\smile} \circ S$, dado que S es correflexiva y por la definición de composición se tiene que $(x, y) \in S$, lo cual es una contradicción. Por lo que $(x, y) \in \neg dom(S) \cup (S^{\smile} \circ (\neg dom(R)) \circ S)$.
  - Si $(x, y) \in (\neg dom(R)) \circ S$, basta ver que $(x, y) \in S^{\smile} \circ (\neg dom(R)) \circ S$.
    Por definición de composición existe un $x_0 \in A$ tal que $(x, x_0) \in \neg dom(R)$ y $(x_0, y) \in S$, sin embargo por definición de la relación negada se tiene que $x = x_0$ y por la coreflexividad de S se tiene que $x_0 = y$, entonces $(x, x) \in S^{\smile}$.
    Por hipótesis concluimos que $(x, y) \in S^{\smile} \circ (\neg dom(R)) \circ S$.

$\boxed{\lambda}$

Con este resultado se finaliza el análisis sobre relaciones negadas, la observación principal es que este operador tiene un comportamiento parecido al complemento de relaciones binarias.

En la siguiente sección se dará un mecanismo que permite la construcción de estructuras infinitas, tales estructuras tienen que cumplir pertenecer a una relación generada por el mayor punto fijo de un sistema de juicios. Este mecanismo llamado *coinducción*, permite hacer un número de composiciones, sobre relaciones binarias, infinitas para generalizar la cerradura reflexiva-transitiva.

2.5. Coinducción

Las definiciones con características similares a la cerradura reflexiva-transitiva son llamadas definiciones inductivas, ya que, las hipótesis en reglas parecidas a crt_2 *inducen* un nuevo objeto, es decir, cada regla construye nuevos elementos. Lourdes González en [9] indica que este tipo de reglas son llamadas *constructores*, mientras que reglas parecidas a crt_1 generan elementos que son llamados *objetos básicos* dado que, son objetos de la definición inductiva cuya existencia se asume.

Si se debilita la condición de que las inferencias sean detenidas[1], es decir, no existan objetos básicos o bien se permita la existencia de inferencias infinitas, entonces este tipo de definiciones son llamadas coinductivas y serán introducidas en esta sección.

[1]Formalmente significa eliminar la condición de que el conjunto tenga un buen orden.

2.5.1. Tipos coinductivos

Un tipo definido inductivamente es aquel que tiene objetos básicos, elementos cuya existencia es asumida, y constructores que son funciones cuyo codominio es el tipo inductivo. Por ejemplo, las listas que contienen números naturales se definen inductivamente de la siguiente forma:

Definición 2.26 (Listas de números naturales)**.** *La definición inductiva de listas de números naturales está dada por las siguientes cláusulas:*

1. *La lista vacía, denotada* nil*, es una lista de números naturales.*
2. *Si x es un número natural y ℓ es una lista de números naturales, entonces $x :: \ell$ es una lista de números naturales. Donde x denota a la cabeza de la lista $x :: \ell$ y ℓ denota su cola.*
3. *Estas y sólo estas son listas de números naturales.*

En el sistema de juicios, la especificación anterior se formaliza de la siguiente manera:

$$\frac{}{\text{nil}}\, list_{nil} \qquad \frac{n \qquad \ell}{x :: \ell}\, list_{cons}$$

Entonces para construir una lista de números naturales ℓ, primero se construye la lista vacía y a partir de esta, se le anexa un número natural al inicio de cada nueva lista resultante del constructor.

Ejemplo 2.1. *La lista que almacena los números* 1 *y* 2 *tiene la siguiente construcción:*

- *Se construye la lista vacía* nil*.*
- *Se anexa el número* 2 *a la lista* nil*, es decir,* 2 ::nil*.*
- *Se anexa el número* 1 *a la lista* 2 ::nil*, es decir,* 1 :: 2 ::nil*.*

Como se indicó, se puede debilitar la condición de buen orden sobre las listas, teniendo dos posibles resultados:

1. Listas infinitas. Listas cuyo número de elementos es estrictamente infinito.
2. Listas potencialmente infinitas. Listas cuyo número de elementos puede o no ser infinito.

Este proceso de debilitar la condición de buen orden, es la razón por la cual se le nombró a este tipo de definiciones: *definiciones coinductivas.*

Analicemos el caso de las listas infinitas, cuyo nombre usual es *Streams* y la definición es la siguiente:

Definición 2.27 (Streams de números naturales)**.** *La definición de streams de números naturales está dada por las siguientes cláusulas:*

1. *Si x es un número natural y ℓ es un stream de números naturales, entonces $x :: \ell$ es un stream de números naturales. Dónde x denota a la cabeza de la lista $x :: \ell$ y ℓ denota la cola de la lista. Dónde x denota a la cabeza del stream $x :: \ell$ y ℓ denota la cola del stream.*

2. *Estos y sólo estos son streams de números naturales.*

En el sistema de juicios, la especificación anterior se formaliza de la siguiente manera:

$$\frac{n : N \qquad \ell : stream_N}{x :: \ell : stream_N}\, stream_{cons}$$

Obsérvese que la única regla disponible es $stream_{cons}$ y al no contar con objetos básicos, entonces no podemos construir streams como se construyen listas finitas. Esto no significa que esta definición sea vacua, es decir, se asumen los objetos infinitos. La interpretación correcta de estas definiciones se da mediante el mayor punto fijo, ya mencionado antes.

Por ejemplo, en el lenguaje de programación `HASKELL` el stream que consiste en números naturales y el stream que consiste en la repetición del número 1, los implementamos de la siguiente forma:

Código 2.1 Ejemplos de streams.

```
-- Stream de números naturales.
stream1 = [1,2..]


-- Stream con repeticiones del número 1.
stream2 = [1,1..]
```

Ahora, supongamos que tenemos un stream que consiste en la sucesión 1, 2, 3, 4, 5 y repeticiones del número 1; y otro stream que consiste en la sucesión 1, 2, 3 y repeticiones del número 1.

$$s_1 = [1, 2, 3, 4, 5, 1, 1, 1, \dots]$$
$$s_2 = [1, 2, 3, 1, 1, 1, \dots]$$

Observamos que el stream s_1 es distinto al stream s_2, ya que, solo los primeros tres elementos de estos streams, son iguales.

Para lograr el análisis anterior, tuvimos que destruir los stream s_1, s_2 en sus partes finitas y compararlas. Esta acción la logramos por medio de las siguientes funciones:

Código 2.2 Destructores del stream.

```
-- | head. Devuelve la cabeza de un stream.

head :: Stream a -> a

head (Cons x xs) = x


-- | tail. Devuelve la cola de un stream.

tail :: Stream a -> Stream a

tail (Cons x xs) = xs
```

Esta clase de funciones son conocidas como *destructores* y como su nombre lo indica, su finalidad es destruir un tipo coinductivo en sus partes finitas y/o infinitas. En el caso particular de los destructores `head` y `tail`, devuelven la cabeza y la cola de un stream, respectivamente.

Por lo que, para hacer el análisis comparativo de los stream s_1 y s_2 se logra con el siguiente programa:

Código 2.3 Comparar dos streams.

```
eqStream :: Eq a => Stream a -> Stream a -> Bool

eqStream (Cons x xs) (Cons y ys) = (x == y) && (eqStream xs ys)
```

Obsérvese que en la evaluación de la expresión `eqStream` s_1 s_2 el valor devuelto es `False`. Sin embargo, si los streams dados como parámetro a `eqStream` son parecidos por mucho que destruyamos la estructura, entonces solamente podemos asegurar que los stream son *equivalentes*.[1]

La definición formal de que dos stream son equivalentes, es la siguiente:

Definición 2.28. *Para cualquier tipo* `a` *y dos streams* s_1, s_2 *de tipo* `a`*, decimos que* s_1 *y* s_2 *son equivalentes, denotado* $s_1 \equiv s_2$*, si cumplen las siguientes condiciones:*

- *Sus cabezas son iguales.*
- *Sus colas son equivalentes.*

Por ejemplo, considerando las siguientes funciones:

[1] Es imposible decir que dos stream son iguales ya que, tendríamos que comparar infinitamente.

Código 2.4 Funciones sobre streams.

```
-- | iterate. Itera una función creando un stream.
iterate :: (a -> a) -> a -> Stream a
iterate f x = Cons x (iterate f (f x))


-- | map. Mapea una función sobre un stream.
map :: (a -> b) -> Stream a -> Stream b
map f (Cons x xs) = Cons (f x) (map f xs)
```

Tenemos los siguientes stream, que son resultado de las funciones anteriores:

$$\begin{aligned} \texttt{iterate}\ f\ (f\ x) &= [f\ x, f\ (f\ x), f\ (f\ (f\ x)), \ldots] \\ \texttt{map}\ f\ (\mathit{iterate}\ f\ x) &= \texttt{map}\ f\ [x, f\ x, f\ (f\ x)] \\ &= [f\ x, f\ (f\ x), f\ (f\ (f\ x)), \ldots] \end{aligned}$$

En principio, los stream mostrados son equivalentes. Esto lo podemos indicar formalmente asumiendo que, sintácticamente los streams tienen un comportamiento similar en la parte infinita y utilizando los destructores, demostrar que las cabezas son iguales y aún se mantiene el comportamiento en las colas de estos stream. Esta forma de razonamiento es conocida como *hipótesis de coinducción*, a continuación demostramos la aseveración anterior utilizando esta técnica.

Lema 2.19. *Para cualquier tipo* `a`*, función* $f : \texttt{a} \to \texttt{a}$ *y* x *elemento de tipo* `a`*, se cumple la siguiente ecuación:*

$$\texttt{iterate}\ f\ (f\ x) \equiv \texttt{map}\ f\ (\texttt{iterate}\ f\ x)$$

Demostración. Se asumirá que el comportamiento de los streams satisface el resultado, es decir, la hipótesis de coinducción establece que:

Para cualquier tipo `a`, función $f : \texttt{a} \to \texttt{a}$ y x elemento de tipo `a`, se cumple la siguiente ecuación:

$$\texttt{iterate}\ f\ (f\ x) \equiv \texttt{map}\ f\ (\texttt{iterate}\ f\ x)$$

Ahora considerando un tipo `a`, una función f y x elemento de tipo `a`, por demostrar que:
$\texttt{iterate}\ f\ (f\ x) \equiv \texttt{map}\ f\ (\texttt{iterate}\ f\ x)$ Hacemos un análisis de casos sobre la equivalencia de streams:

- Por demostrar que la cabeza de cada stream resultante son iguales.

  Por un lado se tiene que:

$$\texttt{head}\ (\texttt{iterate}\ f\ (f\ x)) = \texttt{head}\ (\texttt{Cons}\ (f\ x)\ (\texttt{iterate}\ f\ (f\ x)))$$
$$= f\ x$$

  Por otra parte se tiene que:

$$\texttt{head}\ (\texttt{map}\ f\ (\texttt{iterate}\ f\ x)) = \texttt{head}\ (\texttt{Cons}\ (f\ x)\ (\texttt{map}\ f(\texttt{iterate}\ f\ x)))$$
$$= f\ x$$

  Entonces se satisface este caso.

- Por demostrar que las colas resultantes son equivalentes.

  Por un lado se tiene que:

$$\texttt{tail}\ (\texttt{iterate}\ f\ (f\ x)) \equiv \texttt{iterate}\ f\ (f\ (f\ x))$$

  Por otra parte se tiene que:

$$\texttt{tail}\ (\texttt{map}\ f\ (\texttt{iterate}\ f\ x)) \equiv \texttt{map}\ f\ (\texttt{iterate}\ f\ (f\ x))$$

  Teniendo que $\texttt{iterate}\ f\ (f\ (f\ x)) \equiv \texttt{map}\ f\ (\texttt{iterate}\ f\ (f\ x))$ por la hipótesis de co-inducción.

$\boxed{\lambda}$

Este tipo de demostración puede parecer contraintuitiva por la hipótesis de coinducción, sin embargo, es el tipo de demostración que solicita `COQ`, dicha solicitud es para poder asegurar que sintácticamente se mantendrá la propiedad sobre la estructura, aunque esta sea infinita.

En la siguiente subsección se analiza la creación de relaciones con el mecanismo de coinducción.

2.5.2. Relaciones definidas coinductivamente

Las definiciones inductivas no se restringen a la definición de estructuras de datos, como se dijo en la sección 3.3, utilizando este mecanismo es posible definir relaciones y es efectivo para representar de manera abstracta ciertos entes esenciales en el estudio de la computación teórica.

Por ejemplo, la relación que indica si un proceso está bloqueado [33], la definimos de la siguiente forma:

Ejemplo 2.2. *Sea P un proceso, un proceso bloqueado ($P \nrightarrow$) es aquel que no tiene mas transiciones. Un proceso tiene una traza finita, denotada $P \downarrow$, si P tiene una secuencia de transiciones δ[1] que terminan en un proceso bloqueado.*

La especificación anterior tiene la siguiente definición inductiva:

$$\frac{P \ bloq}{P \downarrow} \qquad \frac{P \to_\delta P' \qquad P' \downarrow}{P \downarrow}$$

Determinar que un proceso P tiene una traza finita, basta aplicar la segunda regla hasta que el árbol de derivaciones termine en una hoja indicando un proceso P' bloqueado.

Considerando los procesos del ejemplo anterior, decimos que un proceso tendrá una $\omega-$traza si tiene una infinidad de reducciones δ, esto lo denotamos como $P \restriction_\delta$. Esta especificación contradice la esencia de las definiciones inductivas ya que, no tenemos forma de indicar un número infinito de reducciones. Esto lo podemos solucionar con los juicios coinductivos.

Definición 2.29. *Sea P un proceso, un proceso tiene una $\omega-$traza si es generada por el siguiente juicio coinductivo:*

$$\frac{P \to_\delta P' \qquad P' \restriction_\delta}{P \restriction_\delta}$$

Análogamente a los streams, la definición anterior no cuenta con reglas que construyan objetos básicos. En el caso particular de relaciones como las $\omega-$trazas, al no poder inferir cuándo un proceso está bloqueado, entonces podemos simular procesos que se ciclan.

Si se considera el juicio coinductivo y la inferencia de objetos básicos, este tipo de definiciones son conocidas como *potencialmente infinitas*. En la siguiente sección se construyen relaciones definidas coinductivamente que cumplan esta característica.

2.5.3. Secuencias infinitas

Como se mencionó, lograr abstraer entes infinitos requiere del mecanismo de coinducción para poder tener inferencias potencialmente infinitas. Por ejemplo, si el número de composiciones en la cerradura transitiva de una relación R fuese infinito, análogamente a las $\omega-$trazas, podemos dar un mecanismo que genere secuencias infinitas de elementos que estén $R-$relacionados.

[1] Las transiciones δ son pasos en la evaluación de procesos de cómputo.

Formalmente, esto se define con juicios coinductivos de la siguiente forma:

Definición 2.30 (Secuencias infinitas)**.** *Sean A un conjunto y R una relación binaria sobre A. Se define el conjunto de secuencias infinitas de elementos R-relacionados, denotada ${R^*}^\infty$, mediante el siguiente juicio coinductivo:*

$$\frac{(x,y) \in R \qquad y\ {R^*}^\infty}{x\ {R^*}^\infty}\ infiIntro$$

Aquí $x\ {R^*}^\infty$ denota que a partir x se genera a una secuencia infinita de elementos R-relacionados.

Con esta definición establecemos que si un elemento y pertenece a una secuencia infinita de R y agregamos el par (x, y) a la relación R, entonces el elemento x sigue perteneciendo a una secuencia infinita de R. Es importante observar que esta definición no define una nueva relación como el caso de la cerradura transitiva, estamos definiendo un *predicado* el cual indica cuándo es posible generar una secuencia infinita de elementos $R-$relacionados.

Análogamente al caso de los streams, no podemos construir directamente una secuencia infinita de elementos $R-$relacionados pero si podemos dar ejemplos de estos últimos. Por ejemplo, considerando la relación mayor estricto sobre los números enteros tenemos la secuencia infinita:

$$5 > 4 > 3 > \ldots$$

Dado que el número 5 genera una secuencia infinita de la relación $>$ y el par $(6, 5)$ pertenece a $>$, entonces el número 6 genera una secuencia infinita de enteros $>$-relacionados.

$$6 > 5 > 4 > 3 > \ldots$$

El problema con los objetos definidos coinductivamente es que resultados como el lema 3.8, no se pueden generar utilizando solamente la composición de relaciones binarias. Sin embargo, es posible utilizar el mecanismo de coinducción para indicar que existen composiciones infinitas.

Definición 2.31. *Sean A un conjunto y R, S relaciones binarias sobre A. Se define co-inductivamente la composición de S con secuencias posiblemente infinitas de R, denotado $coCD(R, S)$, mediante el siguiente sistema de juicios co-inductivos:*

$$\frac{(x,y) \in S}{(x,y) \in coCD(R,S)}\ cod \qquad \frac{(x,y) \in R \qquad (y,z) \in coCD(R,S)}{(x,z) \in coCD(R,S)}\ cotd$$

Esto nos da una observación importante de las definiciones:

- $cd(R, S)$. Indica que un par de elementos pertenece a S o bien, en un número de composiciones finitas de R se determina si un par de elementos pertenece a la relación $R^* \circ S$.
- $coCD(R, S)$. Indica que un par de elementos pertenece a S o bien, en un número finito o infinito de composiciones se determina si un par de elementos pertenece a la composición.

Es decir, el sistema de juicios de la Definición 2.31 captura tanto inferencias finitas como infinitas ya que el juicio *cod* detiene el proceso de inferencia.

Yuta Ikeda y Susumu Nishimura en [16] definen a *coCD* de la siguiente forma:

Definición 2.32. *Sean A un conjunto y R, S relaciones binarias sobre A, la cerradura de $S \circ R$ con secuencias potencialmente infinitas de R se define como:*

$$\nu X.(S \cup R \circ X)$$

Donde ν es el mayor punto fijo que satisface la ecuación:

$$X = S \cup R \circ X$$

Análogamente a definiciones inductivas, el teorema de Knaster-Tarski también establece criterios para definiciones coinductivas:

$$\nu X.f(X) = f(X.f(X)) \ \nu\text{-cómputo} \tag{2.3}$$
$$R \subseteq f(R) \Rightarrow R \subseteq \nu X.f(X) \ \nu\text{-inducción} \tag{2.4}$$

Sea f una función entre relaciones binarias con la siguiente regla de correspondencia $f(X) = S \cup X \circ R$, entonces se cumplen las propiedades de $\nu-$cómputo y $\nu-$inducción. A continuación, se muestra que el resultado de $\nu-$inducción es válido utilizando juicios coinductivos.

Proposición 2.4. *Sean A un conjunto y R, S, T relaciones binarias sobre A. Se cumple que si $T \subseteq S \cup (R \circ T)$, entonces $T \subseteq coCD(R, S)$.*

Demostración. Se asumirá que el comportamiento de la cerradura mantiene el resultado, es decir, la hipótesis de co-inducción establece que:

Sean A un conjunto y R, S, T relaciones binarias sobre A, entonces se cumple que si
$T \subseteq S \cup (R \circ T)$, entonces $T \subseteq coCD(R, S)$.

Sean R, S, T relaciones binarias sobre A, tales que $T \subseteq S \cup (R \circ T)$. Por demostrar que $T \subseteq coCD(R, S)$.

Sea $(x, y) \in T$, por demostrar que $(x, y) \in coCD(R, S)$.

Por hipótesis tenemos que $(x, y) \in S \cup (R \circ T)$, entonces haciendo un análisis de casos sobre la unión:

- Si $(x, y) \in S$, por la regla *cod* se satisface el resultado.
- Si $(x, y) \in R \circ T$, entonces existe un x_0 tal que $(x, x_0) \in R$ y $(x_0, y) \in T$.

  Por una parte se tiene que $(x, x_0) \in R$, entonces basta demostrar que $(x_0, y) \in coCD(R, S)$

en otras palabras, hay que demostrar que si $(x_0, y) \in T$ entonces $(x_0, y) \in coCD(R, S)$.

Por hipótesis de co-inducción e hipótesis se tiene que $T \subseteq coCD(R, S)$, entonces se satisface que $(x_0, y) \in coCD(R, S)$

$\boxed{\lambda}$

Obsérvese que, a diferencia de las demostraciones sobre los stream, los destructores de la cerradura no son tan claros, pero corresponden a los operadores de unión y composición.

El siguiente objetivo es mostrar que la composición inductiva y coinductiva son la misma, este resultado será de utilidad más adelante. Para demostrar este hecho necesitamos de los siguientes resultados.

La primera propiedad nos indica que la composición definida inductivamente está contenida en la definida coinductivamente.

Lema 2.20. *Sean A un conjunto y R, S relaciones binarias sobre A, entonces $cd(R, S) \subseteq coCD(R, S)$.*

Demostración. Sean R, S relaciones binarias sobre A, entonces considerando $x, y \in A$ tales que $(x, y) \in cd(R, S)$. Por demostrar que $(x, y) \in coCD(R, S)$, se procede por inducción sobre $(x, y) \in cd(R, S)$.

- **Caso base:** Sea $(x, y) \in S$ entonces por la regla *cod* se tiene que $(x, y) \in coCD(R, S)$.
- **Hipótesis de inducción:** Sean $x, y \in A$ tales que $(x, y) \in coCD(R, S)$.
- **Paso inductivo:** Sea $z \in A$ tal que $(x, y) \in R$ y $(y, z) \in cd(R, S)$, por demostrar que $(x, z) \in coCD(R, S)$.

  Entonces por la regla *cotd*, hipótesis e hipótesis de inducción se concluye que $(x, z) \in coCD(R, S)$.

$\boxed{\lambda}$

La siguiente propiedad nos dice que si un par de elementos (x, y) pertenecen a $coCD(R, S)$ y no pertenecen a $cd(R, S)$, entonces se debe satisfacer el predicado $y\ R^{*^{\infty}}$.

Lema 2.21. *Sean A un conjunto y R, S relaciones binarias sobre A, entonces se cumple que para cualesquiera $x, y \in A$, si $(x, y) \in coCD(R, S) \cap cd(R, S)^{-}$ entonces $y\ R^{*^{\infty}}$.*

Demostración. Se asumirá que el comportamiento de la cerradura mantiene el resultado, es decir, la hipótesis de co-inducción establece que:

Sean A un conjunto y R, S relaciones binarias sobre A, entonces se cumple que para cualesquiera $x, y \in A$, si $(x, y) \in coCD(R, S) \cap cd(R, S)^-$ entonces $y\ {R^*}^\infty$.

Sean R, S relaciones binarias sobre A y $(x, y) \in coCD(R, S) \cap cd(R, S)^-$, por demostrar que $y\ {R^*}^\infty$. Haciendo un análisis de casos sobre la intersección y la construcción co-inductiva de la cerradura reflexiva-transitiva se tiene que:

- $(x, y) \in S$ y $(x, y) \notin cd(R, S)$, lo cual es una contradicción ya que por el constructor $compDer_1$ se tiene que $(x, y) \in cd(R, S)$.
- $(x, y) \in R$, $(y, z) \in coCD(R, S)$ y $(x, z) \notin cd(R, S)$, entonces por la regla $infiIntro$ e hipótesis se tiene que demostrar que $y\ {R^*}^\infty$.
  Por hipótesis tenemos que $(x, z) \in coCD(R, S) \cap cd(R, S)^-$, entonces por hipótesis de co-inducción se obtiene lo solicitado.

$\boxed{\lambda}$

El siguiente lema garantiza que si un elemento x genera una secuencia infinita de R, entonces el par $(x, y) \in coCD(R, S)$ para cualquier y.

Lema 2.22. *Sean A un conjunto y R, S relaciones binarias sobre A, entonces se cumple que para cualesquiera $x, y \in A$, si $x\ {R^*}^\infty$ entonces $(x, y) \in coCD(R, S)$.*

Demostración. Se asumirá que el comportamiento de las secuencias infinitas mantienen el resultado, es decir, la hipótesis de co-inducción establece que:

Sean A un conjunto y R, S relaciones binarias sobre A, entonces se cumple que para cualesquiera $x, y \in A$, si $x\ {R^*}^\infty$ entonces $(x, y) \in coCD(R, S)$.

Sean R, S relaciones binarias sobre A y $x, y \in A$ tales que $x\ {R^*}^\infty$, por demostrar que $(x, y)\ coCD(R, S)$. Haciendo un análisis de casos sobre la construcción de las secuencias infinitas se tiene que $(x, y_0) \in R$ y $y_0 \in {R^*}^\infty$.

Por hipótesis de co-inducción se tiene que $(y_0, y) \in coCD(R, S)$ entonces por la regla $cotd$ e hipótesis se tiene que $(x, y) \in coCD(R, S)$. $\boxed{\lambda}$

Las propiedades anteriores nos muestran que un par (x, y) pertenece a la composición $coCD(R, S)$ si y sólo si (x, y) pertenece a la construcción inductiva o bien y genera una secuencia infinita de R.

Teorema 2.9. *Sean A un conjunto y R, S relaciones binarias sobre A, entonces se cumple que para cualesquiera $x, y \in A$, $(x, y) \in coCD(R, S)$ si y sólo si $(x, y) \in cd(R, S) \cup S$ o bien $x\ R^{*^{\infty}}$.*

Demostración. Sean R, S relaciones binarias sobre A, considerando dos elementos cualesquiera $x, y \in A$, se procede a demostrar cada implicación:

- Si $(x, y) \in coCD(R, S)$, por demostrar que $(x, y) \in cd(R, S) \cup S$ o bien $x\ R^{*^{\infty}}$.
  Dado que $(x, y) \in cd(R, S)$ o bien $(x, y) \notin cd(R, S)$, se procede a un análisis de casos sobre la disyunción:
  - Si $(x, y) \in cd(R, S)$ queda demostrada esta parte de la implicación.
  - Si $(x, y) \notin cd(R, S)$, entonces se demostrará que x pertenece a una secuencia infinita de R.
    Como por hipótesis $(x, y) \in coCD(R, S)$, entonces por el Lema 2.21 queda demostrada esta parte de la implicación.
- Si $(x, y) \in cd(R, S) \cup S$ o bien $x\ R^{*^{\infty}}$, por demostrar que $(x, y) \in coCD(R, S)$. Se hace un análisis de casos sobre la disyunción:
  - Si $(x, y) \in cd(R, S) \cup S$, entonces $(x, y) \in cd(R, S)$ o bien $(x, y) \in S$.
    Dado que $S \subseteq cd(R, S)$, entonces se cumple que $(x, y) \in cd(R, S)$.
    Por el Lema2.20 queda demostrada esta parte de la implicación.
  - Si $x\ R^{*^{\infty}}$ por el Lema 2.22 queda demostrada esta parte de la implicación.

$\boxed{\lambda}$

Finalmente, dado que la única diferencia entre la composición inductiva y coinductiva es la existencia de composiciones infinitas. Si ningún elemento puede generar una composición infinita entonces las relaciones resultantes son iguales.

Corolario 2.6. *Sean A un conjunto y R, S relaciones binarias sobre A, si para cualquier $x \in A$ no se cumple $x\ R^{*^{\infty}}$, entonces:*

$$cd(R, S) = coCD(R, S)$$

Demostración. Sean R, S relaciones binarias sobre A y se supondrá que para cualquier $x \in A$ no se cumple $x\ R^{*^{\infty}}$.

Por demostrar que $cd(R, S) = coCD(R, S)$, por demostrar:

- $cd(R,S) \subseteq coCD(R,S)$

  Se sigue del Lema 2.20.

- $coCD(R,S) \subseteq cd(R,S)$

  Sea $(x,y) \in coCD(R,S)$, por demostrar que $(x,y) \in cd(R,S)$.

  Por hipótesis y el Teorema 2.9 se tiene que $(x,y) \in cd(R,S) \cup S$ o bien $x\ {R^*}^\infty$, se procede a un análisis de casos sobre la disyunción:

  - Si $(x,y) \in cd(R,S) \cup S$, dado que que $S \subseteq cd(R,S)$ entonces se tiene que $(x,y) \in cd(R,S)$.
  - Si $x\ {R^*}^\infty$, esto contradice la hipótesis de que para cualquier $x \in A$ no se cumple $x\ {R^*}^\infty$. Por lo que se puede inferir que $(x,y) \in cd(R,S)$.

$\boxed{\lambda}$

En otras palabras, si se tiene que demostrar algo de la forma $R \subseteq \mu X.f(S \cup R \circ X)$, entonces la propiedad de μ-inducción no permite seguir con la demostración. Sin embargo, por el Corolario 2.6 indica $\mu X.f(S \cup R \circ X) = \nu X.f(S \cup R \circ X)$, lo que equivale a demostrar que:

$$R \subseteq \nu X.f(S \cup R \circ X)$$

Finalmente, permitiendo utilizar la propiedad de ν-inducción:

$$R \subseteq f(R)$$

Este resultado será de importancia en la interpretación relacional del lenguaje xPath.

Con este resultado queda finalizada la sección de coinducción y el capítulo de álgebra relacional. Es posible obtener mas propiedades utilizando los distintos materiales presentados en el capítulo pero no son de interés para los fines de este trabajo. En el siguiente capítulo, se hará la traducción de las funciones de navegación a un lenguaje relacional y se demostrarán distintas propiedades acerca de estas.

Capítulo 3

El zipper relacional

«La máquina analítica (máquina de Charles Babbage) (...) puede seguir análisis, pero no tiene el poder de anticipar ninguna relación analítica o verdad.»

Lovelace, Ada (1842) [26]

En los preliminares observamos que las funciones de navegación son funciones parciales. Por otra parte, en el capítulo 3 establecimos que traducir funciones parciales a relaciones binarias permite un análisis sin el problema de la parcialidad. En este capítulo traducimos las funciones de navegación de los zipper, que almacenan árboles binarios, a relaciones binarias y analizamos propiedades de simetría entre estas nuevas relaciones.

Para revisar la formalización de los resultados mostrados en este capítulo, se deben consultar los archivos siguientes:

- Las definiciones del zipper que almacena árboles binarios y sus funciones de navegación están en el archivo `Zipper_Def.v`.
- Las demostraciones de la parcialidad de las funciones de navegación están en el archivo `Zipper_Prop.v`.
- Las definiciones de las relaciones de navegación están en el archivo `Zipper_Rel.v`.
- Las propiedades de los elementos del punto anterior están en el archivo `Zipper_AR.v`.

Considerando el conjunto de todas las posibles relaciones binarias entre zippers de árboles binarios, de un tipo específico `a`, el conjunto $\mathcal{P}(Zipper \times Zipper)$ contiene las relaciones binarias que son traducciones de las funciones `dn_L`, `dn_R`, `up_L` y `up_R` [16].

Definición 3.1 (Relaciones de navegación)**.** *Las relaciones binarias Dn y Up sobre el conjunto Zipper están definidas por los siguientes juicios:*

$$\frac{}{((l, L\ i\ r :: c), (Node\ l\ i\ r, c)) \in Dn}\ dngL \qquad \frac{}{((r, R\ i\ l :: c), (Node\ l\ i\ r, c)) \in Dn}\ dngR$$

$$\frac{}{((Node\ l\ i\ r, c), (l, L\ i\ r :: c)) \in Up}\ upgL \qquad \frac{}{((Node\ l\ i\ r, c), (r, R\ i\ l :: c)) \in Up}\ upgR$$

Donde *dngL* significa *down generic left* y análogamente para el nombre de las reglas restantes.

Esta definición puede causar confusión ya que, la lectura de las relaciones es la siguiente:

- $(y, x) \in Dn$ es traducción de `dn_m`$(x) = y$ con `m` el movimiento izquierdo o derecho.
- $(y, x) \in Up$ es traducción de `up_m`$(x) = y$ con `m` el movimiento izquierdo o derecho.

Con esto, las relaciones binarias Dn,Up cumplen el objetivo de ser traducciones de las funciones navegación a un lenguaje relacional. Sin embargo, estas relaciones resultan demasiado generales para identificar si un par de zippers pertenecieron a un movimiento izquierdo o derecho.

Ejemplo 3.1. *Sean* `l,r` *árboles binarios de un tipo* `a` *e* `i` *un elemento del tipo* `a`*, la siguiente ecuación es válida para* `dn_L`*:*

```
dn_L (Node i l r,[]) = (l,[L i r])
```

Por la regla dngL, la ecuación anterior se traduce a la siguiente pertenencia:

$$((l, [L\ i\ r]), (Node\ i\ l\ r, [])) \in Dn$$

Obsérvese que se cambia el tipo de letra de los constructores (por ejemplo `Node`) a itálicas (*Node*) para diferenciar entre el contexto teórico y el de implementación. Yuta Ikeda y Susumu Nishimura en [16] no hacen la diferencia, pero consideramos que hacer esta distinción resulta más sencilla para el análisis.

En este caso, resulta sencillo averiguar que un zipper provino de una aplicación de `dn_L` pero puede complicarse al momento de considerar la composición de varias relaciones de navegación. Por otra parte, como se indicó en el capítulo anterior, las relaciones correflexivas permiten filtrar elementos que pertenezcan a un conjunto. Utilizando ciertas relaciones correflexivas podremos indicar de manera explícita el movimiento a través de un zipper.

Recordemos que utilizando un conjunto C, que es distinto al conjunto base de las relaciones binarias con las que trabajamos, podemos definir una relación binaria inducida por C. Por ejemplo, si consideramos a `L` como el conjunto de todos los posibles contextos cuya cabeza es una migaja izquierda de árboles binarios de tipo `a`, entonces `L`? es la relación binaria correflexiva cuyos elementos son de la forma (`L` $i\ r :: c$, `L` $i\ r :: c$).

Definición 3.2. *Sea* `a` *un tipo cualquiera, se define la relación correflexiva inducida de migajas izquierdas de tipo* `a` *de la siguiente forma:*

$$\texttt{L?} = \{\ (\texttt{L}\ i_1\ r_1 :: c_1,\ \texttt{L}\ i_2\ r_2 :: c_2) \mid \texttt{L}\ i_1\ r_1 :: c_1 = \texttt{L}\ i_2\ r_2 :: c_2\}$$

Entonces considerando a Ctx como el conjunto de todos los posibles contextos tenemos que $\texttt{L} \subseteq Ctx$, por lo que $\texttt{L?} \subseteq \wp(Ctx \times Ctx)$. En principio la relación `L?` no está contenida en el conjunto $\wp(Zipper \times Zipper)$, esto lo resolvemos utilizando el siguiente operador sobre relaciones binarias.

Definición 3.3 (Producto de relaciones binarias)**.** *Sean A, B dos conjuntos, R una relación binaria sobre el conjunto A y S una relación binaria sobre B. Se define el producto de R con S, denotado $R \times S$, como la siguiente relación binaria:*

$$R \times S = \{((x, w), (y, z)) \mid (x, y) \in R \wedge (w, z) \in S\}$$

Y se tiene que $R \times S : A \times B \leftarrow A \times B$.

Utilizando el producto de relaciones binarias sobre la relación $L?$ y la relación binaria identidad Id_T donde T es el conjunto de árboles binarios de tipo `a`, entonces la relación binaria $L? \times Id_T$ está contenida en el conjunto $\wp(Zipper \times Zipper)$.

Análogamente podemos definir la relación binaria `R?` que encapsula migajas derechas. Estas relaciones son correflexivas por definición, de manera que para determinar el movimiento en un zipper, basta utilizar las relaciones `L?` o `R?`.

Definición 3.4 (Relaciones de navegación con movimiento)**.**

$$dn_L = (id \times \texttt{L?}) \circ Dn \quad dn_R = (id \times \texttt{R?}) \circ Dn$$
$$up_L = Up \circ (id \times \texttt{L?}) \quad up_R = Up \circ (id \times \texttt{R?})$$

Con estas nuevas relaciones el par de zippers del ejemplo 3.1 pertenece únicamente a dn_L.

Ejemplo 3.2. *Sean* `l,r` *árboles binarios de un tipo* `a` *e* `i` *un elemento del tipo* `a`*, tal que la siguiente pertenencia se cumple:*

$$((l, [L\ i\ r]), (Node\ i\ l\ r, [])) \in dn_L$$

Para comprobar que $((l, [L\ i\ r]), (Node\ i\ l\ r, [])) \in dn_L$ basta mostrar un zipper z tal que $((l, [L\ i\ r]), z) \in id \times \texttt{L?}$ y $(z, (Node\ i\ l\ r, [])) \in Dn$. Considerando a $z = (l, [L\ i\ r]), z)$ la segunda condición se cumple. Por otra parte, tenemos que $l = l$ y $([L\ i\ r], [L\ i\ r]) \in \texttt{L?}$ ya que,

las cabezas de ambos contextos son el mismo constructor con los mismos parámetros; cumpliendo la primera condición.

Por lo que: $((l, [L\ i\ r]), (Node\ i\ l\ r, [])) \in dn_L$.

Ahora mostremos que $((l, [L\ i\ r]), (Node\ i\ l\ r, [])) \notin dn_R$. *Análogamente al caso anterior, basta mostrar un zipper* z *tal que* $((l, [L\ i\ r]), z) \notin id \times \texttt{R?}$ *o bien* $(z, (Node\ i\ l\ r, [])) \notin Dn$. *La primera condición se cumple ya que la cabeza de cualquier contexto para* z*, por definición de la relación* `R?`*, debe tener como constructor a* `R` *lo cual nunca empatará con el constructor* `L`*. Por lo que no existe un zipper* z *tal que:*

$$((l, [L\ i\ r]), (Node\ i\ l\ r, [])) \in dn_R$$

Observemos que las expresiones $id \times \texttt{L?}$ o $id \times \texttt{R?}$ logran diferenciar entre los movimientos izquierdos o derechos, ya que, se solicita que los focos se mantengan (por id), mientras que las relaciones `L?` y `R?` filtran las migajas de tal forma que se identifique si estos pares son rastros de movimientos izquierdos o derechos.

Con esto ya está hecha la traducción, en las siguientes secciones se hace un análisis relacional para verificar propiedades de simetría, para obtener la intuición buscada: *Poderse mover libremente en un árbol binario.*

3.1. Simetría

Una relación binaria R se dice que es simétrica si es igual a su relación inversa, es decir :

$$R = R^{\smile}$$

Para demostrar que las relaciones de navegación son inversas entre sí, es necesario verificar que las relaciones genéricas de navegación cumplan esta característica.

Teorema 3.1. *Las siguientes ecuaciones se cumplen:*

$$Dn^{\smile} = Up \tag{3.1}$$

$$Up^{\smile} = Dn \tag{3.2}$$

Demostración. Para demostrar que $Dn^{\smile} = Up$, basta demostrar ambas inclusiones. Para demostrar que $Dn^{\smile} \subseteq Up$ basta realizar un análisis de casos de Dn; el primero considerando l, r árboles binarios, i elemento del tipo y c contexto, tenemos que $((l, L\ i\ r :: c), (Node\ l\ i\ r, c)) \in Dn^{\smile}$, por definición de inversa corresponde a la construcción de Up,es decir,
$((Node\ l\ i\ r, c), (l, L\ i\ r :: c)) \in Up$. Análogamente para el segundo caso de Dn.

La otra contención resulta análoga: Realizar un análisis de casos sobre los constructores de Up y utilizar la definición de relación inversa.

Finalmente, utilizando la propiedad de involución de la inversa, se demuestra que $Up^{\smile} = Dn$. $\boxed{\lambda}$

Mediante el resultado anterior, la correflexividad de las relaciones `L?`, `R?` y la relación identidad, observamos que las relaciones de navegación cumplen ser simétricas.

Corolario 3.1. *Las siguientes ecuaciones se cumplen:*

$$dn_L^{\smile} = up_L \tag{3.3}$$

$$dn_R^{\smile} = up_R \tag{3.4}$$

$$up_L^{\smile} = dn_L \tag{3.5}$$

$$up_R^{\smile} = dn_R \tag{3.6}$$

Demostración. Se da la prueba para la Ecuación 3.3, la demostración del resto de ecuaciones es análoga.

Por definición se tiene que $dn_R^{\smile} = ((id \times \texttt{R?}) \circ Dn)^{\smile}$, entonces por semidistribución de la inversa sobre la composición se tiene que $dn_R^{\smile} = Dn^{\smile} \circ (id \times \texttt{R?})^{\smile}$.

Es inmediato que $(id \times \texttt{R?})^{\smile} = id \times \texttt{R?}$ por la correflexividad de la relación, entonces por la Ecuación 3.1 se tiene que $dn_R^{\smile} = Up \circ (id \times \texttt{R?}) = up_R$. $\boxed{\lambda}$

Dado que las relaciones de navegación son simétricas se esperaría que su composición resulte en la relación identidad. Sin embargo, esto no sucede en todos los casos por el comportamiento de las relaciones de navegación genéricas.

Proposición 3.1. *No todos los pares de zipper iguales pertenecen a la composición de Up y Dn, es decir:*

$$Id \subsetneq Up \circ Dn$$

Demostración. Hay que mostrar un par de zipper (z_1, z_2) tales que pertenezcan a la relación identidad pero no a la composición de Up y Dn.

Considerese el par de zipper $(z_1, z_2) = ((Empty, []), (Empty, []))$ y se quiere verificar que $(z_1, z_2) \notin Up \circ Dn$. Es claro que el par $((Empty, []), (Empty, []))$ nunca empatará con los constructores de Up ni de Dn, por lo que se cumple la no pertenencia. $\boxed{\lambda}$

Analizando el resultado anterior, observamos que hay elementos de la identidad que pertenecen a la composición. Estos casos son, cuando los focos (árboles no vacíos) y los contextos son iguales.

Para poder capturar estos elementos es necesario considerar ambos casos (foco y contexto) al mismo tiempo. Esto se logra por medio del producto de relaciones binarias.

Primero definamos una relación binaria cuyos elementos sean el par (t_1, t_2), donde t_1, t_2 son árboles binarios iguales pero no vacíos.

Definición 3.5. *La relación que contiene todos los árboles no vacíos iguales es generada por el siguiente juicio:*

$$\frac{Node\ l_1\ i_1\ r_1 = Node\ l_2\ i_2\ r_2}{(Node\ l_1\ i_1\ r_1, Node\ l_2\ i_2\ r_2) \in \texttt{Node?}}\ ncr$$

La relación binaria $\texttt{Node?} \times Id$ contiene todos los zipper cuyos focos y contextos son iguales, por lo que se tiene el siguiente corolario.

Proposición 3.2. *La composición de Up y Dn cumple la siguiente igualdad:*

$$Up \circ Dn = \texttt{Node?} \times Id$$

Demostración. Para la contención $\texttt{Node?} \times Id \subseteq Up \circ Dn$, se considera un par de zipper de la forma $((Node\ l\ i\ r, c), (Node\ l\ i\ r, c))$ por los constructores de Up y Dn el resultado es directo.
Para la otra contención consideremos un par de zippers $(z_1, z_2) \in Up \circ Dn$, por definición, existe un zipper z tal que $(z_1, z) \in Up$ y $(z, z_2) \in Dn$. Haciendo un análisis de casos sobre $(z_1, z) \in Up$ se tiene que dados i elemento del tipo, l, r árboles binarios y c contexto, $(z_1, z) = ((Node\ l\ i\ r, c), (l, L\ i\ r :: c)$ o bien $(z_1, z) = ((Node\ l\ i\ r, c), (r, R\ i\ l :: c)$, los cuales pertenecen a las relaciones dnL y dnR, respectivamente. $\boxed{\lambda}$

Por otra parte, la relación $Up \circ Dn$ es subconjunto de la relación identidad, lo cual es perfecto para los fines de navegación.

Lema 3.1. *Los pares de zipper (z_1, z_2) pertenecientes a $Up \circ Dn$ resultan ser iguales, en otras palabras:*

$$Up \circ Dn \subseteq Id$$

Demostración. Sea un par de zippers (z_1, z_2) tal que pertenece a $Up \circ Dn$, por definición existe un z tal que $(z_1, z) \in Up$ y $(z, z_2) \in Dn$, haciendo un análisis de casos sobre $(z_1, z) \in Up$ se tiene que dados i elemento del tipo, l, r árboles binarios y c contexto, $(z_1, z) = ((Node\ l\ i\ r, c), (l, L\ i\ r :: c)$ o bien $(z_1, z) = ((Node\ l\ i\ r, c), (r, R\ i\ l :: c)$.

El primer caso indica que $z = (l, L\ i\ r :: c)$, para que $(z, z_2) \in Dn$ necesariamente $z_2 = (Node\ l\ i\ r, c)$ por lo que $(z_1, z_2) \in Id$. Análogamente para el caso donde $z = (r, R\ i\ l :: c)$. $\boxed{\lambda}$

Puede pensarse que conmutando las relaciones de navegación se mantiene la propiedad, pero esto no resulta cierto ya que al realizar el análisis de casos sobre Up o Dn, existen focos o contextos que no permiten la igualdad.

Proposición 3.3. *No todos los pares de zipper (z_1, z_2) pertenecientes a $Dn \circ Up$ resultan ser iguales, en otras palabras:*

$$Dn \circ Up \subsetneq Id$$

Demostración. Por definición el par $((l, L\ i\ r :: c), (r, R\ i\ l :: c))$ pertenece a $Dn \circ Up$ pero no pertenece a la relación identidad. $\boxed{\lambda}$

Como consecuencia de los resultados, las relaciones de navegación son inyectivas, o dualmente, son relaciones binarias simples. Se indica esto, ya que Yuta Ikeda y Susumu Nishimura en [16], únicamente utilizan la propiedad de ser simples para sus cálculos pero en la verificación formal ambas propiedades son usadas.

Teorema 3.2. *Las relaciones dn_L, dn_R, up_L y up_R son simples, es decir, se cumplen las siguientes contenciones:*

$$dn_L \circ \breve{dn_L} \subseteq Id \tag{3.7}$$

$$dn_R \circ \breve{dn_R} \subseteq Id \tag{3.8}$$

$$up_L \circ \breve{up_L} \subseteq Id \tag{3.9}$$

$$up_R \circ \breve{up_R} \subseteq Id \tag{3.10}$$

Demostración. Se demuestran las contenciones 3.7 y 3.9, las restantes son análogas.

- Por la Ecuación 3.3 basta ver que $up_L \circ dn_L \subseteq Id$. Por definición de las relaciones de navegación $up_L \circ dn_L = Up \circ (id \times \mathtt{L?}) \circ (id \times \mathtt{L?}) \circ Dn$. Por la correflexividad de $L?$ se tiene que $id \times \mathtt{L?}$ es correflexiva y utilizando el Lema 3.1 se cumple la contención a la identidad.

- La Ecuación 3.7 no resulta tan sencilla (utilizando solamente ecuaciones y propiedades previas) como la anterior, para resolver esto hay que hacer un análisis de casos sobre $id \times L?$, el cual indica que los movimientos a realizar deben ser izquierdos. Esto obliga a que únicamente se cumplan los constructores dn_L y up_L, teniendo finalmente la identidad.

$\boxed{\lambda}$

En caso de querer ver la demostración detallada del teorema anterior, habrá que revisar la verificación formal. Ya que esta última tiene catorce variables y dos hipótesis correspondientes a los constructores de las relaciones de navegación, causando que la transcripción de la demostración resulte demasiado larga de dar.

Con esto se han resuelto cuestiones simétricas básicas acerca de las relaciones navegación, en la siguiente sección se dan algunas propiedades que afinan los posibles movimientos en un árbol encapsulado en un zipper.

3.2. Movimientos finos

A continuación demostramos la siguiente propiedad: Si a un zipper le podemos aplicar un movimiento arriba a la izquierda (up_L), entonces al zipper resultante del movimiento anterior no se le podría aplicar un movimiento abajo a la derecha (dn_L); y viceversa.

Lema 3.2. *Se cumplen las siguientes ecuaciones:*

$$up_L \circ dn_R = \emptyset \tag{3.11}$$

$$up_R \circ dn_L = \emptyset \tag{3.12}$$

Demostración. Para la primera ecuación solamente es necesario demostrar que $up_L \circ dn_R \subseteq \emptyset$.

Por las definiciones de composición y las relaciones up_L y dn_R, dados x, y, x_0, x_1 zippers si $(x, y) \in up_L \circ dn_R$, entonces por definición $(x, x_1) \in Up$, $(x_1, x_0) \in id \times \texttt{L?}$, $(x_0, x_2) \in id \times \texttt{R?}$ y $(x_2, y) \in Dn$.

Se observa que el zipper x_0 debe ser de la forma $(t, L\ i\ r :: c)$ por la hipótesis $(x_1, x_0) \in id \times \texttt{L?}$ y de la forma $(t, R\ i\ l :: c)$ por la hipótesis $(x_0, x_2) \in id \times R?$.

Implica que $L\ i\ r = R\ i\ l$ lo cual es una contradicción ya que son distintos constructores.

Por lo que los pares $(x_1, x_0), (x_0, x_2)$ no pueden existir, causando que la composición $up_L \circ dn_R$ resulte vacía.

Para la ecuación $up_R \circ dn_L = \emptyset$ el razonamiento es análogo. $\boxed{\lambda}$

Otra propiedad que salta a la vista es: Existen zippers tales que se les puede aplicar un movimiento abajo a la derecha y acto seguido aplicarles un movimiento arriba a la izquierda.

Lema 3.3. *Se cumple la siguiente ecuación:*

$$dn_R \circ up_L \neq \emptyset$$

Demostración. Sean los zipper $z_1 = (r, R\ i\ l :: c)$ y $z_2 = (l, L\ i\ r :: c)$ con i elemento del tipo, l, r árboles binarios y c contexto.

Entonces veamos que existe un zipper z tal que $(z_1, z) \in dn_R$ y $(z, z_2) \in up_L$, tomando a $z = (Node\ l\ i\ r, c)$ resulta directo de las definiciones de Up, Dn, `L?`, `R?` y producto que se cumple que $(z_1, z_2) \in dn_R \circ up_L$, es decir, la composición no es vacía. $\boxed{\lambda}$

Por otra parte, por la naturaleza de las relaciones de navegación solamente se han considerado las relaciones `L?` y `R?` para asegurar la existencia de migajas y la relación `Node?` para asegurar que un árbol no es vacío. Hay dos casos que son de importancia: Si el contexto está vacío y queremos hacer un movimiento hacia arriba o bien cuando el árbol éste vacío y queramos hacer un movimiento hacia abajo.

Definición 3.6. *La relación que representa el conjunto de los contextos vacíos, está generada por el siguiente juicio:*

$$\frac{}{([], []) \in \texttt{Top?}}\ nilcr$$

Definición 3.7. *La relación que representa el conjunto de hojas, está generada por el siguiente juicio:*

$$\frac{}{(Empty, Empty) \in \texttt{Leaf?}}\ emcr$$

Antes de seguir, es pertinente acalarar la diferencia de nombres utilizados aquí y en [16]:

- El árbol binario vacío se denota con el constructor `Leaf`, pero decidimos utilizar `Empty`.
- El contexto vacío de denota con el constructor `Top`, pero decidimos utilizar `Nil`.

Estas nuevas relaciones filtrarán elementos del zipper que no corresponden a ninguna de las relaciones anteriores, ya que estas últimas no consideran contextos o árboles binarios vacíos.

Lema 3.4. *Se cumple la siguiente ecuación:*

$$\texttt{L?} \circ \texttt{Top?} = \emptyset$$

Demostración. Por definición $(L\ i\ r :: c, L\ i\ r :: c) \in$ `L?` para i elemento del tipo, l, r árboles binarios y c contexto, entonces no es posible que $(L\ i\ r :: c, []) \in$ `Top?`, por lo que la composición es vacía. $\boxed{\lambda}$

Existen otras propiedades parecidas al Lema 3.4, que involucran las relaciones `Top`? y `Leaf`? y que se demuestran de manera análoga.

Con estas propiedades podemos verificar que ecuaciones brindan relaciones vacías.

Teorema 3.3. *Se cumplen las siguientes ecuaciones de las relaciones de navegación:*

$$up_L \circ (Id \times \texttt{Top}?) = \emptyset \tag{3.13}$$

$$up_R \circ (Id \times \texttt{Top}?) = \emptyset \tag{3.14}$$

$$dn_L \circ (\texttt{Leaf}? \times Id) = \emptyset \tag{3.15}$$

$$dn_R \circ (\texttt{Leaf}? \times Id) = \emptyset \tag{3.16}$$

Demostración. Únicamente se mostrará la Ecuación 3.13, la demostración de las otras ecuciones es análoga.

Por definición $up_L = Up \circ (Id \times \texttt{L}?)$, con esto basta ver que

$Up \circ (Id \times \texttt{L}?) \circ (Id \times \texttt{Top}?) = \emptyset$. Haciendo un análisis de casos sobre Up se tiene que la segunda entrada del par de zippers $((Node\ l\ i\ r, c), (l, L\ i\ r :: c))$ o bien el par $((Node\ l\ i\ r, c), (r, R\ i\ l :: c))$, debe cumplir la pertenencia $(Node\ l\ i\ r, []) \in \texttt{Top}?$.

Esto nunca sucede por la definición de $Top?$, entonces se cumple que $up_L \circ (Id \times \texttt{Top}?) \subseteq \emptyset$. $\boxed{\lambda}$

Con este resultado finalizamos las propiedades básicas de navegación del zipper relacional.

3.3. Atajos

Gracias a la simetría obtenemos provecho de las relaciones de navegación. Por ejemplo, por la Inclusión 3.7 del Teorema 3.2 tenemos que la relación $dn_L \circ up_L$ es correflexiva, entonces al ser compuesta con dn_L se tiene que $dn_L \circ up_L \circ dn_L \subseteq dn_L$. Un resultado interesante de este razonamiento es que, estas relaciones no solamente cumplen una inclusión, cumplen la igualdad.

Teorema 3.4. *Los siguientes atajos de las relaciones de navegación son válidos:*

$$dn_L = dn_L \circ \breve{dn_L} \circ dn_L \tag{3.17}$$

$$dn_R = dn_R \circ \breve{dn_R} \circ dn_R \tag{3.18}$$

$$up_L = up_L \circ \breve{up_L} \circ up_L \tag{3.19}$$

$$up_L = up_R \circ \breve{up_R} \circ up_R \tag{3.20}$$

Demostración. Se realizará la demostración de la Ecuación 3.17, el resto de las demostraciones son análogas.

Por lo observado al inicio de esta sección, la contención $dn_L \circ dn_L^{\smile} \circ dn_L \subseteq dn_L$ es clara. Demostremos la otra contención.

Sea un par de zippers $(z_1, z_2) \in dn_L$ basta demostrar que $(z_2, z_2) \in dn_L^{\smile} \circ dn_L$.

Haciendo un análisis de casos sobre z_2 y después sobre el foco que almacena se tienen dos casos:

- Si el foco es $Empty$, es falso que se cumpla $(z_2, z_2) \in Dn$.
- Dados i elemento del tipo, l, r árboles binarios y c contexto se tiene que demostrar que $((Node\ l\ i\ r, c), (Node\ l\ i\ r, c)) \in dn_L^{\smile} \circ dn_L$.

  Considerando el zipper $(l, L\ i\ r :: c)$, por la Ecuación 3.3 y definición de up_L y dn_L se cumple la pertenencia.

$\boxed{\lambda}$

A diferencia de [16], en la demostración anterior no fue necesario utilizar las propiedades `UpDn_Shunt` (mencionadas en la página 4 del artículo citado) ya que, al tratar de utilizarlas en el asistente de pruebas esta demostración se vuelve más larga. Una de las aplicaciones de los resultados anteriores, es el uso del zipper relacional como modelo de interpretación para el análisis del lenguaje xPath. En el siguiente capítulo se da una interpretación relacional de las expresiones xPath para obtener sus equivalentes.

Capítulo 4

Modelo relacional para xPath

«Porgramación funcional: El camino más corto para escribir y verificar un programa.»

Leroy, Xavier (2015) [21]

El lenguaje xPath dado en el capítulo 1, difiere del descrito por Dan Olteanu *et. al* en [31] y del lenguaje `XPATH`[1] al agregar el operador not. Yuta Ikeda y Susumu Nishimura en [16] deciden agregar dicho operador ya que, la interpretación con relaciones correflexivas permite una interpretación adecuada de dicho operador.

En este capítulo damos una interpretación relacional al lenguaje xPath como en [16], con la diferencia de que se realiza utilizando la teoría descrita en el capítulo 3 y utilizando la prueba asistida por computadora, para darle formalidad a los argumentos.

Los archivos que contienen la verificación formal de los resultados mostrados en este capítulo son:

- La gramática del lenguaje xPath y su interpretación relacional de las expresiones generadas por este lenguaje están en el archivo `XPath_Def.v`.
- Las equivalencias de ciertas expresiones xPath y su demostración, están en el archivo `XPath_Prop.v`.
- El análisis de una expresión xPath con presencia del operador not está en el archivo `XPath_Neg.v`.

4.1. Representación en árboles binarios

Como se vio en el capítulo 1, los documentos `XML` al tener una estructura jerárquica, son representados de manera fiel con los árboles $n-$arios. Sin embargo, esta abstracción no es la única. Yuta

[1] Respecto a su primera versión.

Ikeda y Susumu Nishimura en [16] dan una representación en árboles binarios para poder utilizar el zipper relacional que encapsula árboles binarios.

Utilizamos el siguiente procedimiento para transformar árboles $n-$arios en árboles binarios:

- Toda rama izquierda apunta al hijo más izquierdo de un nodo x.
- Toda rama derecha apunta al hermano inmediato a la derecha de x

Este procedimiento es el algoritmo *LCRS* (*left-child, right-sibling representation* por sus siglas en inglés), el cual toma como entrada un árbol n-ario y lo convierte a un árbol binario [6].

Los ejes de los nodos pueden ser clasificados en las siguientes categorías:

- Eje del mismo nodo. El eje del nodo *self* es un eje del nodo neutral que navega sobre el mismo nodo.
- Ejes del nodo siguiente. Los ejes del nodo *child*, *foll-sibling*, *desc*, *desc-or-self* y *following* siempre navegan hacia abajo.
- Ejes del nodo reverso. Los ejes del nodo *parent*, *presc-sibling*, *anc*, *anc-or-self* y *preceding* siempre navegan hacia arriba.

Es decir, a excepción del eje del nodo *self* los ejes del nodo se abstraen en un árbol binario. Esta representación de los documentos `XML` permite que la interpretación de los ejes del nodo siempre se ejecuten de acuerdo al orden del documento [16].

Por ejemplo, Yuta Ikeda y Susumu Nishimura en [16] brindan un ejemplo considerando el siguiente documento `XML`:

Código 4.1 Ejemplo dado en [16].

```
<a>
  <b></b>
  <c>
    <d></d>
    <e></e>
    <f>
      <g><h></h></g><i></i>
    </f>
    <j></j>
    <k><l></l></k>
  </c>
</a>
```

Cuya representación en árbol $n-$ario se muestra en la Figura 4.1.

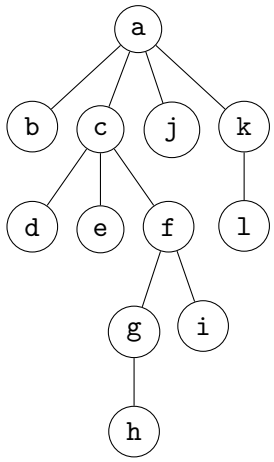


Figura 4.1: Árbol $n-$ario representante del Código 4.1.

Utilizando el algoritmo *LCRS*, el árbol $n-$ario de la Figura 4.1 lo convertimos en el árbol binario mostrado en la Figura 4.2.

Entonces, navegar en documentos `XML` es equivalente a navegar en un zipper tal que contenga el árbol binario que representa un documento `XML`. Por lo que la interpretación relacional de una expresión xPath $I(e)$, devuelve una relación que consiste en pares de zippers (z_1, z_2) tales que z_2 es el zipper que contiene el árbol binario que representa el documento `XML` original, mientras que z_1 es un zipper que representa un fragmento del documento `XML` de acuerdo a la expresión e.

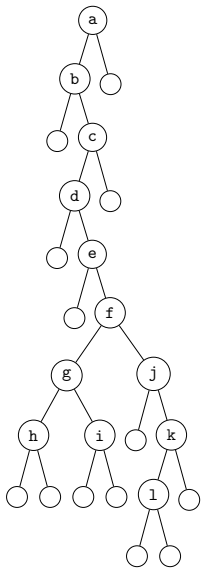


Figura 4.2: Árbol binario representante del código 4.1.

4.2. Modelo relacional

Considerando a $\wp(Zipper \times Zipper)$ como el conjunto de todos los posibles pares (z_1, z_2), con z_1, z_2 zippers que almacenan árboles binarios, los cuales son la abstracción de los documentos `XML`, definimos a la interpretación del lenguaje xPath como una función que dada una expresión xPath, devuelve relaciones contenidas en $\wp(Zipper \times Zipper)$.

Para lograr el objetivo anterior, la interpretación de cada expresión la generamos de acuerdo a la especificación dada en el capítulo 1, por ejemplo dado un nodo contexto x:

- La expresión *self* indica que se obtienen todas las etiquetas, y su contenido, almacenadas en el nodo contexto de x. Considerando como entrada al zipper $(t, [])$, donde t es el árbol binario representante de un documento `XML`, en este caso se debería devolver el mismo zipper. Por lo que la interpretación del eje self es la relación Id.
- La expresión *child* indica todas las etiquetas hijas, y su contenido, del nodo contexto x, entonces primero obtenemos todos los nodos hijos derechos, una vez teniendo estas etiquetas obtenemos el último nodo hijo izquierdo. Por lo que, la interpretación del eje child es la relación $dn_R^* \circ dn_L$.
- El resto de expresiones se interpretan de manera similar.

De manera que, la función de interpretación la definimos de acuerdo a cada una de las reglas de producción que definen al lenguaje xPath.

La función I_a es la interpretación de camino que definimos a continuación.

Definición 4.1 (Interpretación de ejes del nodo)**.** *Considerando a A como el conjunto de los ejes del nodo, definimos la interpretación de los ejes del nodo como:*

$$I_a : A \rightarrow (Zipper \times Zipper)$$

$$I_a(\text{self}) = Id \tag{4.1}$$

$$I_a(\text{child}) = dn_R^* \circ dn_L \tag{4.2}$$

$$I_a(\text{foll-sibling}) = dn_R^+ \tag{4.3}$$

$$I_a(\text{desc}) = (dn_R^* \circ dn_L)^+ \tag{4.4}$$

$$I_a(\text{desc-or-self}) = (dn_R^* \circ dn_L)^* \tag{4.5}$$

$$I_a(\text{following}) = (dn_R^* \circ dn_L)^* \circ dn_R^+ \circ (up_L \circ up_R^*)^* \tag{4.6}$$

$$I_a(\text{preceding}) = (dn_R^* \circ dn_L)^* \circ up_R^+ \circ (up_L \circ up_R^*)^* \tag{4.7}$$

$$I_a(\text{parent}) = up_L \circ up_R^* \tag{4.8}$$

$$I_a(\text{prec-sibling}) = up_R^* \tag{4.9}$$

$$I_a(\text{anc}) = (up_L \circ up_R^*)^+ \tag{4.10}$$

$$I_a(\text{anc-or-self}) = (up_L \circ up_R^*)^* \tag{4.11}$$

La función I_p es la interpretación de camino que definimos a continuación.

Definición 4.2 (Interpretación de caminos)**.** *Considerando a P como el conjunto de los caminos, definimos revusrivamente la interpretación de los caminos como:*

$$I_p : P \rightarrow (Zipper \times Zipper)$$

$$I_p(a :: *) = (Node? \times Id) \circ I_a(a) \tag{4.12}$$

$$I_p(a :: \beta) = \beta? \circ I_a(a) \tag{4.13}$$

$$I_p(p_1/p_2) = I_p(p_1) \circ I_p(p_2) \tag{4.14}$$

$$I_p(p[q]) = I_q(q) \circ I_p(p) \tag{4.15}$$

Donde I_q es la interpretación de caminos y Q es el conjunto de predicados:

$$I_q : Q \rightarrow (Zipper \times Zipper)$$

$$I_q(q_1 \text{ and } q_2) = I_q(q_1) \cap I_q(q_2) \tag{4.16}$$

$$I_q(q_1 \text{ or } q_2) = I_q(q_1) \cup I_q(q_2) \tag{4.17}$$

$$I_q(\text{not } q) = \neg I_q(q) \tag{4.18}$$

$$I_q(p) = dom(I_p(p)) \tag{4.19}$$

La función I_e es la interpretación de una expresión xPath que definimos a continuación.

Definición 4.3 (Interpretación de xPath)**.** *Considerando a E como el conjunto de las expresiones xPath, definimos revusrivamente la interpretación de estas expresiones como:*

$$I_e : E \rightarrow (Zipper \times Zipper)$$

$$I_e(/p) = I_p(p) \circ (Id \times Top?) \circ (up_L \cup up_R)^* \tag{4.20}$$

$$I_e(p) = I_p(p) \tag{4.21}$$

$$I_e(e_1 \cup e_2) = I_e(e_1) \cup I_e(e_2) \tag{4.22}$$

$$I_e(e_1 \cap e_2) = I_e(e_1) \cap I_e(e_2) \tag{4.23}$$

Usualmente, cuándo se ha llegado a este punto del desarrollo de cualquier lenguaje, una de las primeras preguntas que nos hacemos es: ¿Cómo definir cuándo dos expresiones son equivalentes? En el caso de computación, saber que dos expresiones son equivalentes permite tomar la decisión de definir un lenguaje muy pequeño o bien definir un lenguaje más amplio pero arriesgando la eficiencia del compilador.

Abordemos ahora la cuestión de verificar cuándo dos expresiones xPath son equivalentes. Para esto utilizamos las propiedades simétricas de las relaciones de navegación, con el fin de tener la teoría para poder implementar un analizador de documentos `XML` eficiente [16].

Para realizar esta tarea, primero definimos formalmente cuándo dos expresiones xPath son equivalentes:

Definición 4.4 (Equivalencia de expresiones xPath)**.** *Sean e_1, e_2 dos expresiones xPath, se dice que e_1 es equivalente a e_2, si se cumple la siguiente ecuación:*

$$I_e(e_1) = I_e(e_2)$$

En otras palabras, las relaciones binarias resultantes de la interpretación deben ser iguales.

En la siguiente subsección damos un ejemplo de equivalencia entre dos expresiones xPath.

4.2.1. La simetría child/parent

La especificación: *Obtener todos los nodos hijos de la etiqueta β cuyo padre sea la etiqueta α.* La representamos con la siguiente expresión:

$$\text{child} :: \beta[\text{parent} :: \alpha]$$

Sin embargo, puede ser parafraseada como la siguiente especificación: *Al considerar los nodos con etiqueta α, obtener los nodos hijos de las etiquetas β.* Que la representamos con la expresión:

$$\text{self} :: \alpha/\text{child} :: \beta$$

Para verificar que este parafraseo es válido, serán necesarios los siguientes lemas:

Lema 4.1. *La siguiente inclusión se cumple:*

$$up_R^* \circ dn_R^* \subseteq up_R^* \cup dn_R^*$$

Demostración. Sea $(x, y) \in up_R^* \circ dn_R^*$ por el Lema 2.9, tenemos que $(x, y) \in cd(up_R, dn_R^*)$, procedemos por inducción sobre esta expresión.

- **Caso base:** Si $(x, y) \in dn_R^*$ entonces se cumple que $(x, y) \in up_R^* \cup dn_R^*$.
- **Hipótesis de inducción:** Sea z un zipper tal que $(x, y) \in up_R$, $(y, z) \in cd(up_R, dn_R^*)$ y $(y, z) \in up_R^* \cup dn_R^*$.
- **Paso inductivo:** Haciendo un análisis de casos sobre $(y, z) \in up_R^* \cup dn_R^*$ se tiene:
  - Si $(y, z) \in up_R^*$ por la regla crt_2 se cumple la propiedad.
  - Si $(y, z) \in dn_R^*$, haciendo un análisis de casos sobre la construcción de la cerradura. utilizando la Ecuación 3.4 y la inyectividad de dn_L en el caso recursivo se demuestra la propiedad.

$\boxed{\lambda}$

Lema 4.2. *La siguiente inclusión se cumple:*

$$dn_R \circ Q \subseteq Q \circ dn_R$$

Donde $Q = (dn_L \circ \alpha? \circ up_L)[up_R]^*$.

Demostración. Sea $(x, y) \in dn_R \circ Q$, dado que Q es correflexiva, por las ecuaciones 3.18 y 3.4 y la inyectividad de dn_R, se tiene que $(x, y) \in dn_R \circ Q \circ up_R \circ dn_R$.
Entonces de la composición existe un zipper x_0 tal que $(x, x_0) \in dn_R \circ Q \circ up_R$ y $(x_0, y) \in dn_R$, entonces por la Ecuación 3.6 se tiene que $(x, x_0) \in \breve{up_R} \circ Q \circ up_R$, dado que $up_R \in [up_R]$ y por definición de la composición existen zipper x_1, x_2 tales que $(x, x_2) \in \breve{up_R}$, $(x_2, x_1) \in Q$ y $(x_1, x_0) \in up_R$, por lo que por la regla cs_2 se cumple que $(x, x_0) \in Q$, finalizando la demostración. $\boxed{\lambda}$

Lema 4.3. *Para cualesquiera relaciones binarias* R, S*, se cumple la siguiente inclusión:*

$$(R)[S]^* \subseteq \breve{S}^* \circ R \circ S^*$$

Demostración. Sea $(x, y) \in (R)[S]^*$, haciendo inducción sobre la estructura de la cerradura simétrica, el caso base se cumple de manera directa; mientras que el paso inductivo se cumple gracias a la transitividad de la cerradura reflexiva-transitiva. $\boxed{\lambda}$

Con los lemas anteriores, es posible demostrar la equivalencia de estas expresiones:

Proposición 4.1. *La siguiente equivalencia es válida:*

$$\text{child} :: \beta[\text{parent} :: \alpha] \equiv \text{self} :: \alpha/\text{child} :: \beta$$

Demostración. Por definición las expresiones son equivalentes si se cumple la siguiente ecuación:

$$I_e(\text{child} :: \beta[\text{parent} :: \alpha]) = I_e(\text{self} :: \alpha/\text{child} :: \beta)$$

Por un lado, de la definición de la interpretación se tiene que:

$$I_e(\text{child} :: \beta[\text{parent} :: \alpha]) = dom(\alpha? \circ up_L \circ up_R^*) \circ \beta? \circ dn_R^* \circ dn_L \tag{4.24}$$

Como $\bigcup[up_R] = up_R$ y dado que up_R es inyectiva, se sigue del Teorema 2.6 que a partir de la Ecuación 4.24 se cumple:

$$dom(\alpha? \circ up_L \circ up_R^*) \circ \beta? \circ dn_L = (dom(\alpha? \circ up_L))[up_R]^* \circ \beta? \circ dn_R^* \circ dn_L \tag{4.25}$$

Entonces, dado que $\alpha?$ es correflexiva, por el Teorema 2.2, por el Corolario 2.2 y por la Ecuación 3.5 se cumple la ecuación:

$$(dom(\alpha? \circ up_L))[up_R]^* \circ \beta? \circ dn_L = (dn_L \circ \alpha? \circ up_L)[up_R]^* \circ \beta? \circ dn_R^* \circ dn_L \tag{4.26}$$

Por otra parte, por la definición de la interpretación se tiene la ecuación:

$$I_e(\text{self} :: \alpha/\text{child} :: \beta) = \beta? \circ dn_R^* \circ dn_L \circ Id \circ \alpha? \tag{4.27}$$

Basta demostrar que se cumpla la siguiente igualdad de relaciones:

$$(dn_L \circ \alpha? \circ up_L)[up_R]^* \circ dn_R^* \circ \beta? \circ dn_R^* \circ dn_L = \beta? \circ dn_R^* \circ dn_L \circ Id \circ \alpha? \tag{4.28}$$

Al considerar a Q como $(dn_L \circ \alpha? \circ up_L)[up_R]^*$, dado que up_L es inyectiva y $\alpha?$ es correflexiva por definición y dn_L,up_L son inyectivas, se sigue del Lema 2.12 que Q es correflexiva, por el Lema 2.2 y neutro de Id bajo la composición se tiene que demostrar la ecuación:

$$\beta? \circ Q \circ dn_R^* \circ dn_L = \beta? \circ dn_R^* \circ dn_L \circ \alpha? \tag{4.29}$$

Donde basta demostrar la ecuación:

$$Q \circ dn_R^* \circ dn_L = dn_R^* \circ dn_L \circ \alpha? \tag{4.30}$$

Demostremos cada inclusión:

- $Q \circ dn_R^* \circ dn_L \subseteq dn_R^* \circ dn_L \circ \alpha?$.

  Sea $(x_0, z) \in Q \circ dn_R^* \circ dn_L$, por definición existen los zipper x_2, x_1 tales que $(x_0, x_2) \in Q$, $(x_2, x_1) \in dn_R^*$ y $(x_1, y) \in dn_L$.

  La definición de la cerradura simétrica aplicada a Q establece que dicha cerradura es la mínima relación binaria de la unión infinita de relaciones de la forma
  $up_R^{\smile} \circ (dn_L \circ \alpha? \circ up_L) \circ up_R$, por lo que se cumple que $(x_0, x_2) \in up_R^{\smile *} \circ (dn_L \circ \alpha? \circ up_L) \circ up_R^*$, este hecho es demostrado en el Lema 4.3.

  Por definición de composición existen los zipper x_4, x_3 tales que $(x_0, x_4) \in up_R^{\smile *}$, $(x_4, x_3) \in dn_L \circ \alpha? \circ up_L$, $(x_3, x_2) \in up_R^*$, $(x_2, x_1) \in dn_R^*$ y $(x_1, y) \in dn_L$.

  Tenemos que $(x_3, x_1) \in up_R^* \circ dn_R^*$, lo cual significa que (x_3, x_1) pertenece a la composición de relaciones de la forma $Id \cup up_R \cup \cdots \cup (up_R \circ \dots) \cup \dots$ y
  $Id \cup dn_R \cup \cdots \cup (dn_R \circ \dots) \cup \dots$, por distribución de la composición sobre la unión se tiene que $(x_3, x_1) \in up_R^* \cup dn_R^*$, este hecho se demuestra en el Lema 4.1.

  Nuevamente por definición de composición existe x_5 zipper tal que $(x_4, x_5) \in dn_L \circ \alpha?$ y $(x_5, x_3) \in up_L$, donde realizando un análisis de casos sobre $(x_3, x_1) \in up_R^* \cup dn_R^*$, se tiene:

  - Si $(x_3, x_1) \in up_R^*$, por el Teorema 2.4 se tiene que $(x_3, x_1) \in up_R^{*2}$. Haciendo un análisis de casos sobre la construcción de la cerradura se tiene:
    - Caso 1: $(x_3, x_1) \in Id$, es decir, $x_3 = x_1$ entonces $(x_5, x_1) \in up_L$ y $(x_1, y) \in dn_L$, por inyectividad de dn_L y la Ecuación 3.3 se concluye que que $(x_5, y) \in Id$.
    - Caso 2: Este caso no puede suceder, ya que, existen y_0, z zipper tales que $(x_1, y_0) \in up_R^{*2}$, $(y_0, z) \in up_R$ y $(z, y) \in dn_L$, entonces por la Ecuación 3.12 se tiene que $(y_0, y) \in \emptyset$.

    Por lo que $(x_5, y) \in Id$.
  - Si $(x_3, x_1) \in dn_R^*$, resulta análogo al caso anterior, pero haciendo un análisis de casos sobre la cerradura dn_R^* y utilizando la Ecuación 3.11 en lugar de la Ecuación 3.12.

  Con esto se tiene que $(x_0, x_4) \in up_R^{\smile *}$ y $(x_4, y) \in dn_L \circ \alpha?$.

- $dn_R^* \circ dn_L \circ \alpha? \subseteq Q \circ dn_R^* \circ dn_L$.

  Por el Lema 2.9 basta demostrar que $cd(dn_R, dn_L \circ \alpha?) \subseteq Q \circ dn_R^* \circ dn_L$, entonces sean x_0, y zipper tales que $(x_0, y) \in cd(dn_R, dn_L \circ \alpha?)$ se procede con inducción sobre esta expresión.

- **Caso base:** Se cumple que $(x_0, y) \in dn_L \circ \alpha?$, por la Ecuación 3.17 se tiene que $(x_0, y) \in dn_L \circ dn_L^{\smile} \circ dn_L \circ \alpha?$, entonces dado que $dn_L^{\smile} \circ dn_L$ y $\alpha?$ son correflexivas entonces $(x_0, y) \in dn_L \circ \alpha? \circ dn_L^{\smile} \circ dn_L$.

  Por definición de composición existen x_1, x_2, x_3 zipper tales que $(x_0, y) \in dn_L$, $(x_1, x_2) \in \alpha?$, $(x_2, x_3) \in up_L$ (por la Ecuación 3.3) y $(x_3, y) \in dn_L$, entonces por la regla cs_1 se tiene que $(x_0, x_3) \in Q$, dado que $(x_3, x_3) \in dn_R^*$ por la regla crt_1 entonces $(x_3, y) \in dn_R^* \circ dn_L$.

- **Hipótesis de inducción:** Sea z un zipper tal que $(x_0, y) \in dn_R$, $(y, z) \in cd(dn_R, dn_L \circ \alpha?)$ y $(y, z) \in Q \circ dn_R^* \circ dn_L$.

- **Paso inductivo:** Por demostrar que $(x_0, z) \in Q \circ dn_R^* \circ dn_L$.

  Por definición de composición existe x_1 zipper tal que $(x_0, x_1) \in Q$ y $(x_1, z) \in dn_R^* \circ dn_L$, entonces $(x_0, x_1) \in Q \circ dn_R \circ Q$.

  Dado que $Q = (dn_L \circ \alpha? \circ up_L)[up_R]^*$ y la cerradura simétrica indica la unión infinita de relaciones de la forma $up_R^{\smile} \circ (dn_L \circ \alpha? \circ up_L) \circ up_R$ al componer estas uniones con dn_R y utilizando la distributividad de la composición sobre la unión se cumple que $dn_R \circ Q \subseteq Q \circ dn_R$, este hecho se demuestra en el Lema 4.2.

  Por esta razón tenemos que $(x_0, x_1) \in Q \circ dn_R$ y por la regla crt_2 se cumple que $(x_0, z) \in Q \circ dn_R^* \circ dn_L$.

$\boxed{\lambda}$

Se observa que la demostración de esta equivalencia resulta larga y complicada de seguir. Por lo que en las siguientes secciones se darán *esbozos* de las pruebas, con el fin de seguir los argumentos dados, apoyándose en el asistente de pruebas `COQ`.

4.3. De la mano con `COQ`

Observamos que las demostraciones en la sección anterior resultan largas, tanto para su lectura como para su escritura. Por otra parte, los resultados presentados en este trabajo fueron verificados formalmente, utilizando el asistente de pruebas `COQ`. Por estas razones, en esta sección únicamente daremos esbozos de las demostraciones, correspondientes a las propiedades que en breve son presentadas.

Para seguir de manera clara los esbozos o bien si se tiene interés en revisar la verificación formal, debemos tener a la mano los archivos de `COQ` donde figura cada propiedad y su demostración

correspondiente. Esto con el fin de llevar una lectura paralela entre el escrito y las pruebas formales.

El desarrollo en COQ de este trabajo puede ser descargado en el siguiente enlace:

https://bitbucket.org/FerGaMen/tesis_fgm/src/master/

Por otra parte, cada lema, teorema, etc. mostrado a continuación tiene como nombre propio la etiqueta que le asignamos en el proceso de verificación formal, para que la consulta de cada demostración formal sea rápida. Por ejemplo, el teorema 4.8 corresponde al teorema `ec4_3_i_1` del archivo `XPath_Prop.v`. Se le asigna el nombre `ec4_3_i_1` ya que, es parte de la demostración del lema i para validar la ecuación 4.3 dada por Yuta Ikeda y Susumu Nishimura en [16].

4.3.1. La simetría descendant/ancestor

Las propiedades enunciadas y su demostración formal correspondientes a esta subsección, se encuentran en el archivo `XPath_Prop.v`.

De manera intuitiva, si las expresiones de *parent* y *child* son simétricas respecto a la composición, entonces pensaríamos que su dual lo cumple, es decir, que las expresiones *descendant* y *ancestor* también son simétricas respecto a la composición. Esto no sucede ya que, el eje *ancestor* captura hasta el nodo raíz de un documento `XML`. Para lograr que esta propiedad se cumpla, basta considerar que la simetría se dará si y sólo si el análisis del archivo en cuestión se realiza desde su etiqueta raíz.

La especificación: *Obtener todas las etiquetas descendientes β, las cuales tengan a las etiquetas α como antecesoras.* La representamos con la siguiente expresión:

$$\text{desc} :: \beta[\text{anc} :: \alpha]$$

De acuerdo al argumento anterior, pensaríamos que de manera similar a la simetría child/parent, la expresión anterior es equivalente a:

$$\text{desc-or-self} :: \alpha/\text{desc} :: \beta$$

La traducción a una especificación informal de esta última expresión es: *Considerando los ejes de los nodos descendientes con identificador α o el mismo eje del nodo, obtener aquellos cuyos descendientes sean los ejes del nodo con identificador β.*

Lo anterior establece que estas expresiones únicamente serán equivalentes si se les considera desde la etiqueta raíz, para verificar esta aseveración son necesarios los siguientes resultados:

Lema 4.4 (`upl_uprcrt_cm`)**.** *La siguiente ecuación se cumple:*

$$(up_L \circ up_R^*)^+ = up_L \circ (\bigcup [up_L, up_R])^*$$

Demostración. Por demostrar:

- $(up_L \circ up_R^*)^+ \subseteq up_L \circ (\bigcup[up_L, up_R])^*$.
  Esto lo demostramos utilizando el Corolario 2.4 y haciendo inducción sobre la estructura de ci. En el paso inductivo debemos utilizar el Lema 2.9.

- $up_L \circ (\bigcup[up_L, up_R])^* \subseteq (up_L \circ up_R^*)^+$.
  La demostramos utilizando el Corolario 2.4 y haciendo inducción sobre la estructura de ci. En el paso inductivo debemos desdoblar la Definición 2.19 y utilizar el Teorema 2.4.

$\boxed{\lambda}$

Lema 4.5 (`domdnl`)**.** *La siguiente inclusión se cumple:*

$$dom(dn_L) \subseteq Node? \times Id$$

Demostración. Basta realizar un análisis de casos sobre los árboles almacenados en los zipper, los cuales pertenecen a la relación dn_L. Estos árboles no pueden ser vacíos por el constructor de dn_L y dado que $dom(dn_L) = Id \cap dn_L^{\smile} \circ dn_L$, tenemos que los árboles y los contextos son iguales, logrando la inclusión de derecha a izquierda. La inclusión contraria se cumple por pura construcción del operador producto de relaciones y considerando la relación $Node?$. $\boxed{\lambda}$

Lema 4.6 (`dombetaCoref_upl`)**.** *La siguiente ecuación se cumple:*

$$dom(\alpha? \circ up_L) = dn_L \circ \alpha? \circ up_L$$

Demostración. Por demostrar que:

- $dom(\alpha? \circ up_L) \subseteq dn_L \circ \alpha? \circ up_L$
  La demostramos desdoblando la definición de dominio, utilizando la distribución de la inversa sobre la composición y la Ecuación 3.5.

- $dn_L \circ \alpha? \circ up_L \subseteq dom(\alpha? \circ up_L)$
  La demostramos desdoblando la definición de inversa, utilizando la distribución de la inversa sobre la composición, la Inclusión 3.7 del Teorema 3.2 y la Ecuación 3.3.

$\boxed{\lambda}$

Lema 4.7 (ec4_3_i_2)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$dn_R \circ Q \subseteq (dn_L \circ \alpha? \circ up_L \circ dn_R) \cup (dn_R \circ Q \circ up_R \circ dn_R) \cup (dn_L \circ Q \circ up_L \circ dn_R)$$

Demostración. Tenemos el siguiente análisis ecuacional:

$$(dn_L \circ \alpha? \circ up_L \circ dn_R) \cup (dn_R \circ Q \circ up_R \circ dn_R) \cup (dn_L \circ Q \circ up_L \circ dn_R) =$$

(por la Ecuación 3.13)

$$(dn_L \circ \alpha? \circ up_L \circ dn_R) \cup (dn_R \circ Q \circ up_R \circ dn_R) \cup (dn_L \circ Q \circ \emptyset) =$$

(por propiedades de álgebra relacional)

$$(dn_L \circ \alpha? \circ up_L \circ dn_R) \cup (dn_R \circ Q \circ up_R \circ dn_R) =$$

(por conmuatividad de correflexivas)

$$(dn_L \circ \alpha? \circ up_L \circ dn_R) \cup (up_R \circ dn_R \circ Q) =$$

(por la Ecuación 3.18)

$$(dn_L \circ \alpha? \circ up_L \circ dn_R) \cup (dn_R \circ Q)$$

$\boxed{\lambda}$

Lema 4.8 (ec4_3_i_1)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$dn_L \circ Q \subseteq (dn_L \circ \alpha? \circ up_L \circ dn_L) \cup (dn_R \circ Q \circ up_R \circ dn_L) \cup (dn_L \circ Q \circ up_L \circ dn_L)$$

Demostración. Razonamiento análogo al Lema 4.7. $\boxed{\lambda}$

Lema 4.9 (ec4_3_i_4)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$dn_R \circ Q \subseteq Q \circ dn_R$$

Demostración. Por el Lema 4.7 basta hacer un análisis de casos sobre las uniones. En el primer caso la demostración es resultado de la regla cs_1, el segundo caso por la regla cs_2 y la Ecuación 3.4 y el último caso por la regla cs_2 y la Ecuación 3.3. $\boxed{\lambda}$

Lema 4.10 (ec4_3_i_3)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$dn_L \circ Q \subseteq Q \circ dn_L$$

Demostración. Utilizando el Lema 4.8, la demostración es análoga al Lema 4.9. $\boxed{\lambda}$

Lema 4.11 (ec4_3_i)**.** *Si* $C = dn_R^* \circ dn_L$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$C \circ Q \subseteq Q \circ C$$

Demostración. Por el Lema 2.9 basta demostrar que $cd(dn_R, dn_L \circ Q) \subseteq Q \circ C$. Hacemos inducción sobre la cerradura *cd*:

- **Caso base**: Por distribución de la composición respecto a la unión, las inversas de las relaciones de navegación y el Lema 4.9 se cumple que $dn_L \circ Q \subseteq Q \circ dn_L$.
- **Paso inductivo**: Análogo al caso base se cumple que $dn_R \circ Q \subseteq Q \circ dn_R$; utilizando la hipótesis de inducción y definición se composición se cumple el paso inductivo.

$\boxed{\lambda}$

Lema 4.12 (ec4_3_ii_1)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$Q \circ dn_R^* \subseteq dn_R^* \circ Q$$

Demostración. Por el Corolario 2.4 basta demostrar que $ci(Q, dn_R) \subseteq dn_R^* \circ Q$, hacemos inducción sobre la cerradura *ci*:

- **Caso base**: Por definición de composición se cumple.
- **Paso inductivo:** Por hipótesis de inducción y realizando un análisis sobre la construcción de la cerradura simétrica, es decir, sobre Q, dos casos no serán válidos por la Ecuación 3.13 mientras que el último resulta correcto por la ecuación 3.10, por otra parte utilizando el Teorema 2.4, los constructores de la cerradura reflexiva-transitiva versión dos y la Ecuación 3.5 se cumple este caso.

$\boxed{\lambda}$

Lema 4.13 (ec4_3_ii)**.** *Si* $C = dn_R^* \circ dn_L$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$Q \circ C \subseteq C \circ (Q \cup \alpha?)$$

Demostración. Por el Lema 4.12 tenemos que $Q \circ dn_R^* \subseteq dn_R^* \circ Q$. Entonces por el Lema 2.9 se tiene que $dn_R^* \circ Q \circ dn_L \subseteq cd(dn_R, Q \circ dn_L)$, se procede por inducción sobre la cerradura:

- **Caso base**: Por distributividad, inyectividad de dn_L y las ecuaciones 3.3 y 3.5 se cumple el caso base.

  Como observación uno de los casos que involucra este caso, no es posible por la Ecuación 3.14.

- **Paso inductivo:** Por hipótesis de inducción y realizando un análisis de casos sobre esta y los constructores de la cerradura reflexiva-transitiva se cumple este caso.

$\boxed{\lambda}$

Lema 4.14 (ec4_3)**.** *Si* $C = dn_R^* \circ dn_L$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$C^+ \circ \alpha? \circ C^* \subseteq Q \circ C^+$$

Demostración. Desdoblando la Definición 2.19 y utilizando el Lema 2.9, basta hacer una inducción sobre la cerradura $cd(C, C \circ \alpha? \circ C^*)$:

- **Caso base**: Utilizando el Corolario 2.4 se realiza otra inducción sobre la cerradura $ci(C, \alpha?)$:
  - **Caso base**: Por el Lema 2.9, habrá que hacer otra inducción sobre la cerradura $cd(dn_R, dn_L \circ \alpha?)$ donde:
    - **Caso base**: Por la Ecuación 3.17, dado que $\alpha?$ y $dn_L^\smile \circ dn_L$ son correflexivas entonces conmutan demostrando el caso base.
    - **Paso inductivo**: Por la idea de las cerraduras reflexiva-transitiva y simétrica las relaciones dn_R y Q conmutan respecto a la composición (demostrado en el Lema 4.9), se cumple el paso inductivo.
  - **Paso inductivo**: Se demuestra por hipótesis de inducción, definición de composición y desdoblamiento de la cerradura transitiva.

- **Paso inductivo:** Por hipótesis de inducción y el Lema 4.9 se demuestra el lema.

$\boxed{\lambda}$

Lema 4.15 (ec4_4). *Si* $C = dn_R^* \circ dn_L$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, *entonces:*

$$C^+ \circ Q \subseteq Q \circ C^+$$

Demostración. Desdoblando la Definición 2.19 y utilizando el Lema 2.9, procedemos por inducción sobre $cd(C, C \circ Q)$:

- **Caso base:** Se cumple por el Lema 4.11 y definición de composición.
- **Paso inductivo:** Por hipótesis de inducción, definición de composición y el Lema 4.11 se demuestra el lema.

$\boxed{\lambda}$

Lema 4.16 (ec4_5). *Si* $C = dn_R^* \circ dn_L$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, *entonces:*

$$Q \circ C^+ \subseteq (C^+ \circ \alpha? \circ C^*) \cup (C^+ \circ Q)$$

Demostración. Por el Corolario 2.4 se tiene que demostrar $cd(Q \circ C, C) \subseteq (C^+ \circ \alpha? \circ C^*) \cup (C^+ \circ Q)$, procediendo sobre inducción sobre la estructura de ci, se tiene:

- **Caso base**: Por el Lema 4.13 tenemos que $Q \circ C \subseteq C \circ (Q \cup \alpha?)$.
  Entonces por definición de composición y realizando un análisis de casos sobre la unión el caso base queda demostrado.
- **Paso inductivo**: Por hipótesis de inducción y realizando un análisis de casos sobre la unión, se tiene que:
  - Caso 1: Se cumple por el Teorema 2.4.
  - Caso 2: Por definición de composición y utilizando el Lema 2.9 habrá que demostrar que $cd(C, C \circ Q \circ C) \subseteq (C^+ \circ \alpha? \circ C^*) \cup (C^+ \circ Q)$, esto se logra por inducción sobre la cerradura cd, donde el caso base se demuestra por el Lema 4.13, utilizando la definición de composición sobre el análisis de casos de la unión y las reglas de construcción de la cerradura reflexiva-transitiva; mientras que el paso inductivo hay que hacer un análisis

de casos sobre la unión dada en la hipótesis de inducción y la prueba resulta de los constructores de la cerradura reflexiva-transitiva.

$\boxed{\lambda}$

Recordemos que la meta es demostrar la siguiente equivalencia:

$$/\text{desc} :: \beta[\text{anc} :: \alpha] \equiv /\text{desc-or-self} :: \alpha/\text{desc} :: \beta$$

Para esto consideremos la primera expresión, eliminando la restricción de que sea una ruta exacta, es decir, la siguiente expresión:

$$\text{desc} :: \beta[\text{anc} :: \alpha]$$

Con los resultados anteriores obtenemos la siguiente equivalencia.

Lema 4.17 (ec4_2)**.** *La siguiente equivalencia es válida:*

$$\text{desc} :: \beta[\text{anc} :: \alpha] \equiv \text{desc-or-self} :: \alpha/\text{desc} :: \beta \cup \text{self} :: *[\text{anc} :: \alpha]/\text{desc} :: \beta$$

Demostración. Por definición de la interpretación se tiene que demostrar la ecuación:

$$\begin{aligned} dom(\alpha? \circ (up_L \circ up_R^*)^+) \circ \beta? \circ C^+ = \\ ((\beta? \circ C^+ \circ \alpha? \circ C^*) \cup (\beta? \circ C^+ \circ dom(\alpha? \circ (up_L \circ up_R^*)^+) \circ \\ (Node? \times Id) \circ Id)) \circ (Id \times Top?) \circ (up_L \cup up_R)^* \end{aligned} \tag{4.31}$$

donde $C = dn_R^* \circ dn_L$.

Considerando a $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, como up_L, up_R son inyectivas y es fácil verificar que $dn_L \circ \alpha? \circ up_L$ es correflexiva entonces Q es correflexiva, por el Lema 2.12, a parte por el Lema 4.4, el Lema 4.6, el Teorema 2.6 y el Lema 2.2 la Ecuación 4.31 se transforma a:

$$\beta? \circ Q \circ C^+ = (\beta? \circ C^+ \circ \alpha? \circ C^*) \cup (\beta? \circ C^+ \circ Q \circ (Node? \times Id) \circ Id) \tag{4.32}$$

Por demostrar ambas inclusiones:

- Por el Lema 4.16 tenemos que $Q \circ C^+ \subseteq (C^+ \circ \alpha? \circ C^*) \cup (C^+ \circ Q)$. Por lo que hay que demostrar que:

$$\begin{aligned} \beta? \circ ((C^+ \circ \alpha? \circ C^*) \cup (C^+ \circ Q)) \subseteq \\ (\beta? \circ C^+ \circ \alpha? \circ C^*) \cup (\beta? \circ C^+ \circ Q \circ (Node? \times Id) \circ Id) \end{aligned}$$

  Por la definición de composición y realizando un análisis de casos sobre la unión se tiene:

- Caso 1: Por definición de composición se cumple la union izquierda.
- Caso 2: Dado que $Node? \times Id \subseteq Id$ y definición de composición se debe demostrar que:

$$C^+ \circ Q \subseteq C^+ \circ Q \circ (Node? \times Id)$$

Por el Lema 4.15, tenemos que $C^+ \circ Q \subseteq Q \circ C^+$, entonces hay que demostrar que:

$$(C^+ \circ Q) \cap (Q \circ C^+) \subseteq C^+ \circ Q \circ (Node? \times Id)$$

Finalmente por definición de intersección, composición y por la Ecuación 4.5 se demuestra la inclusión:

$$(C^+ \circ Q) \cap (Q \circ C^+) \subseteq C^+ \circ Q \circ dom(dn_L)$$

- Para la otra inclusión se hace un análisis de casos sobre la unión:
  - Caso 1: Por el Lema 4.14 tenemos que $C^+ \circ \alpha? \circ C^* \subseteq Q \circ C^+$.
  - Caso 2: Por la neutralidad de Id respecto a la composición, definición de composición y la correflexividad de $Node?$ se tiene que $\beta? \circ C^+ \circ Q \circ (Node? \times Id) \subseteq \beta? \circ C^+ \circ Q$, entonces por la inclusión del Lema 4.15 se cumple este caso.

$\boxed{\lambda}$

Con el Lema 4.17 y nuevamente agregando la restricción de que la expresión es una ruta exacta, obtenemos la equivalencia que buscábamos.

Proposición 4.2 (`ec4_6`)**.** *La siguiente equivalencia es válida:*

$$/\text{desc} :: \beta[\text{anc} :: \alpha] \equiv /\text{desc-or-self} :: \alpha/\text{desc} :: \beta$$

Demostración. Por definición de la interpretación se tiene que:

$$I_e(/\text{desc} :: \beta[\text{anc} :: \alpha]) = (I_e(\text{desc} :: \beta[\text{anc} :: \alpha])) \circ (Id \times Top?) \circ (up_L \cup up_R)^* \tag{4.33}$$

Como se mencionó, las expresiones desc :: β[anc :: α] y desc-or-self :: α/desc :: β no son equivalentes, ya que la segunda no contiene nodos que si se consideran en la primera, estos nodos son: Los nodos que a partir de cualquier nodo cuyos antecesores sean el nodo α y sus descendientes sean β. En otras palabras, la siguiente expresión xPath:

$$\text{self} :: *[\text{anc} :: \alpha]/\text{desc} :: \beta$$

Por lo que la expresión desc :: β[anc :: α] resulta equivalente a
desc-or-self :: α/desc :: $\beta \cup$ self :: $*$[anc :: α]/desc :: β, este hecho se demuestra en en el Lema 4.17. Entonces de la Ecuación 4.33 se obtiene la siguiente ecuación:

$$(I_e(\text{desc-or-self :: } \alpha/\text{desc :: } \beta \cup \text{self :: } *[\text{anc :: } \alpha]/\text{desc :: } \beta)) \circ (Id \times Top?) \circ (up_L \cup up_R)^* = \quad (4.34)$$
$$(I_e(\text{desc-or-self :: } \alpha/\text{desc :: } \beta)) \circ (Id \times Top?) \circ (up_L \cup up_R)^*$$

Considerando a $C = dn_R^* \circ dn_L$ y a $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, por la definición de interpretación y la Ecuación 4.34, obtenemos la ecuación:

$$((\beta? \circ C^+ \circ \alpha? \circ C^*) \cup (\beta? \circ C^+ \circ dom(\alpha? \circ (up_L \circ up_R^*)^+) \circ (Node? \times Id) \circ Id)$$
$$\circ (Id \times Top?) \circ (up_L \cup up_R)^* = \quad (4.35)$$
$$\beta? \circ C^+ \circ \alpha? \circ C^* \circ (Id \times Top?) \circ (up_L \cup up_R)^*$$

Demostremos ambas inclusiones:

- La inclusión de izquierda a derecha se cumple por la definición de unión y composición.
- Para la otra inclusión, utilizando la definición de composición hay que hacer un análisis de casos sobre la unión:

  $$(\beta? \circ C^+ \circ \alpha? \circ C^*) \cup (\beta? \circ C^+ \circ dom(\alpha? \circ (up_L \circ up_R^*)^+) \circ (Node? \times Id) \circ Id)$$

  - El primer caso es directo.
  - Considerando la correflexividad de $Node?$ y la neutralidad de la relación identidad respecto a la composición se tiene que demostrar:

    $$(\beta? \circ C^+ \circ dom(\alpha? \circ (up_L \circ up_R^*)^+)) \circ (Id \times Top?) \circ (up_L \cup up_R)^* \subseteq$$
    $$\beta? \circ C^+ \circ \alpha? \circ C^* \circ (Id \times Top?) \circ (up_L \cup up_R)^*$$

    Tenemos que $(up_L \circ up_R^*)^+ = up_L \circ (\bigcup [up_L, up_R])^*$ (donde la expresión $\bigcup [x_1, x_2, \ldots, x_n]$ es la relación $x_1 \cup x_2 \cup \cdots \cup x_n$) por el Lema 4.4.
    Por lo que hay que demostrar que:

    $$(\beta? \circ C^+ \circ dom(\alpha? \circ up_L \circ (up_L \cup up_R)^*) \circ (Id \times Top?) \circ (up_L \cup up_R)^* \subseteq$$
    $$\beta? \circ C^+ \circ \alpha? \circ C^* \circ (Id \times Top?) \circ (up_L \cup up_R)^*$$

Por la inyectividad de up_L, up_R y el Teorema 2.6 se tiene que demostrar que:

$$(\beta? \circ C^+ \circ (dom(\alpha? \circ up_L))[up_L, up_R]^* \circ (Id \times Top?) \circ (up_L \cup up_R)^* \subseteq$$
$$\beta? \circ C^+ \circ \alpha? \circ C^* \circ (Id \times Top?) \circ (up_L \cup up_R)^*$$

Por el Lema 4.6 se tiene demostrar que:

$$\beta? \circ C^+ \circ Q \circ (Id \times Top?) \circ (up_L \cup up_R)^* \subseteq$$
$$\beta? \circ C^+ \circ \alpha? \circ C^* \circ (Id \times Top?) \circ (up_L \cup up_R)^*$$

Haciendo un análisis sobre la construcción de Q y por las ecuaciones 3.13, 3.14 se prueba la inclusión.

$\boxed{\lambda}$

Con esto finaliza el análisis de la simetría *descendant/ancestor*, sería deseable que todas las demostraciones hubiesen resultado en un razonamiento ecuacional como en el Lema 4.7. Sin embargo, esto es un impedimento del propio sistema de juicios. Lo que se puede rescatar de este análisis es la posibilidad de obtener los zippers exactos, tales que cumplan las propiedades anteriores.

Hasta este momento se han trabajado sobre expresiones xPath sin el operador de negación, en la siguiente sección hacemos el análisis sobre una expresión que tenga involucrada a este operador.

4.3.2. El caso de la expresión descendant/not ancestor

Las propiedades enunciadas y su demostración formal correspondientes a esta subsección, se encuentran en el archivo `XPath_Neg.v`.

Se tiene la siguiente cuestión: ¿Existirá una relación binaria más fácil de calcular que la dada por la interpretación de la expresión /desc :: β[not(anc :: α)]? Para resolver esto deberíamos obtener el resultado de la interpretación de la expresión anterior y utilizando álgebra relacional buscar una relación binaria más fácil de calcular.

Utilizando las propiedades básicas se tiene el siguiente análisis:

$$I(\text{desc :: [not(anc :: } \alpha)]) =$$
Por definición
$$\neg dom(\alpha? \circ (up_L \circ up_R^*)^+) \circ \beta? \circ (dn_R^* \circ dn_L)^+ =$$
Por el Lema 4.4

$$\neg dom(\alpha? \circ up_L \circ (\bigcup [up_L, up_R])^*) \circ \beta? \circ (dn_R^* \circ dn_L)^+ =$$

Como $[up_L, up_R]$ es una lista de relaciones inyectivas, por el Teorema 2.6

$$\neg (dom(\alpha? \circ up_L))[up_L, up_R]^* \circ \beta? \circ (dn_R^* \circ dn_L)^+ =$$

por el Lema 4.6

$$\neg (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^* \circ \beta? \circ (dn_R^* \circ dn_L)^+$$

Considerando a $C = dn_R^* \circ dn_L$ y $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, la expresión se reduce a:

$$\neg Q \circ \beta? \circ C^+$$

Para encontrar una relación binaria que sea más fácil de calcular que $\neg Q \circ \beta? \circ C^+$, necesitamos de los siguientes resultados:

Lema 4.18 (`dnrS_dnl_m`)**.** *Si $C = dn_R^* \circ dn_L$, entonces:*

$$C^+ = (dn_R \cup dn_L)^* \circ dn_L$$

Demostración. Por demostrar:

- $C^+ \subseteq (dn_R \cup dn_L)^* \circ dn_L$

  Utilizando el Corolario 2.4 basta demostrar que $ci(C, C) \subseteq (dn_R \cup dn_L)^* \circ dn_L$, entonces haciendo inducción sobre la cerradura ci, el caso base se demuestra por construcción de la cerradura reflexiva-transitiva, mientras que en el paso inductivo se utilizará la transitividad de la cerradura reflexiva-transitiva, el Corolario 2.4 y el Teorema 2.4 para resolver la primera inclusión.

- $(dn_R \cup dn_L)^* \circ dn_L \subseteq C^+$

  Por el Lema 2.9 basta demostrar que $cd(dn_R \cup dn_L, dn_L) \subseteq C^+$, utilizando inducción sobre cd tenemos que el caso base lo demostramos por construcción de la cerradura reflexiva-transitiva, mientras que el paso inductivo se debe hacer un análisis de casos sobre la unión en la hipotesis de inducción, donde cada caso se demuestra por construcción de la cerradura reflexiva-transitiva.

$\boxed{\lambda}$

Lema 4.19 (`dnrS_dnl_NbetaA_m`)**.** *Si $C = dn_R^* \circ dn_L$, entonces:*

$$(C \circ \neg\alpha?)^+ = (dn_R \cup dn_L \circ \alpha?)^* \circ dn_L \circ \neg\alpha?$$

Demostración. Por demostrar:

- $(C \circ \neg\alpha?)^+ \subseteq (dn_R \cup dn_L \circ \alpha?)^* \circ dn_L \circ \neg\alpha?$

  Utilizando el Corolario 2.4 por demostrar que:

  $$ci(dn_R^* \circ dn_L \circ \neg\alpha?, dn_R^* \circ dn_L \circ \neg\alpha?) \subseteq (dn_R \cup dn_L \circ \alpha?)^* \circ dn_L \circ \neg\alpha?$$

  Utilizando inducción sobre la cerradura ci, el caso base se demuestra por inducción sobre la cerradura reflexiva-transitiva, mientras que el paso inductivo se utilizará la transitividad de la cerradura reflexiva-transitiva.

- $(dn_R \cup dn_L \circ \alpha?)^* \circ dn_L \circ \neg\alpha? \subseteq (C \circ \neg\alpha?)^+$

  Utilizando el Lema 2.9 basta demostrar que:

  $$cd(dn_R \cup (dn_L \circ \neg\alpha?), dn_L \circ \neg\alpha) \subseteq (C \circ \neg\alpha?)^+$$

  Utilizando inducción sobre la cerradura ci, el caso base se demuestra por construcción de la cerradura reflexiva-transitiva, mientras que el paso inductivo se demuestra por análisis de casos y constructores de la cerradura reflexiva-transitiva, que da la hipótesis de inducción.

$\boxed{\lambda}$

Por otra parte, si $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$ y $C = dn_R^* \circ dn_L$, entonces por la correflexividad de Q tenemos el siguiente razonamiento ecuacional:

$$\neg Q \circ \beta? \circ C^+ = \beta? \circ (C \circ \neg\alpha?)^+ \circ \neg Q$$
$$\beta? \circ \neg Q \circ C^+ = \beta? \circ (C \circ \neg\alpha?)^+ \circ \neg Q$$

Por lo que hay que demostrar la siguiente ecuación:

$$\neg Q \circ C^+ = (C \circ \neg\alpha?)^+ \circ \neg Q$$

Para poder demostrar la ecuación anterior, es necesario tener los siguientes lemas.[1]

Lema 4.20 (`Q_mucomp`)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$Q = (dn_L \circ \alpha? \circ up_L)$$
$$\cup\, (dn_L \circ Q \circ up_L)$$
$$\cup\, (dn_R \circ Q \circ up_R)$$

[1] Hay que recordar que los juicios son una forma de definir cerraduras, estos juicios son un sistema equivalente al de Kanster-Tarski.

Demostración. Por demostrar ambas inclusiones:

- La primera inclusión ($\subseteq$) es un análisis de casos sobre las uniones y la construcción de la cerradura simétrica, en los juicios recursivos se deberán utilizar las ecuaciones 3.5 y 3.6.
- La otra inclusión es un análisis de casos sobre las uniones, en el primer caso de usa la regla cs_1 y en el restante la regla cs_2 con las ecuaciones 3.5 y 3.6.

$\boxed{\lambda}$

Aplicando las leyes de De Morgan, el Lema 2.18 y el Lema 4.20 tenemos que:

$$\neg Q = R_1 \cap R_2 \cap R_3 \tag{4.36}$$

Donde:

$$\begin{aligned} R_1 &= \neg dom(up_L) \cup (dn_L \circ \neg\alpha? \circ up_L) \\ R_2 &= \neg dom(up_L) \cup (dn_L \circ \neg Q \circ up_L) \\ R_3 &= \neg dom(up_R) \cup (dn_R \circ \neg Q \circ up_R) \end{aligned}$$

Para utilizar estas leyes primero hay que demostrar que R_1, R_2 y R_3 son relaciones correflexivas.

Lema 4.21 (`R1_coref`,`R2_coref`,`R3_coref`)**.**

$$R_1 = \neg dom(up_L) \cup (dn_L \circ \neg\alpha? \circ up_L)$$

$$R_2 = \neg dom(up_L) \cup (dn_L \circ \neg Q \circ up_L)$$

$$R_3 = \neg dom(up_R) \cup (dn_R \circ \neg Q \circ up_R)$$

Las relaciones R_1,R_2 y R_3 son correflexivas.

Demostración. Las tres son correflexivas ya que que las relaciones negadas por definición son correflexivas, luego por la correflexividad de $up_L^{\smile} \circ up_L$ y finalmente por la Ecuación 3.5 para R_1 y R_2 y por la Ecuación 3.6 para R_3, se cumple correflexividad para R_1,R_2 y R_3. $\boxed{\lambda}$

Con esto, demostramos la Ecuación 4.36.

Lema 4.22. *Si*

$$R_1 = \neg dom(up_L) \cup (dn_L \circ \neg\alpha? \circ up_L)$$

$$R_2 = \neg dom(up_L) \cup (dn_L \circ \neg Q \circ up_L)$$

$$R_3 = \neg dom(up_R) \cup (dn_R \circ \neg Q \circ up_R)$$

$$Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$$

entonces:

$$Q = R_1 \cap R_2 \cap R_3$$

Demostración. Aplicando la Ecuación 4.20 y las leyes de De Morgan hay que demostrar:

$$\neg(dn_L \circ \alpha? \circ up_L) \cap \neg(dn_L \circ Q \circ up_L) \cap \neg(dn_R \circ Q \circ up_R) = R_1 \cap R_2 \cap R_3$$

- Para el caso $\neg(dn_L \circ \alpha? \circ up_L) \subseteq R_1$, utilizando el hecho de la correflexividad de $\alpha?$, entonces hay que demostrar que $\neg(dn_L \circ \alpha? \circ up_L) \subseteq \neg dom(up_L) \cup dn_L \circ \neg dom(\alpha?) \circ up_L$.

  Como up_L es inyectiva y simple, por el Lema 2.18 y la Ecuación 3.3 basta demostrar que $\neg(dn_L \circ \alpha? \circ up_L) \subseteq \neg dom(\alpha? \circ up_L)$.

  Esto resulta de la definición de relación negada, la igualdad dada por $\alpha?$ y la Ecuación 3.5.

  Para los otros dos casos se tiene un razonamiento similar:

  - Demostrar que $\neg(dn_L \circ Q \circ up_L) \subseteq R_2$, se utiliza la Ecuación 3.5.
  - Demostrar que $\neg(dn_R \circ Q \circ up_R) \subseteq R_3$, se utiliza la Ecuación 3.6.

- Utilizando el hecho de que Q es correflexiva, el razonamiento es análogo al caso anterior.

$\boxed{\lambda}$

Lo que buscamos es una relación que sea igual a la relación $\neg Q \circ \beta? \circ C^+$, pero más sencilla de calcular. Para lograr esto, analicemos las composiciones de las relaciones de navegación con las relaciones R_1, R_2 y R_3.

Como primera observación se tiene que la relación $\neg dom(up_L)$ está contenida en R_1 y R_2, entonces al componer con la relación dn_L resulta en la relación vacía.

Lema 4.23 (`negDomUpL_dnL_Vacia`)**.** *La siguiente ecuación es válida:*

$$\neg dom(up_L) \circ dn_L = \emptyset$$

Demostración. Dado que la inclusión de derecha a izquierda siempre es válida, basta demostrar la otra inclusión.

Por las ecuaciones 3.5, 2.1 y por distribución de la inversa respecto a la composición y su definición demostrar la primera inclusión se reduce a demostrar que

$(\neg dom(up_L)) \circ dom(up_L) \circ up_L^{\smile} \subseteq \emptyset$. Como las relaciones negadas y las relaciones dominio son correflexivas, entonces por el Lema 2.5 basta demostrar que $((\neg dom(up_L)) \cap dom(up_L)) \circ up_L^{\smile} \subseteq \emptyset$, entonces tenemos que $(\neg dom(up_L)) \cap dom(up_L) = \emptyset$. $\boxed{\lambda}$

Por otra parte, dado que la relación $\neg dom(up_R)$ está contenida en R_3, su comportamiento respecto a dn_L es distinto al resultado anterior.

Lema 4.24 (dnL_cont_negDomUpRDnL)**.** *La siguiente inclusión es válida:*

$$dn_L \subseteq (\neg dom(up_R)) \circ dn_L$$

Demostración. Por la Ecuación 3.5 y el Teorema 2.1 basta demostrar la inclusión $dom(up_L) \circ up_L^{\smile} \subseteq (\neg dom(up_R)) \circ dn_L$ donde $dom(up_L) \subseteq \neg dom(up_R)$, como el dominio es correflexivo, por el Lema 2.15 se tendría que demostrar que $\neg\neg dom(up_L) \subseteq dom(up_R)$.
Aplicando la conexión de Galois 2.7 equivale a demostrar $Id \subseteq (\neg dom(up_L)) \cup (\neg dom(up_R))$, por las leyes de De Morgan, basta demostrar que $Id \subseteq \neg(dom(up_L) \cap dom(up_R))$. Dado que $(dom(up_L) \cap dom(up_R) = \emptyset$ ya que las relaciones up_L y up_R son simples (correflexivas considerando sus inversas), entonces aplicando el Lema 2.5 y las ecuaciones 3.5, 3.6 y 3.2 se llega a la contradicción. $\boxed{\lambda}$

Con esto se pueden obtener las siguientes propiedades del comportamiento de las relaciones R_1, R_2 y R_3 respecto a dn_L y dn_R.

Lema 4.25 (R1_dnL)**.** *Si* $R_1 = \neg dom(up_L) \cup (dn_L \circ \neg\alpha? \circ up_L)$*, entonces:*

$$R_1 \circ dn_L = dn_L \circ \neg\alpha?$$

Demostración. La igualdad $R_1 \circ dn_L = ((\neg dom(up_L)) \circ dn_L) \cup (dn_L \circ (\neg\alpha?))$ es válida por la propiedad de distribución de la composición respecto a la unión, es decir:

$$((\neg dom(up_L)) \circ dn_L) \cup (dn_L \circ (\neg\alpha?) \circ (up_L \circ dn_L)) = ((\neg dom(up_L)) \circ dn_L) \cup (dn_L \circ (\neg\alpha?))$$

Por demostrar:

- $((\neg dom(up_L)) \circ dn_L) \cup (dn_L \circ (\neg\alpha?) \circ (up_L \circ dn_L)) \subseteq ((\neg dom(up_L)) \circ dn_L) \cup (dn_L \circ (\neg\alpha?))$
  Hacemos un análisis de casos sobre la unión, donde el primer caso no es válido por el lema 4.23 y el segundo caso es resuleto por la Ecuación 3.3 y la correflexividad de $dn_L^{\smile} \circ dn_L$.
- La otra inclusión es análoga al caso anterior.

Ahora, por transitividad basta demostrar que:

$$((\neg dom(up_L)) \circ dn_L) \cup (dn_L \circ (\neg\alpha?)) = dn_L \circ \neg\alpha?$$

Donde la relación $(\neg dom(up_L)) \circ dn_L)$ es vacía por el Lema 4.23. $\boxed{\lambda}$

Lema 4.26 (R2_dnL)**.** *Si* $R_2 = \neg dom(up_L) \cup (dn_L \circ \neg Q \circ up_L)$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$R_2 \circ dn_L = dn_L \circ \neg Q$$

Demostración. La demostración es análoga a la anterior. $\boxed{\lambda}$

Lema 4.27 (R3_dnL)**.** *Si* $R_3 = \neg dom(up_R) \cup (dn_R \circ \neg Q \circ up_R)$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$R_3 \circ dn_L = dn_L$$

Demostración. Por la distribución de la unión sobre la composición, basta demostrar:

$$((\neg dom(up_R)) \circ dn_L) \cup (dn_R \circ Q \circ up_R \circ dn_L) = dn_L$$

Por demostrar:

- $((\neg dom(up_R)) \circ dn_L) \cup (dn_R \circ Q \circ up_R \circ dn_L) \subseteq dn_L$
  Hacemos un análisis de casos sobre la unión, el primer caso se satisface por el Lema 4.24 y el segundo caso resulta en la relación vacía por la Ecuación 3.2.
- $dn_L \subseteq ((\neg dom(up_R)) \circ dn_L) \cup (dn_R \circ Q \circ up_R \circ dn_L)$
  Se sigue del Lema 4.24.

$\boxed{\lambda}$

Lema 4.28 (R1_dnR)**.** *Si* $R_1 = \neg dom(up_L) \cup (dn_L \circ \neg \alpha? \circ up_L)$*, entonces:*

$$R_1 \circ dn_R = dn_R$$

Demostración. Por la distribución de la unión sobre la composición, basta demostrar que:

$$((\neg dom(up_L)) \circ dn_R) \cup (dn_L \circ (\neg \alpha?)) = dn_R$$

Por demostrar:

- $((\neg dom(up_L)) \circ dn_R) \cup (dn_L \circ (\neg \alpha?)) \subseteq dn_R$
  Hacemos un análisis de casos sobre la unión, el primer caso por definición de dominio se satisface el resultado y el segundo resulta en la relación vacía por la Ecuación 3.2.

- $dn_R \subseteq ((\neg dom(up_L)) \circ dn_R) \cup (dn_L \circ (\neg \alpha?))$

  Por la Ecuación 3.6 y el Teorema 2.1 propiedades de relaciones inversas y definición de dominio, basta demostrar que:

  $$dom(up_R) \circ dn_R \subseteq (\neg dom(up_L)) \circ dn_R$$

  Por definición de composición basta demostrar que $dom(up_R) \subseteq \neg dom(up_L)$. Por la doble negación en relaciones binarias y la conexión de Galois de la diferencia y unión basta ver que:

  $$Id \subseteq (\neg dom(up_R)) \cup (\neg dom(up_L))$$

  Por definición de relación negada, las leyes de De Morgan y las ecuaciones 3.5 y 3.2 se cumple la inclusión.

$\boxed{\lambda}$

Lema 4.29 (R2_dnR)**.** *Si* $R_2 = \neg dom(up_L) \cup (dn_L \circ \neg Q \circ up_L)$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, *entonces:*

$$R_2 \circ dn_R = dn_R$$

Demostración. Análoga a la ecuación anterior. $\boxed{\lambda}$

Lema 4.30 (R3_dnR)**.** *Si* $R_3 = \neg dom(up_R) \cup (dn_R \circ \neg Q \circ up_R)$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, *entonces:*

$$R_3 \circ dn_R = dn_R \circ \neg Q$$

Demostración. Por la distribución de la inversa sobre la unión basta demostrar la igualdad:

$$((\neg dom(up_R)) \circ dn_R) \cup (dn_R \circ (\neg Q) \circ up_R \circ dn_R) = dn_R \circ \neg Q$$

Por demostrar:

- $((\neg dom(up_R)) \circ dn_R) \cup (dn_R \circ (\neg Q) \circ up_R \circ dn_R) \subseteq dn_R \circ \neg Q$

  Hacemos un análisis de casos sobre la unión, donde:

  - Este caso no es válido: Por propiedades de la inversa, la Ecuación 3.6 y el Teorema 2.1 basta demostrar:

    $$(\neg dom(up_R)) \circ dom(up_R) \circ dn_R \subseteq dn_R \circ \neg Q$$

Dado que toda relación binaria negada es correflexiva, entonces por el Lema 2.5 se cumple:

$$((\neg dom(up_R)) \cap dom(up_R)) \circ dn_R \subseteq dn_R \circ \neg Q$$

Donde por la ley del tercer excluido se cumple la contradicción.

- El segundo caso es resultado de la correflexividad de $up_R \circ up_R^{\smile}$ y la Ecuación 3.6.

- La otra inclusión es resultado de la Ecuación 3.4.

$\boxed{\lambda}$

Dado que $\neg Q$ es correflexiva y dn_R son correflexivas entonces conmutan respecto a la composición y casi conmutación con dn_L, esto último por la verificación que debe hacer con etiquetas α.

Lema 4.31 (`lema_i`)**.** *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$\neg Q \circ dn_R = dn_R \circ \neg Q$$

Demostración. Se tiene el siguiente razonamiento ecuacional:

$$\neg Q \circ dn_R = R_3 \circ dn_R$$

(por el Lema 4.30)

$$\neg Q \circ R_2 \circ dn_R = R_3 \circ R_2 \circ dn_R$$

(por el Lema 4.29)

$$\neg Q \circ R_2 \circ R_1 \circ dn_R = R_3 \circ R_2 \circ R_1 \circ dn_R$$

(por el Lema 4.28)

$$\neg Q \circ dn_R = R_3 \circ R_2 \circ R_1 \circ dn_R \text{ por 4.29, 4.28}$$

$$(R_1 \cap R_2 \cap R_3) \circ dn_R = R_3 \circ R_2 \circ R_1 \circ dn_R$$

(por el Lema 4.36)

Por demostrar:

- $(R_1 \cap R_2 \cap R_3) \circ dn_R \subseteq R_3 \circ R_2 \circ R_1 \circ dn_R$

  Se sigue de la correflexividad de la relación R_1.

- $R_3 \circ R_2 \circ R_1 \circ dn_R \subseteq (R_1 \cap R_2 \cap R_3) \circ dn_R$

  Se sigue de las correflexividades de las relaciones R_1, R_2 y R_3.

$\boxed{\lambda}$

Lema 4.32 (`lema_ii`). *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$, *entonces:*

$$\neg Q \circ dn_L = dn_L \circ \alpha? \circ \neg Q$$

Demostración. La demostración es análoga a la ecuación anterior, considerando que:

$$\neg Q = R_1 \cap R_2 \cap R_3 \text{ por la Ecuación 4.36}$$

$\boxed{\lambda}$

Dado que los árboles y contextos con los que se trabaja son finitos, ya que son representaciones abstractas de documentos `XML`, podemos asumir lo siguiente [16]:

Para cualquier relación binaria S sobre los zipper que encapsulan árboles binarios, no existe un zipper tal que genere secuencias infinitas de zippers $S-$relacionados. En otras palabras y abusando un poco de la notación:

$$\forall S : Zipper \times Zipper\ \neg\exists x : zipper,\ (x\ S^{*^\infty}) \tag{4.37}$$

Con esto ya se puede utilizar el Corolario 2.6 para las relaciones sobre el conjunto $Zipper$, el cual garantiza que las cerraduras $cd(S, R)$ y $coCD(S, R)$ son iguales. Esto permite que en expresiones de la forma $R \subseteq \mu X.f(X)$ utilicemos la propiedad de ν-inducción.

Un resultado utilizando el Corolario 2.6, es el siguiente:

Lema 4.33 (`ec5_1_pre_i_ii`). *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$ *y* $C = dn_R^* \circ dn_L$, *tal que:*

$$(\neg Q) \circ C \subseteq (dn_L \circ (\neg\alpha?) \circ (\neg Q)) \cup (dn_R \circ (\neg Q) \circ C)$$

Entonces se cumple que:

$$(\neg Q) \circ C \subseteq (C \circ (\neg\alpha?) \circ (\neg Q))$$

Demostración. Por el Lema 2.9 basta demostrar que:

$$(\neg Q) \circ C \subseteq cd(dn_R, dn_L \circ (\neg\alpha?) \circ (\neg Q))$$

Por la Hipótesis 4.37 y el Corolario 2.6 tenemos que:

$$(\neg Q) \circ C \subseteq coCD(dn_R, dn_L \circ (\neg\alpha?) \circ (\neg Q))$$

Y utilizando la Proposición 2.4, se obtiene la inclusión anterior. $\boxed{\lambda}$

Con esto, resulta más notorio considerar la verificación de etiquetas α para que las relaciones $\neg Q$ y C estén cerca de cumplir la conmutatividad.

Lema 4.34 (`lema_iii`). *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$ *y* $C = dn_R^* \circ dn_L$, *entonces:*

$$(\neg Q) \circ C = C \circ \alpha? \circ (\neg Q)$$

Demostración. Por demostrar:

- $(\neg Q) \circ C \subseteq C \circ \alpha? \circ (\neg Q)$

  Por los lemas 4.33, 4.31 y 4.32 basta demostrar que:

  $$(\neg Q) \circ C \subseteq ((\neg Q) \circ dn_L) \cup ((\neg Q) \circ dn_R \circ C)$$

  Esta inclusión es válida por el Lema 2.9 y haciendo un análisis de casos sobre la construcción de la cerradura *cd*.

- $C \circ \alpha? \circ (\neg Q) \subseteq (\neg Q) \circ C$

  Por el Lema 2.9 basta demostrar que:

  $$cd(dn_R, dn_L \circ (\neg\alpha?) \circ (\neg Q)) \subseteq (\neg Q) \circ C$$

  Esto lo demostramos por inducción sobre la cerradura *cd*, tal que en el paso inductivo por el Lema 4.31 se satisface esta inclusión.

$\boxed{\lambda}$

Generalizando el resultado anterior a C^+, se obtiene el resultado esperado.

Corolario 4.1 (`subEc5_1`). *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$ *y* $C = dn_R^* \circ dn_L$, *entonces:*

$$(C \circ \neg\alpha?)^+ \circ \neg Q = \neg Q \circ C^+$$

Demostración. Por demostrar:

- $(C \circ \neg\alpha?)^+ \circ \neg Q \subseteq \neg Q \circ C^+$

  Por el Lema 4.19 y el Lema 2.9 basta demostrar que:

  $$cd(dn_R \cup (dn_L \circ \neg\alpha?), dn_L \circ (\neg\alpha?) \circ \neg Q) \subseteq (\neg Q) \circ C^+$$

  Entonces por inducción sobre la estructura de la cerradura *cd*, tal que por el Lema 4.34 se satisfacen el caso base y el paso inductivo.

- $\neg Q \circ C^+ \subseteq (C \circ \neg\alpha?)^+ \circ \neg Q$

  Por el Corolario 2.4 basta demostrar que:

  $$ci((\neg Q) \circ C, C) \subseteq (C \circ \neg\alpha?)^+ \circ \neg Q$$

  Haciendo inducción sobre la cerradura ci, tal que por el Teorema 2.4 se satisfacen el caso base y el paso inductivo.

$\boxed{\lambda}$

Esto brinda una relación binaria que es igual a la interpretación de la expresión $I(\text{desc} :: [\text{not}(\text{anc} :: \alpha)])$, es decir:

$$I(\text{desc} :: [\text{not}(\text{anc} :: \alpha)]) = (C \circ \neg\alpha?)^+ \circ \neg Q \tag{4.38}$$

Con esto, solamente falta considerar que la expresión es una ruta absoluta. Para esto observemos que $\neg Q$ es correflexiva, entonces al componerlo con la relación $Id \times Top?$ causará que la primera sea proyectada.

Lema 4.35. *Si* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$*, entonces:*

$$(\neg Q) \circ (Id \times Top?) = Id \times Top?$$

Demostración. Por la correflexividad de ambas relaciones y el Lema 2.5 basta demostrar que:

$$(\neg Q) \cap (Id \times Top?) = Id \times Top?$$

La inclusión de izquierda a derecha se cumple por definición.

Mientras que la otra inclusión se deberá demostrar que $Id \times Top? \subseteq \neg Q$, que por la correflexividad de $Id \times Top?$ y doble negación basta demostrar que:

$$\neg\neg Id \times Top? \subseteq \neg Q$$

Y utilizando la conexión de Galois de la diferencia y la unión, y las leyes de De Morgan basta demostrar que:

$$id \subseteq \neg((Id \times Top?) \cap Q)$$

Dado que $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$ es válida el resultado. $\boxed{\lambda}$

Finalmente, tenemos el siguiente razonamiento ecuacional:

$$
\begin{aligned}
& I(/\text{desc} :: [\text{not}(\text{anc} :: \alpha)]) = \\
& \text{Por la Ecuación 4.38} \\
& \beta? \circ (C \circ \neg\alpha?)^+ \circ \neg Q \circ (Id \times Top?) \circ (up_L \cup up_R)^* = \\
& \text{Por la Ecuación 4.35} \\
& \beta? \circ (C \circ \neg\alpha?)^+ \circ (Id \times Top?)(up_L \cup up_R)^*
\end{aligned}
$$

Con esto finalizan el capítulo y el trabajo, las siguientes páginas contienen conclusiones, trabajo a futuro y un anexo indicando los elementos de `COQ` no estudiados a fondo en este trabajo.

Capítulo 5

Conclusiones y trabajo a futuro

5.1. Conclusiones

Este trabajo tuvo como meta verificar formalmente la teoría dada en el artículo *Calculating Tree Navigation with Symmetric Relational Zipper* de Yuta Ikeda y Susumu Nishimura [16], utilizando el asistente de pruebas `COQ`, mostrando así los beneficios de la prueba asistida por computadora en el proceso de verificación formal.

La verificación formal mostró que el artículo contiene un error en la demostración del resultado:

Teorema 5.1. *Las relaciones binarias* $C = dn_R^* \circ dn_L$ *y* $Q = (dn_L \circ \alpha? \circ up_L)[up_L, up_R]^*$ *satisfacen la siguiente inclusión:*

$$\neg Q \circ C \subseteq C \circ \neg\alpha? \circ \neg Q \tag{5.1}$$

Ya que Ikeda y Nishimura establecen el siguiente cálculo:

$$\begin{aligned} & \neg Q \circ C \subseteq C \circ \neg\alpha? \circ \neg Q \\ \equiv\ & \text{CLOSURE-UEP} \\ & \neg Q \circ C \subseteq \nu X.(dn_L \circ \neg\alpha? \circ \neg Q \cup dn_R \circ X) \end{aligned}$$

Y la relación binaria $C \circ \neg\alpha? \circ \neg Q$ tiene un esqueleto de la forma $R^* \circ S$, mientras que CLOSURE-UEP se aplica a relaciones binarias cuyo esqueleto sea de la forma $S \circ R^*$. Por lo que el presente trabajo redefine la ecuación CLOSURE-UEP de la siguiente forma:

$$R^* \circ S = coCD(R, S)$$

Que traducido al estilo punto fijo del teorema de Knaster-Tarski es:

$$R^* \circ S = \nu X.(S \cup R \circ X)$$

A pesar de este error, la ecuación 5.1 no queda invalidada. Por lo que se concluye que podría haber sido un error de redacción.

Por otra parte, el reto de formalizar de tal forma que COQ acepte las definiciones establecidas en el estilo de punto fijo dados por el teorema de Knaster-Tarski no es tan transparente como se esperaba. Ikeda y Nishimura definen la cerradura simétrica de la siguiente forma:

$$(R)[S_1, \ldots, S_n]^* = \mu X.(R \cup (\bigcup_{i=1}^{n}(S_i^{\smile} \circ X \circ S_i))) \tag{5.2}$$

No es posible implementar esta cerradura de manera directa como las otras cerraduras trabajadas, ya que, la variable de punto fijo X se encuentra dentro de una función (la unión), la cual es complicada de implementar debido al uso de los índices i y n. Esto se solucionó parafraseando la definición: Cualquier par (x, y) que pertenezca a la cerradura simétrica, debe cumplir que $(x, y) \in R$ o bien debe existir una relación binaria S_i que pertenezca a la lista $[S_1, \ldots, S_n]$ y que $(x, y) \in S_i^{\smile} \circ (R)[S_1, \ldots, S_n]^* \circ S_i$. Esta definición de la cerradura simétrica fue aceptada por el asistente y es aportación nuestra. Más aún se demostró que esta nueva definición cumple las propiedades dadas por Ikeda y Nishimura. De este proceso se concluye que a pesar de que una traducción directa de una definición dada en la teoría sea difícil o casi imposible de obtener, es posible obtener los mismos resultados mediante una definición alternativa.

Finalmente, Ikeda y Nishimura al agregar el predicado de negación al lenguaje xPath se vieron en la necesidad de definir un complemento para relaciones binarias correflexivas. El reto en esta ocasión fue darle seguimiento a los resultados respecto a la negación de relaciones binarias, ya que ellos utilizan en cierta parte del análisis un resultado de cerraduras definidas coinductivamente. COQ cuenta con un mecanismo para las definiciones coinductivas, sin embargo, al existir poca información introductoria al tema de coinducción utilizando el asistente, resultó un reto bastante interesante demostrar la ecuación CLOUSURE-UEP, ya que la táctica `cofix` no resultó suficiente para demostrar los teoremas y se recurrió a una traducción, y demostración, del resultado $\nu-$inducción del teorema de Knaster-Tarski en el asistente.

5.2. Trabajo a futuro

Hay varias líneas que se pueden investigar a partir de este proceso de verificación:

1. Definir más tácticas de automatización para que el asistente haga inferencias con la confianza plena de que resultarán correctas.[1]

2. Crear unas notas introductorias al tema de coinducción utilizando COQ, existen varios escritos y presentaciones del tema, sin embargo, se asumen conocimientos que dificilmente tendría un alumno de licenciatura.

3. Ikeda y Nishimura mencionan mecanismos para la traducción de cerraduras a funciones, habría que analizar la posibilidad de implementar este mecanismo para generar una implementación certificada de la interpretación lenguaje xPath.

[1] Se hizo un intento, sin embargo si no se piensa bien la automatización el asistente puede entrar en ciclo. Por ejemplo indicarle que automáticamente utilice la conmutatividad de un operador.

4. En el artículo de Dan Olteanu *et al.* [31] se encuentran más ecuaciones sobre expresiones xPath utilizando una teoría distinta a la propuesta por Yuta Ikeda y Susumu Nishimura en [16], sería interesante verificar de manera formal dichas ecuaciones utilizando el enfoque de sistemas de juicios.

5. En [16] se menciona que con los resultados del artículo de Connor McBride [25], se puede definir una interpretación más cercana a la estructura de las expresiones xPath. Habría que verificar formalmente esta aseveración.

6. En la investigación de la estructura del zipper, se supo de la existencia de una estructura llamada *web*. «*Cuyo propósito es el mismo al del zipper, con la diferencia que es paramétrico en el tipo del término subyacente* » (Ralf Hinze y Johan Jeuring, 2001, p. 1) [11]. Resultaría interesante analizar la estructura y ver sus posibles aplicaciones.

Apéndice A

COQ

El asistente de pruebas COQ es un sistema interactivo creado para la verificación formal de resultados y la generación de programas funcionales certificados. En este trabajo, utilizamos dicho sistema para la verificación formal de la teoría descrita por Yuta Ikeda y Susumu Nishimura en [16].

Ya sea para la verificación formal o la certificación de programas funcionales, COQ cuenta con una interfaz gráfica, llamada CoqIDE, para interactuar con el usuario.

Figura A.1: Interfaz gráfica CoqIDE [18].

Observemos que la interfaz cuenta con tres áreas:

- Usuario. Esta área corresponde a la parte izquierda, en la cual el usuario ingresará la teoría y/o el programa con el que trabajará.

- Metas. Esta área corresponde a la parte superior derecha, en la cual se mostrarán las metas. Es decir, demostraciones que están pendientes.
- Mensajes. Esta área corresponde a la parte inferior derecha, en la cual COQ da las notificaciones al usuario. Por ejemplo, *Lemma has defined*.

Para revisar la forma de trabajar con el asistente, sugerimos revisar los trabajos que se mencionan en la siguiente sección.

A.1. Resultados usando a COQ

En los últimos años se ha usado y estudiado el asistente de pruebas COQ en la licenciatura de Ciencias de la Computación impartida en la Facultad de Ciencias, UNAM. Lo cual ha generando desde seminarios como *Semántica y Verificación* y *Razonamiento Automatizado* [28], hasta tesis de verificación formal de resultados en distintas áreas de la computación teórica. Por mencionar algunas:

- *Teoría de categorías aplicadas en el análisis de mónadas* de Cenobio Vázquez [34].
- *Deducción natural en lógica modal* de Selene Pilares [22].
- *Verificación formal de compiladores* de Ángel Zúñiga [36].

Los trabajos citados brindan una breve introducción al asistente de pruebas COQ, además de existir bastantes recursos para uso del asistente. Por ejemplo:

- *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant* de Adam Chlipala [5].
- *Interactive Theorem Proving and Program Development: Coq´Art The Calculus of Inductive Constructions* de Yves Bertrot y Pierre Castran [3].
- *Documentation: The Coq Proof Assistant* del Instituto Nacional de Investigación en Ciencias de la Computación y Automatización (INRIA, por sus siglas en francés) [17].

Dado que estas fuentes son suficientes para poder trabajar de manera básica con el asistente, en este trabajo no damos ni introducción ni uso básico del asistente. A continuación una breve descripción técnica del desarrollo en COQ, de este trabajo.

A.2. Desarrollo e implementación

El desarrollo del trabajo consistió en el siguiente procedimiento:

1. Hicimos el primer intento de la verificación formal del artículo de Yuta Ikeda y Susumu Nishimura [16].
2. Describimos en un documento escrito, los resultados del paso anterior.

3. Afinamos las demostraciones del paso 1, para escribir menos en el paso 2.

Con lo que obtuvimos los siguientes resultados:

Nombre del programa	Zipper_Rel
Número de módulos	6
Total de líneas de código	5426
Un archivo Makefile	

Figura A.2: Ficha técnica de la implementación.

En caso de ser la primera vez que se usará el código, se deberá compilar utilizando las siguientes órdenes, en la línea de comandos para sistemas Linux:

Código A.1 Primera compilación de código COQ.

```
coq_makefile -f _CoqProject -o Makefile

make
```

Si el código ya fue compilado pero se hicieron modificaciones en cualquiera de los archivos tipo v, entonces solamente hay que utilizar las siguientes órdenes:

Código A.2 Primera compilación de código COQ.

```
make clean

make
```

En caso de agregar mas módulos o bibliotecas se sugiere seguir el siguiente manual:

http://blog.zhenzhang.me/2016/09/19/coq-dev.html

En las siguientes secciones describimos: la creación de tácticas para la automatización de ciertas pruebas y la creación de definiciones coinductivas.

A.3. Creación de tácticas con Ltac

Esta sección comenzará con una de las definiciones más fuertes implementadas en COQ: La igualdad. Dado que el núcleo del asistente es una implementación de una teoría de tipos llamada *cálculo de construcciones inductivas* [3], la igualdad está definida de la siguiente forma:

Definición A.1. *Dos elementos x, y son iguales si y sólo si son sintácticamente iguales.*

Decimos que esta igualdad es demasiado fuerte ya que, ciertas comparaciones resultan inadecuadas. Por ejemplo, si simulamos conjuntos con listas, entonces considerando la lista `[1,2,3]` observamos que es distinta a la lista `[1,3,2]`. La igualdad previamente descrita indica que este razonamiento es correcto pero ante la teoría de conjuntos es incorrecto.

Particularmente en este trabajo, queremos verificar cuándo dos relaciones binarias son iguales. La teoría de conjuntos establece la siguiente definición de igualdad sobre relaciones binarias [2].

Definición A.2. *Sea A un conjunto y R, S dos relaciones binarias sobre A. Decimos que R es igual a S si se cumple la siguiente sentencia:*

$$R \subseteq S \text{ y } S \subseteq R$$

Donde $\subseteq$ es la relación de contención entre relaciones binarias.

Obsérvese que esta definición no es puramente sintáctica, por lo que no es posible usar la igualdad A.1. Para resolver esto, necesitamos definir una equivalencia entre relaciones binarias e indicarle al asistente que asuma que dicha equivalencia es análoga a la igualdad sintáctica.

```
Notation "R ⊆ S" := (inclusion _ R S) (at level 80).

Notation "R ≡ S" := (same_relation _ R S) (at level 80).

Axiom eqRelEq: ∀ (A: Type) (R S:relation A), R = S ↔ (R ≡ S).
```

Figura A.3: Definición de igualdad de relaciones binarias en COQ.

Un problema de definir de esta forma la igualdad entre relaciones binarias es, cada vez que necesitemos demostrar la igualdad entre dos relaciones, tendríamos que ejecutar los siguientes comandos: `apply eqRelEq`, `unfold same_relation`, `split`, `unfold inclusion` e `intros`. Aún utilizando el operador de secuencia (`;`), este procedimiento es lo suficientemente *mecánico* que resulta mejor opción automatizarlo. Para realizar esta tarea, COQ cuenta con un lenguaje (externo a GALLINA) llamado $\mathcal{L}_{tac}$ que permite la combinación de tácticas a través de una casa de patrones sobre la meta. Por otra parte, el lenguaje GALLINA tiene implementado a $\mathcal{L}_{tac}$ como un comando llamado `Ltac`, dicho comando es la implementación de este lenguaje [5], [7].

En otras palabras, si una meta cumple cierta sintaxis, entonces le indicamos al asistente que cada vez que usemos una táctica definida a través de `Ltac`, un conjunto de tácticas serán aplicadas de manera automática. Por ejemplo, en nuestro caso particular de la igualdad entre relaciones binarias, esta nueva táctica la implementamos de la siguiente forma:

```
Ltac destruct_eqRel :=
  match goal with
  | [ ⊢ ? R = ?S ] ⇒ apply eqRelEq;
                     unfold same_relation;
                     split;unfold inclusion;intros
  end.
```

Figura A.4: La táctica `destructEqRel`.

Al aplicar esta táctica a una meta de la forma `? R = ? S`, se generan dos metas:

- Si un par x pertenece a la relación R, habrá que demostrar que x pertenece a S.
- Si un par x pertenece a la relación S, habrá que demostrar que x pertenece a R.

Ahora un caso mas complicado: En las relaciones inducidas, si un par (x, y) pertenece a una relación inducida A, entonces $x =_A y$. Esto quiere decir que para demostrar que estas relaciones son correflexivas, debemos ejecutar la siguiente secuencia de tácticas: `intros;unfold coref;unfold inclusion;intros` para desdoblar la definición de una relación inducida y las tácticas `destruct H;rewrite H;reflexivity` para demostrar que $(x, y) \in Id$. Esto lo podemos encapsular en dos tácticas: La primera, que desdoble hasta las expresiones mínimas y la segunda, que pruebe la reflexividad de la igualdad sobre los elementos x e y. Esto lo implementamos de la siguiente forma:

```
Ltac destruct_coref :=
  match goal with
  | [ ⊢ ∀ A, coref ?B ] ⇒
    intro;unfold coref;unfold inclusion;intros
  end.

Ltac prueba_coref :=
  match goal with
  | [ H : ?P ⊢ id ?x ?y ] ⇒ destruct H;rewrite H;
                            reflexivity
  end.
```

Figura A.5: La táctica `destructEqRel`.

Observamos que el asistente no solamente caza patrones en la conclusión, también es capaz de cazar patrones en las hipótesis ya que, COQ considera tanto las hipótesis y la conclusión como una única meta a demostrar.

Con este ejemplo finalizamos la sección de creación de tácticas, existen mayores beneficios de este comando [5] pero no son de interés para este trabajo. En la siguiente sección describimos el mecanismo para dar definiciones coinductivas y como mostrar propiedades sobre estas utilizando la táctica `cofix`.

A.4. Coinducción

Una vez que hemos definido una estructura con un sistema de juicios coinductivos, la implementación al asistente es casi directa. Por ejemplo, la definición 2.30 corresponde a las secuencias infinitas de una relación R, este predicado lo podemos implementar de la siguiente forma:

```
CoInductive infiStar {A:Type} ( R:relation A): A → Prop :=
| infiIntro (x y:A): R x y → infiStar R y → infiStar R x.
```

Figura A.6: Definición coinductiva.

Por otra parte, uno de los grandes beneficios de COQ es la creación automática del principio de inducción utilizando el comando `Inductive`. Sin embargo, al definir entes coinductivos utilizando el comando `CoInductive`, el asistente no tiene el mismo soporte para definir de manera automática un «principio de coinducción». Lo que brinda es una táctica llamada `cofix`, la cual solicita a la estructura como copunto fijo para su demostración.

Observése que si el copunto fijo es erroneo, COQ lo rechazará de forma inmediata y con justa razón. En caso de aceptar una prueba donde la estructura no cumple las condiciones de ser un copunto fijo las demostraciones resultarían en utilizar como hipótesis lo que se quiere demostrar.[1]

[1] Es lo que la documentación oficial titula *respetar la guardia*.

Bibliografía

[1] Aarts, C., Backhouse, R. C., Boiten, E. A., Doornbos, H., van Gasteren, N., van Geldrop, R., Hoogendijk, P. F., Voermans, E., y van der Woude, J. (1995). Fixed-point calculus. *Information Processing Letters*, 53(3):131 – 136. The calculational method. 71

[2] Amor Montaño, J. A. (2013). *Teoría de conjuntos para estudiantes de ciencias*. Las prensas de ciencias, Ciudad de México,México. 59, 154

[3] Bertot, Y. y Castran, P. (2010). *Interactive Theorem Proving and Program Development: Coq´Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated. 152, 153

[4] Bray, T. y et al. (2006). Extensible markup language (xml) 1.1 (second edition). *W3C Recommendation*. Disponible en: https://www.w3.org/TR/2006/REC-xml11-20060816/. Consultado: 6/11/2017. 27, 28, 34

[5] Chlipala, A. (2013). *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press. 152, 154, 156

[6] Cormen, T. H., Leiserson, C. E., Rivest, R. L., y Stein, C. (2009). *Introduction to Algorithms, Third Edition*. The MIT Press. 116

[7] Delahaye, D. (2000). A tactic language for the system coq. En Parigot, M. y Voronkov, A., editores, *Logic for Programming and Automated Reasoning*, pp. 85–95, Berlin, Heidelberg. Springer Berlin Heidelberg. 154

[8] Givant, S. (2006). The calculus of relations as a foundation for mathematics. *Journal of Automated Reasoning*, 37(4):277–322. Disponible en: https://doi.org/10.1007/s10817-006-9062-x. 58, 60, 61

[9] González Huesca, L. d. C. (2007). Coinducción: de la teoría de categorías a la programación funcional. 90

[10] Harper, P. R. (2012). *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA. 72

[11] Hinze, R. y Jeuring, J. (2001). Weaving a web. *J. Funct. Program.*, 11(6):681–689. 149

[12] Huet, G. (1997). The zipper. *J. Funct. Program.*, 7(5):549–554. IX, 15, 17, 22

[13] Hughes, J. (1989). Why functional programming matters. *Comput. J.*, 32(2):98–107. 1

[14] Huntington, E. V. (1933). New sets of independent postulates for the algebra of logic, with special reference to Whitehead and Russell's Principia Mathematica. *Trans. Am. Math. Soc.*, 35:274–304. 61

[15] Huntington, E. V. (2016). Sets of independent postulates for the algebra of logic. *Transactions of the American Mathematical Society*, 5(3):288–309. 60, 61

[16] Ikeda, Y. y Nishimura, S. (2011). Calculating tree navigation with symmetric relational zipper. En *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '11, pp. 101–110, Nueva York, NY, EUA. ACM. v, ix, ix, xviii, 22, 24, 59, 62, 64, 69, 72, 74, 77, 78, 84, 98, 103, 104, 109, 111, 113, 115, 116, 120, 125, 143, 147, 149, 151, 152

[17] INRIA (2017). Documentation: The coq proof assistant. Disponible en: https://coq.inria.fr/documentation. 152

[18] INRIA (2018). Coqide screenshots. Disponible en: https://coq.inria.fr/coqide-screenshots. xvi, 151

[19] Jacobson, N. (2009). *Basic Algebra I: Second Edition*. Basic Algebra. Dover Publications. 60

[20] Lauro Aguilar, M. D. A. (2015). Manufactura de tipos de datos mediante multiconjuntos. 3, 5

[21] Leroy, X. (2015). Colloquium d'informatique. Disponible en: http://video.upmc.fr/differe.php?collec=S_C_colloquium_lip6_2012&video=19. 115

[22] Linares Arévalo, P. S. (2015). Deducción natural: Una implementación en coq. 152

[23] Lipovaca, M. (2011). *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA. 12

[24] Martin-Löf, P. E. R. (1983). On the meanings of the logical constants and the justifications of the logical laws. Disponible en: http://www.cs.cornell.edu/courses/cs671/1999fa/martin.html. 57, 70

[25] McBride, C. (2001). The Derivative of a Regular Type is its Type of One-Hole Contexts. Disponible en: http://www.cs.nott.ac.uk/~ctm/diff.ps.gz. 3, 5, 20, 149

[26] Menabrea, L. F. (1843). Analytical engine. Disponible en: https://www.fourmilab.ch/babbage/sketch.html. 103

[27] Miranda Perea, F. E. (2017a). Lenguajes de programación: Notas de clase. Disponible en: http://lya.fciencias.unam.mx/favio. 3

[28] Miranda Perea, F. E. (2017b). Seminarios: Razonamiento automatizado y semántica y verificación. Disponible en: http://lya.fciencias.unam.mx/favio/cursos.html. 152

[29] Miranda Perea, F. E. y Viso Gurovich, E. (2016). *Matemáticas discretas*. La prensas de las ciencias, Ciudad de México, México. x, 70, 72, 84

[30] Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA. IX, 1, 2

[31] Olteanu, D. y et al. (2002). *XPath: Looking Forward*, pp. 109–127. Springer Berlin Heidelberg, Berlin, Heidelberg. IX, 34, 115, 149

[32] Robie, J. y et al. (2017). Xml path language (xpath) 3.1. *W3C Recommendation*. Disponible en: https://www.w3.org/TR/xpath-31/#id-introduction. Consultado: 9/11/2017. 35

[33] Sangiorgi, D. (2011). *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA. 71, 95

[34] Vázquez Reyes, C. M. (2016). Mónadas en la programación funcional: Una prueba de su equivalencia con las ternas de Kleisli. 58, 152

[35] Warwick, C., Warwick, C., Terras, M., y Nyhan, J. (2012). *Digital Humanities in Practice*. Facet Publishing. 27

[36] Zúñiga Chávez, A. F. (2016). Un compilador correcto verificado de mini-ml a la máquina secd en coq. 152