Sistemas de reescritura desde el punto de vista de la programación funcional

Miguel Ángel Porras Naranjo

Sistemas de reescritura desde el punto de vista de la programación funcional

Miguel Ángel Porras Naranjo

Memoria presentada como parte de los requisitos para la obtención del título de Grado en Matemáticas por la Universidad de Sevilla.

Tutorizada por

Prof. José A. Alonso
Prof. María José Hidalgo

Índice general

Introducción

La **reescritura** es una técnica que consiste en reemplazar términos de una expresión con **términos equivalentes**. Por ejemplo, dadas las reglas $x * 0 \Rightarrow 0$ y $x + 0 \Rightarrow 0$ pueden usarse para simplificar la siguiente expresión:

$$x + (\underline{x * 0}) \longrightarrow \underline{x + 0} \longrightarrow x$$

Pero no sólo se puede aplicar para ecuaciones. Se puede usar para demostrar enunciados. Por ejemplo, dadas las reglas $s(M) \leq s(N) \Rightarrow M \leq N$ y $0 \leq N \Rightarrow \texttt{True}$ pueden aplicarse a:

$$s(0) \leq s(x) \longrightarrow 0 \leq s(x) \longrightarrow \texttt{True}$$

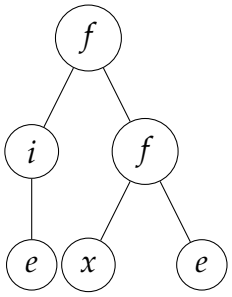
Donde $s(x)$ es la función sucesor de x.

En definitiva, podemos aplicar reescritura a cualquier tipo de expresión (ya sea lógica, ecuacional, etc) siempre y cuando se pueda hacer una **sustitución**. Esta sustitución se hace de término a término.

Sin embargo, debemos formalizar qué entendemos por término. Un **término** estará construido por **variables, símbolos constantes y símbolos de funciones**. Al estar constituidos por funciones, una representación natural de los términos se realiza mediante árboles. Por ejemplo el término $f(i(e), f(x, e))$ se representa como,

Donde i, f representa un símbolo de función de aridad 1 y 2 respectivamente, e representa un símbolo constante y x representa una variable.

Los anteriores ejemplos también se pueden escribir como términos,

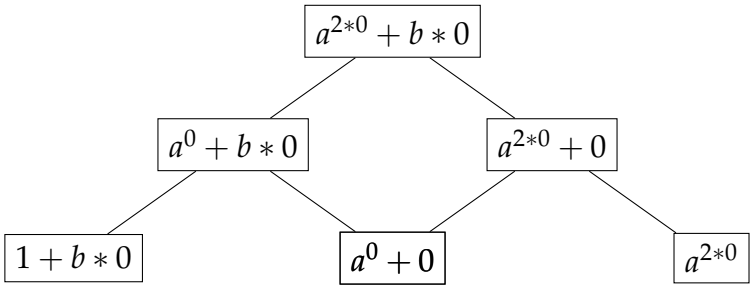


$$x + (x * 0) \longrightarrow +(x, *(x, 0))$$

$$s(0) \leq s(x) \longrightarrow \leq (s(0), s(x))$$

Un **sistema de reescritura de términos** es un conjunto de sustituciones término a término. A estas sustituciones las llamaremos reglas.

Estos conjuntos de reglas determinan ciertas propiedades según los elementos que contengan. Por ejemplo,



Este diagrama de árbol representa para cada rama una aplicación de las distintas reglas. Podemos hacernos dos preguntas fundamentales, si este árbol será **finito** y si **todas las hojas son la misma**; es decir, siguiendo cualquier rama, llegar hasta el mismo término. Estas dos preguntas son los principales problemas de la reescritura.

Analizar si este árbol es finito es equivalente a preguntarse si en algún momento podemos dejar de aplicar las reglas. Por ejemplo, para el término $s(1) + s(0)$ y para la regla $s(3) \Rightarrow 4$, no podemos seguir sustituyendo términos, por tanto diremos que el sistema **termina**.

Es importante observar que no siempre tener un mayor número de reglas es mejor. Por ejemplo, para las reglas $x + 0 \Rightarrow x$ y $x \Rightarrow x + 0$;

$$x + 0 \longrightarrow x \longrightarrow x + 0 \longrightarrow \dots$$

Otro de los problemas que tenemos que tratar es el de ver si el árbol termina en una misma hoja. Si esta situación ocurre en nuestro sistema diremos que es **confluente**.

Para analizar si un sistema es confluente o no debemos estudiar el momento en el que se crea una ramificación del árbol. La idea intuitiva es ver si la sustitución que se hace en cada rama se solapa entre sí. A estos puntos los llamaremos **pares críticos**.

También podemos modificar el conjunto reglas de los reescritura para que se verifiquen estas dos propiedades. A este proceso se le llama **completación**. Los cambios que se realizan al conjunto de reglas van desde eliminar alguna regla, hasta añadir propias.

Como es de esperar, el algoritmo de completación debe tener en cuenta los pares críticos. Una primera idea de implementación sería modificar las reglas hasta no tener ningún par crítico.

Sin embargo el cálculo de pares críticos es muy costoso si hablamos de grandes sistemas de reescritura. Por ello introduciremos **el algoritmo de Huet**, que pese a usar pares críticos en el proceso, reduce la cantidad de cálculos a realizar.

A continuación, describiré brevemente los contenidos de cada capítulo.

En el capítulo 1, introduciremos los **sistemas de reducción abstractos** y definiremos algunas de sus características fundamentales, algunas de ellas se volverán a tratar con mayor profundidad en capítulos posteriores.

En el capítulo 2, estudiaremos las definiciones de **términos y sustituciones**, así como sus características. Además, relacionando términos y sustituciones, daremos una caracterización de $\overset{*}{\longleftrightarrow}$,

En el capítulo 3, formalizaremos la definición de **sistema de reescritura de términos** y estudiaremos el problema de **unificación** para términos.

En el capítulo 4, resolveremos el problema de decidir si un sistema de reescritura es **terminante**. Veremos que, en general, el problema es indecidible. Por ello, le daremos varios enfoques según las propiedades que tenga el sistema para poder decidir la terminación.

En el capítulo 5, estudiaremos la **confluencia** para los sistemas de reescritura. Probaremos que el problema de decidir si un sistema es confluente es indecidible. Sin embargo, a partir del estudio de pares críticos y ortogonalidad, podremos dar varios resultados sobre la confluencia.

Por último, en el capítulo 6, enunciaremos dos algoritmos de **completación**. Para ambos, estudiaremos su funcionamiento y si son correctos.

Capítulo 1

Sistemas de reducción abstractos

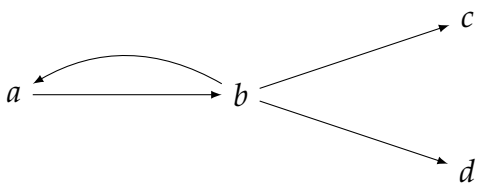
En este capítulo, definiremos que entendemos como un sistema de reducción abstracto, y a partir de este, daremos la definición de orden, que sera imprescindible para capítulos posteriores. También veremos en qué consiste la inducción bien fundada, el orden lexicográfico y un poco de teoría de multiconjuntos. Estas dos últimas incluirán una implementación en Haskell que se puede encontrar en `Orden.hs`

Equivalencia y reducción

El objetivo principal de esta sección es definir toda la estructura de los sistemas de reducción abstractos.

Definición 1.1. *Un **sistema de reducción abstracto** es un par $(A, \to)$ donde la reducción $\to$, es una relación en el conjunto A; es decir, $\to \subseteq A \times A$.*

Para simplificar la notación, si $(a, b) \in \to$, escribiremos $a \to b$. Un sistema de reducción abstracto se puede entender como un grafo dirigido. Por ejemplo,



Definición 1.2. *Diremos que b es una* ***reducción*** *de a si* $a \to a_0 \to a_1 \to \cdots \to b$.

En el ejemplo anterior, c es una reducción de a. Pero d no es una reducción de c

Definición 1.3. *Diremos que a y b son* ***equivalentes****, y lo denotaremos por* $a \overset{*}{\leftrightarrow} b$*, si b es una reducción de a y a es una reducción de b.*

En el ejemplo, a y b son equivalentes. También trabajaremos con el concepto de composición.

Definición 1.4. *Dadas dos relaciones* $R \subseteq A \times B$*, y* $S \times B \times C$*, definimos su* ***composición*** *como*

$$R \circ S := \{(x,z) \in A \times C | \exists y \in B.(x,y) \in R \wedge (y,z) \in S\}$$

A partir de una reducción $\to$ se pueden calcular diferentes clausuras,

Definición 1.5. *A partir de la definición de reducción, consideraremos las siguientes nociones,*

$\overset{0}{\to}$	$:=$	$\{(x,x) \mid x \in A\}$	***Identidad***
$\overset{i+1}{\longrightarrow}$	$:=$	$\overset{i}{\to}$ *ó* $\to$	***(i+1)-plegado, i*** ≥ 0
$\overset{+}{\to}$	$:=$	$\cup_{i>0} \overset{i}{\to}$	***Clausura transitiva***
$\overset{*}{\to}$	$:=$	$\overset{+}{\to} \cup \overset{0}{\to}$	***Clausura transitiva reflexiva***
$\overset{=}{\to}$	$:=$	$\to \cup \overset{0}{\to}$	***Clausura reflexiva***
$\overset{-1}{\longrightarrow}$	$:=$	$\{(y,x) \mid x \to y\}$	***Inversa***
$\leftarrow$	$:=$	$\overset{-1}{\longrightarrow}$	***Inversa***
$\leftrightarrow$	$:=$	$\to \bigcup \leftarrow$	***Clausura simétrica***
$\overset{+}{\leftrightarrow}$	$:=$	$(\leftrightarrow)^+$	***Clausura transitiva simétrica***
$\overset{*}{\leftrightarrow}$	$:=$	$(\leftrightarrow)^*$	***Clausura reflexiva transitiva simétrica***

Ejemplo 1.1. A partir de la reducción: Las relaciones según la definición anterior son,

A continuación, veremos una serie de definiciones que serán imprescindibles a la hora de estudiar los sistemas de reescritura.

Definición 1.6. *Diremos que P es la* ***clausura*** *de R si P es el menor conjunto con la propiedad P que contiene a R.*

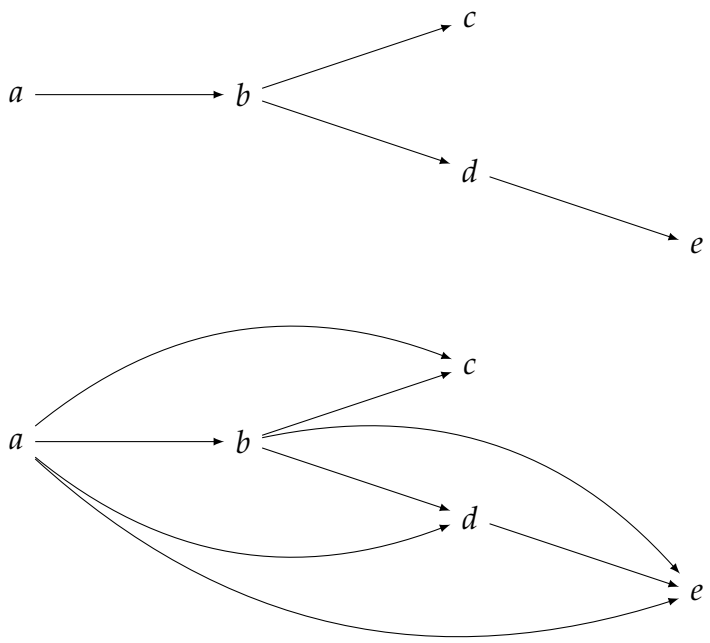


Figura 1.1: $\xrightarrow{+}$

Definición 1.7. *Diremos que x es **reducible** si existe un y tal que $x \to y$.*

Definición 1.8. *Diremos que x esta en su **forma normal** si x no es reducible.*

Definición 1.9. *Diremos que y es un **sucesor** de x si $x \xrightarrow{+} y$*

Definición 1.10. *Diremos que x e y son **confluentes** si hay algún z tal que $x \xrightarrow{*} z \xleftarrow{*} y$. Lo denotaremos por $x \downarrow y$.*

Definición 1.11. *Llamaremos a una reducción $\to$,*

Church-Rosser	*si $x \xleftrightarrow{*} y \Leftrightarrow x \downarrow y$*
confluente	*si $y_1 \xleftarrow{*} x \xrightarrow{*} y_2 \Rightarrow y_1 \downarrow y_2$*
semi-confluente	*si $y_1 \leftarrow x \xrightarrow{*} y_2 \Rightarrow y_1 \downarrow y_2$*
terminante	*si no hay una cadena infinita $a_0 \to a_1 \to \ldots$*
normalizado	*si cada elemento tiene forma normal*
convergente	*si es confluente y terminante*

La línea discontinua en los siguientes diagramas representa la existencia de una reducción. Por ejemplo, en Church Rosser nos indica que si $x \xleftrightarrow{*} y$, entonces $\exists z$ tal que $x \xleftrightarrow{*} z$, e $y \xleftrightarrow{*} z$

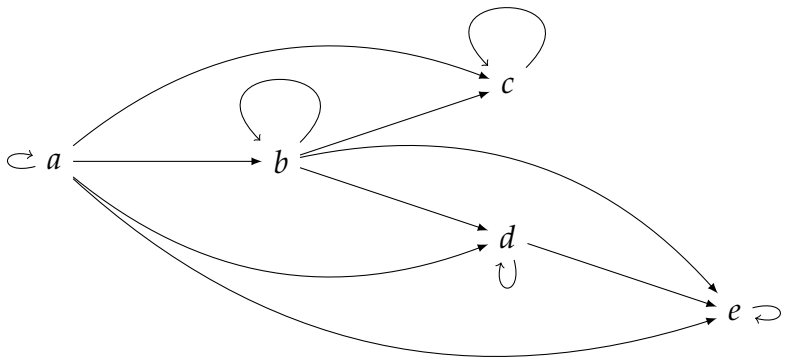


Figura 1.2: $\xrightarrow{*}$

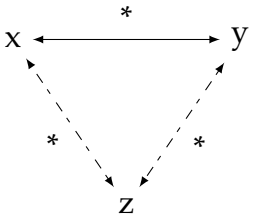


Figura 1.3: Church Rosser

Resultados básicos

A partir de las anteriores definiciones, vamos a demostrar una serie de propiedades que las relacionan entre sí.

Teorema 1.1. *Las siguientes condiciones son equivalentes,*

1. $\rightarrow$ *tiene la propiedad de Church–Rosser.*
2. $\rightarrow$ *es confluente.*
3. $\rightarrow$ *es semi–confluente.*

Demostración. La implicación $(1 \Rightarrow 2)$ se prueba suponiendo $y_1 \xleftarrow{*} x \xrightarrow{*} y_2$ por tanto, $y_1 \xleftrightarrow{*} y_2$, y por la propiedad de Church–Rosser $y_1 \downarrow y_2$.

La implicación $(2 \Rightarrow 3)$ es trivial. ∎

Corolario 1.1. Si $\rightarrow$ es confluente y $x \xleftrightarrow{*} y$ entonces,

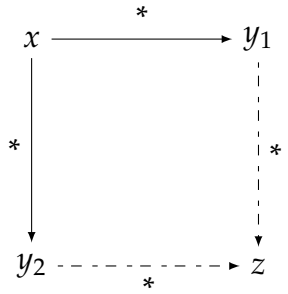


Figura 1.4: Confluente

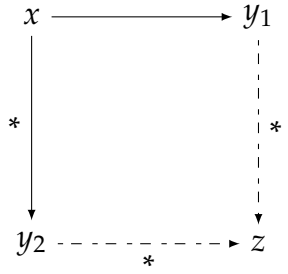


Figura 1.5: Semi-confluente

1. $x \xrightarrow{*} y$, si y es una forma normal.
2. $x = y$, si x e y están en su forma normal.

Teorema 1.2. *Si $\rightarrow$ es normalizado y confluente entonces $x \overset{*}{\leftrightarrow} y$ syss $x\downarrow = y\downarrow$. Es decir, cada elemento tiene una única forma normal.*

Inducción bien fundada

El principio de la inducción bien fundada (IBF) se trata de una generalización de la inducción en $(\mathbb{N}, >)$ para $(A, \rightarrow)$. Formalmente, la expresaremos como la siguiente regla:

$$\frac{\forall x \in A.(\forall y \in A.x \xrightarrow{+} y \Rightarrow P(y)) \Rightarrow P(x)}{\forall x \in A.P(x)}$$

Teorema 1.3. *Si $\rightarrow$ es terminante, entonces (IBF) se verifica.*

Teorema 1.4. *Si (IBF) se verifica, entonces $\rightarrow$ es terminante.*

Hemos llegado a que la definición de una relación terminante, es equivalente a comprobar que verifica (IBF). Esta propiedad será importante a la hora de resolver el problema de la terminación, en el capítulo 4.

Definición 1.12. *Llamaremos a una relación* $\to$*,*

Ramificación finita	*Si cada elemento tiene algunos sucesores finitos directos.*
Globalmente finita	*Si cada elemento tiene algunos sucesores finitos.*
Acíclico	*Si no hay ningún elemento* a*, tal que* $a \xrightarrow{+} a$*.*

Vamos a probar dos resultados de esta definición. Una relación globalmente finita es equivalente a decir que la relación $\xrightarrow{+}$ es una ramificación finita.

Lema 1.1. Una relación de ramificación finita es globalmente finita si es terminante.

Demostración. Usaremos la propiedad (IBF). Sea $\to$ una relación de ramificación finita y terminante. Vamos a probar que cada elemento del conjunto de los sucesores es finito. Como es cierto para cada sucesor directo por hipótesis, aplicando (IBF) llegamos a que su conjunto de sucesores es finito, y por tanto globalmente finito.

Lema 1.2. Una relación acíclica es terminante si es globalmente finita.

Resultados sobre la terminación

Uno de los métodos que estudiaremos para probar que $\to$ es terminante, se basa en (IBF). A partir de $(A, \to)$, definiremos un sistema de reducción abstracto terminante $(B, >)$. Definiremos una función $\varphi : A \to B$ que debe tener la propiedad de monotonía, (ie $x \to x' \Rightarrow \varphi(x) > \varphi(x')$). En esta sección demostraremos que si esta aplicación existe, $\to$ es terminante.

Fundamento: Sea $\to$ terminante. Si no lo fuera, habría una cadena $x_0 \to x_1 \to \ldots$, y esta induciría a $\varphi(x_0) > \varphi(x_1) > \ldots$.

Lema 1.3. Una reducción de ramificación finita $(A, \to)$ es terminante syss existe una aplicación $\varphi : A \to \mathbb{N}$, que verifique $x \to x' \Rightarrow \varphi(x) > \varphi(x')$.

Demostración. La implicación hacia la derecha es trivial. Al existir una aplicación φ que verifique la monotonía, nos da una cadena de x_i, y dicha cadena es trivialmente terminante.

Para la demostración de la implicación contraria, supondremos que $\varphi(x)$, es el número de sucesores de x. Debe ser finito por el lema 1.1. Como $\to$ es terminante, y por tanto acíclico (ya que no debe tener ningún elemento que $x \xrightarrow{+} x$), para todo $x \to x' \Rightarrow \varphi(x) > \varphi(x')$ (ie, x' tiene menos sucesores que x).

Orden lexicográfico

Definición 1.13. *Dados dos órdenes estrictos $(A, >_A)$ y $(B, >_B)$, **el producto lexicográfico** $>_{A\times B}$ en $A \times B$ se define por*

$$(x, y) >_{A\times B} (x', y') :\Leftrightarrow (x >_A x') \vee (x = x' \wedge y >_B y').$$

Lema 1.4. El producto lexicográfico de dos órdenes estrictos, es un orden estricto.

Teorema 1.5. *El producto lexicográfico de dos relaciones terminantes, es terminante.*

Demostración. Por reducción a lo absurdo. Supondremos que $>_A$ y $>_B$ es terminante y que la cadena $(a_0, b_0) > (a_1, b_1) > \dots$ es infinita. Como $>_A$ es terminante, debe existir un k, tal que $a_i = a_{i+1}$ con $i \geq k$. Pero esto indica que $>_B$ tiene una cadena infinita, y esto contradice la terminación de $>_B$.

A continuación, extrapolaremos la idea del producto lexicográfico de dos elementos, a n elementos.

Definición 1.14. *Dado n órdenes estrictos $(A_i, >_i)$, $i = 1, \dots, n$ su **producto lexicográfico** $>_{1..n}$ se define por*

$$(x_1, \dots, x_n) >_{1..n} (y_1, \dots, y_n) :\Leftrightarrow \exists k \geq n.(\forall i < k.x_i = y_i) \wedge x_k >_k y_k.$$

Notación 1.1. Si todos los $(A_i, >_i)$ son iguales, denotaremos el producto lexicográfico como $>_{lex}^n$.

Definición 1.15. *Dado un orden estricto $(A, >)$, el **orden lexicográfico** $>_{lex}^*$ en A^* se define por*

$$x >_{lex}^* y :\Leftrightarrow (|x| > |y|) \vee (|x| = |y| \wedge x >_{lex}^{|x|} y).$$

Donde $|x|$ es la longitud de x.

Lema 1.5. Si $>$ es un orden estricto, entonces $>^*_{lex}$ es un orden estricto. Si $>$ es terminante, $>^*_{lex}$ también lo es.

Otra variación del orden lexicográfico será la siguiente.

Definición 1.16. *Diremos que* $a >_{Lex} b$ *syss* b *es un prefijo de* a*, ó si* $a = x\alpha y$*,* $b = x\beta y$ *y* $\alpha > \beta$*.*

Ejemplo 1.2. Si $a > b$, $aaaa >_{Lex} aaa >_{Lex} abba$.

Lema 1.6. Si $>$ es un orden estricto, también lo es $>_{Lex}$.

Multiconjuntos y su orden

Una manera de construir órdenes terminantes es mediante los multiconjuntos. La definición totalmente informal es que se tratan de conjuntos con elementos repetidos. Formalmente, los definiremos como sigue,

Definición 1.17. *Un* ***multiconjunto*** M *sobre un conjunto* A *es una función* $M : A \to \mathbb{N}$*, tal que* $M(x)$ *es el número de veces que aparece repetido* x*. Un multiconjunto es finito si* $\{x \in A : M(x) > 0\}$ *es finito. Mediante* $\mathcal{M}(A)$ *se denotará el conjunto de todos los multiconjuntos finitos sobre* A*.*

Como uno de nuestros principales problemas es la terminación, nos interesan que los multiconjuntos con los que trabajemos sean finitos.

Ejemplo 1.3. $M = \{a, b, c, b\}$ es un multiconjunto, y es equivalente a $\{c, b, b, a\}$. También podemos expresarlo como $\{a \mapsto 1, b \mapsto 2, c \mapsto 1\}$

Definición 1.18. *Algunas operaciones básicas y relaciones en* $\mathcal{M}(A)$ *son,*

Elemento en M	$x \in M :\Leftrightarrow M(x) > 0$.
Inclusión	$M \subseteq N :\Leftrightarrow \forall x \in A. M(x) \leq N(x)$.
Unión	$(M \cup N)(x) := M(x) + N(x)$.
Diferencia	$(M - N)(x) := M(x) \dot{-} N(x)$, *donde* $n \dot{-} m$ *es* $n - m$ *si* $m \geq n$*,* 0 *en caso contrario.*

Ya podemos definir un orden para los multiconjuntos a partir de la anterior definición.

Definición 1.19. *Dado un orden estricto* $>$ *en un conjunto* A*, definimos el* ***orden de los multiconjuntos*** $>_{mul}$ *en* $\mathcal{M}(A)$ *como,* $M >_{mul} N$ *syss existe* $X, Y \in \mathcal{M}(A)$*, tales que,*

- $\emptyset \neq X \subseteq M$,
- $N = (M - X) \cup Y$, y
- $\forall y \in Y. \exists x \in X. x > y$.

Aunque la definición a primera vista puede resultar un poco compleja, lo que viene a decir es que si $x_1, \ldots, x_m$ son elementos de M, y $y_1, \ldots, y_m$ son elementos de N, X e Y vienen definidos como

$$\underbrace{x_1, \ldots, x_i}_{X} \underbrace{x_{i+1}, \ldots, x_m, \ldots, y_1, \ldots, y_n}_{Y}$$

y deben cumplir que para todo $y \in Y. \exists x \in X. x > y$. Notase que X e Y se pueden determinar de varias formas.

Lema 1.7. Si $>$ es un orden estricto, $>_{mul}$ también lo es.

La propiedad más importante de este capítulo es la siguiente.

Teorema 1.6. *$>_{mul}$ es terminante syss $>$ es terminante*

Lema 1.8. Si $>$ es un orden estricto y $M, N \in \mathcal{M}(A)$, entonces

$$M >_{mul} N \Leftrightarrow M \neq N \wedge \forall n \in N - M. \exists m \in M - N. m > n.$$

Implementación de los órdenes en Haskell

En esta sección vamos a implementar en el lenguaje de programación funcional Haskell, algunas de las definiciones que hemos estudiado en los anteriores apartados. En el apéndice A se definirán una lista de funciones básicas de la librería Data.List, en caso de que el lector no este afianzado con el lenguaje.

Hay que destacar el tipo de dato `Ordering`, que ya viene predefinido.

Prelude

```
data Ordering = EQ | LT | GT
```

Consideramos `EQ` si $a = b$, `LT` si $a < b$, y `GT` si $a > b$.

- `(ordLex r xs ys)` es el orden lexicográfico inducido por `r` sobre `xs` e `ys`.

```
>>> ordLex compare [1,2] [1,5]
LT
>>> ordLex compare [1,2] [1,2]
EQ
>>> ordLex compare [3,1] [2,4]
GT
>>> ordLex compare [1] [1,3]
LT
>>> ordLex compare [1,3] [1]
GT
>>> let r x y = compare (abs x) (abs y)
>>> ordLex compare [-4,3] [2,5]
LT
> ordLex r       [-4,3] [2,5]
GT
```

La definición de la función es,

```
ordLex :: (a -> b -> Ordering) -> [a] -> [b] -> Ordering
ordLex _ [] []  = EQ
ordLex _ [] _   = LT
ordLex _ _  []  = GT
ordLex r (x:xs) (y:ys) =
    case a of
      EQ -> ordLex r xs ys
      _  -> a
    where a = r x y
```

- `(ordenMulticonj r xs ys)` es la comparación de los multiconjuntos `xs` e `ys`. Por ejemplo,

```
>>> ordenMulticonj (M.fromOccurList [(7,2)])
  (M.fromOccurList [(7,2)])
EQ
>>> ordenMulticonj (lista2Multiconj [1,2,3,2,5])
  (lista2Multiconj [4,5,6,3])
LT
>>> ordenMulticonj (lista2Multiconj [4,5,6,3])
  (lista2Multiconj [1,2,3,2,5])
GT
```

La definición de la función es

```
ordenMulticonj :: Ord a => M.MultiSet a -> M.MultiSet a
                  -> Ordering
ordenMulticonj = compare
```

Resultados sobre la confluencia

Definición 1.20. *Diremos que una relación $\rightarrow$ es **localmente confluente** si*

$$y_1 \leftarrow x \rightarrow y_2 \Rightarrow y_1 \downarrow y_2$$

La definición de localmente confluente es mucho mas débil que la de confluente. No obstante, nos da un resultado importante en relación con la terminación.

Lema 1.9. Una relación terminante es confluente, si es localmente confluente.

Definición 1.21. *Una relación $\rightarrow$ es **fuertemente confluente** si,*

$$y_1 \leftarrow x \rightarrow y_2 \Rightarrow \exists z. y_1 \xrightarrow{*} z \xleftarrow{=} y_2$$

Hay que tener en cuenta, que para $y_1 \leftarrow x \rightarrow y_2$, debe verificarse tanto $y_1 \xrightarrow{*} z_1 \xleftarrow{=} y_2$, como $y_1 \xrightarrow{*} z_2 \xleftarrow{=} y_2$

Lema 1.10. Una relación fuertemente confluente es confluente.

Lema 1.11. Si $\xrightarrow{*}_1 = \xrightarrow{*}_2$, entonces $\rightarrow_1$ es confluente syss $\rightarrow_2$ es confluente.

Lema 1.12. Si $\rightarrow_1 \subseteq \rightarrow_2 \subseteq \xrightarrow{*}_1$ entonces $\xrightarrow{*}_1 = \xrightarrow{*}_2$

Corolario 1.2. Si $\rightarrow_1 \subseteq \rightarrow_2 \subseteq \xrightarrow{*}_1$ y $\rightarrow_2$ es fuertemente confluente, entonces $\rightarrow_1$ es confluente.

Definición 1.22. *Una relación $\rightarrow$ tiene la **propiedad del diamante** si*

$$y_1 \leftarrow x \rightarrow y_2 \Rightarrow \exists z. y_1 \rightarrow z \leftarrow y_2.$$

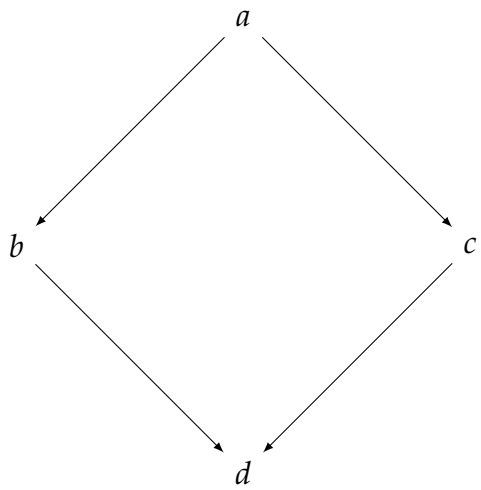


Figura 1.6: Propiedad del diamante

Capítulo 2

Formalización del concepto de términos

En este capítulo definiremos el concepto de términos y sustitución. Este estudio sobre el álgebra universal, es el principal objeto de estudio de este trabajo. Dedicaremos una sección para cada definición y propiedades básicas de cada una, incluyendo en ellas su implementación en Haskell. Dicho código se puede encontrar en `Terminos.hs`

Términos

En esta sección vamos a definir formalmente el concepto intuitivo que entendemos por término. Por ejemplo, si consideramos f una función de aridad 1, x una variable, entonces $f(x)$ es un término. Para formalizar qué funciones podemos usar en cada contexto y qué aridad tienen, introducimos el concepto de signatura.

Definición 2.1. *Una **signatura** Σ es una familia de conjuntos de símbolos de funciones $\{\Sigma_n\}$, donde para cada símbolo $f \in \Sigma_n$, f tiene aridad n. Los elementos $\Sigma^{(0)}$ se llaman símbolos constantes.*

Por ejemplo en álgebra, el concepto de signatura de grupo es $\Sigma_G = \{e, i, f\}$, donde e es la identidad con aridad 0, i es la inversa con aridad 1 y f es la suma con aridad 2.

Con esta idea, podemos introducir el concepto de términos.

Definición 2.2. *Sea Σ una signatura, y X un conjunto numerable de variables tal que, $\Sigma \cap X = \emptyset$. El* ***conjunto de los términos de primer orden*** *denotado por $T(\Sigma, X)$ de los Σ–términos en X se define inductivamente como,*

- $X \subseteq T(\Sigma, X)$
- *Si* $n \geq 0$, $f \in \Sigma^{(n)}$ *y* $t_1, \ldots, t_n \in T(\Sigma, X)$ *entonces* $f(t_1, \ldots, t_n) \in T(\Sigma, X)$

Obsérvese que una variable es un término. Por ejemplo con el Σ_G anterior, e, i, $f(e, i(e))$ son ejemplos de términos de Σ_G.

Para su implementación en Haskell necesitamos definir qué entendemos por una variable.

- En primer lugar, definimos los símbolos o nombres como,

```
type Nombre = String
```

- Los índices son números naturales.

```
type Indice = Int
```

- Los nombres de variables son pares formados por un nombre y un índice.

```
type Nvariable = (Nombre,Indice)
```

  Por ejemplo, x_3 se representa como `("x",3)`.

Como un término es una variable o un término compuesto, definimos el tipo de dato `Termino` de la siguiente manera,

```
data Termino = V Nvariable
             | T String [Termino]
             deriving (Eq, Show)
```

Por ejemplo,

- x_2 se representa por `(V ("x", 2))`

- a se representa por `(T "a" [])`
- $f(x_2, g(x_2))$ se representa por `(T "f" [V "x" 2, T "g" [V "x" 2]])`

A continuación definimos algunos conceptos importantes sobre los términos.

Definición 2.3. ***El conjunto de variables de un término*** *t, denotado por $V(t)$, se define de manera recursiva como,*

$$V(t) = \begin{cases} \{t\} & si\ t \in X \\ \bigcup_{i=1}^{n} V(t_i) & si\ t = f(t_1, \ldots, t_n) \end{cases}$$

Por ejemplo el término $f(i(i(e)), f(i(e), x)) \in \Sigma_G$, tiene como conjunto de variables $\{x\}$.

Para su implementación en Haskell definiremos la función `conjuntoVar t`que devuelve la lista de variables del término `t`.

```
>>> conjuntoVar (V ("x",2))
[("x",2)]
>>> conjuntoVar (T "f" [T "e" []])
[]
>>> conjuntoVar (T "f" [T "g" [V ("x",1)], T "h" [], V ("y",1)])
[("x",1),("y",1)]
```

Su definición es,

```
conjuntoVar :: Termino -> [Nvariable]
conjuntoVar (V x)    = [x]
conjuntoVar (T _ ts) = concatMap conjuntoVar ts
```

Definición 2.4. *La **longitud** de un término t, denotado por $l(t)$, se define de manera recursiva como,*

$$l(t) = \begin{cases} 1 & si\ t \in X \\ 1 + \sum_{i=1}^{n} l(t_i) & si\ t = f(t_1, \ldots, t_n) \end{cases}$$

Por ejemplo el término $f(i(i(e)), f(i(e), x)) \in \Sigma_G$, tiene longitud 8. En Haskell, creamos la función `longitudTerm t` que calcula la longitud del término `t`

```
>>> longitudTerm (V ("x",2))
1
>>> longitudTerm (T "f" [T "e" []])
2
>>> longitudTerm (T "f" [T "g" [V ("x",1)], T "h" [], V ("y",1)])
5
```

Su definición es,

```
longitudTerm :: Termino -> Int
longitudTerm (V _)    = 1
longitudTerm (T _ xs) = 1 + sum (map longitudTerm xs)
```

Definición 2.5. *El **tamaño** de un término t, denotado por $|t|$, se define de manera recursiva como,*

$$|t| = \begin{cases} 0 & si\ t \in X \\ 0 & si\ t = a \\ 1 + \sum_{i=1}^{n} |t_i| & si\ t = f(t_1, \ldots, t_n) \end{cases}$$

Por ejemplo el término $f(i(i(e)), f(i(e), x)) \in \Sigma_G$, tiene tamaño 5. La función que calcula el tamaño de un término `t`, la llamaremos `tamanoTerm t`.

```
>>> tamanoTerm (V ("x",2))
0
>>> tamanoTerm (T "f" [T "e" []])
1
>>> tamanoTerm (T "f" [T "g" [V ("x",1)], T "h" [], V ("y",1)])
2
```

Su definición es,

```
tamanoTerm :: Termino -> Int
tamanoTerm (V _)    = 0
tamanoTerm (T _ []) = 0
tamanoTerm (T _ xs) = 1 + sum (map tamanoTerm xs)
```

Una representación interesante de los términos compuestos es mediante árboles como se observa en la figura 2.1.

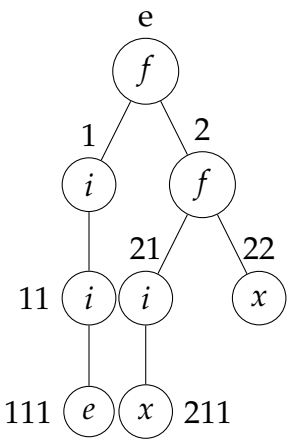


Figura 2.1: Representación de $f(i(i(e)), f(i(x), x))$ en forma de árbol.

Las definiciones anteriores se pueden ver más claras con esta representación. El conjunto de variables de un término t son las hojas del árbol, la longitud es el número de nodos del árbol, y el tamaño es el numero de nodos del árbol menos sus hojas. Las variables que se encuentren en las hojas de este árbol, las llamaremos variables suelo.

En la figura 2.1 cada nodo lleva una numeración. Esto es lo que conocemos en términos como la posición. Nos resultará útil a la hora de definir subtérminos y de dotar de cierta estructura al concepto de término.

Definición 2.6. *Sea Σ una signatura, X un conjunto de variables disjuntas de Σ, y $s \in T(\Sigma, X)$. El* ***conjunto de posiciones*** *$Pos\ (s)$ se define de manera recursiva como,*

- *Si $s \in X$, entonces $Pos\ (s) := \epsilon$, donde ϵ es la cadena vacía.*
- *Si $s = f(s_1, \ldots, s_n)$ entonces,*

$$Pos\ (s) := \{\epsilon\} \cup \{1p | p \in Pos(s_1)\} \cup \cdots \cup \{np | p \in Pos\ (s_n)\}$$

El conjunto de Pos (s) serán cadenas de números naturales positivos. Por ejemplo, Pos $f(f(i(i(e)),f(i(x),x))) = \{\epsilon, 1, 11, 111, 2, 21, 211, 22\}$.

La implementaremos en Haskell bajo el nombre de la función `conjuntoPos t`.

```
>>> conjuntoPos (V ("x",1))
[[]]
>>> conjuntoPos (T "f" [V("x",1)])
[[],[1]]
>>> conjuntoPos (T "f" [T "i" [T "i" [T "e" []]],
                        T "f" [T "i" [V("x",1)], V("x",1)]])
[[],[1],[1,1],[1,1,1],[2],[2,1],[2,1,1],[2,2]]
```

Su definición es,

```
conjuntoPos :: Termino -> [[Int]]
conjuntoPos (V _) = [[]]
conjuntoPos (T _ (xs)) = conjuntoPos' (reverse zs)
    where zs = (zip xs [1..])
          conjuntoPos' [] = [[]]
          conjuntoPos' ((t,n):ts) = (conjuntoPos' ts) ++
                                    (map (n:) (conjuntoPos t))
```

A continuación, introducimos la noción de subtérmino.

Definición 2.7. *Sea $s \in T(\Sigma, X)$ y $p \in Pos\ (s)$. El **subtérmino** de s en la posición p, denotado por $s|_p$ se define por inducción*

$$\begin{aligned} s|_\epsilon &:= s \\ f(s_1, \ldots, s_n)|_{iq} &:= s_i|_q \end{aligned}$$

De la misma manera, podemos extrapolar este concepto a los árboles. Un término t es subtérmino del término s, si el árbol generado por el término t es un subárbol para el árbol generado por s.

Implementamos la función `esSubtermino t s` que comprueba si t es subtérmino de s. Por ejemplo,

```
>>> esSubtermino (V ("x",1)) (T "f" [V("x",1)])
True
>>> esSubtermino (V ("x",2)) (T "f" [V("x",1)])
False
```

Queda implementada de la siguiente manera,

```
esSubtermino :: Termino -> Termino -> Bool
esSubtermino t s@(T _ xs)
    | t == s = True
    | otherwise = or (map (esSubtermino t) xs)
esSubtermino t s = t == s
```

Una función que usaremos en los capítulos siguientes es `ocurre v t`, que es la función `esSubtermino` restringido a variables. A partir de ahora diremos que v ocurre en t si la variable v es subtérmino de t. Por ejemplo,

```
>>> ocurre ("x",2) (V ("x",2))
True
>>> ocurre ("x",3) (V ("x",2))
False
>>> ocurre ("x",2) (T "f" [V ("x",2), T "g" [V ("x",2)]])
True
>>> ocurre ("y",5) (T "f" [V ("x",2), T "g" [V ("x",2)]])
False
```

Dado la naturaleza de la restricción variables, podríamos definir `ocurre` como,

```
ocurre :: Nvariable -> Termino -> Bool
ocurre v t = esSubtermino (V v) t
```

Sin embargo, podemos mejorar la definición sin usar la función `esSubtermino`,

```
ocurre :: Nvariable -> Termino -> Bool
ocurre a (V x)    = a == x
ocurre a (T _ ts) = any (ocurre a) ts
```

Con la noción de subtérmino, introducimos una nueva función de términos en términos. La idea es, a partir de $p \in \text{Pos}\ (s)$ y de dos términos s, t sustituir el subtérmino $s|_p$ por t. En la noción de árbol, sería cambiar todo el subárbol $s|_p$ por t, dejando el resto de s intacto.

Formalmente lo definimos como sigue,

Definición 2.8. *Sea $p \in Pos(s)$. Denotamos $s[t]_p$ como,*

$$\begin{array}{rcl} s[t]_\epsilon & := & t \\ f(s_1,\ldots,s_n)[t]|_{iq} & := & f(s_1,\ldots,s_i[t]_q,\ldots,s_n) \end{array}$$

La función que calcule el término resultante la llamaremos `sustPosSubtermino`. Debemos tener en cuenta que la de definición depende de la posición, por ello añadiremos una excepción si p no esta bien definido. Por ejemplo,

```
>>> sustPosSubtermino (T "f" [V ("s",1),V ("s",2),V ("s",3)])
                      (V ("t",1)) [2,1]
T "f" [V ("s",1),V ("t",1),V ("s",3)]
sustPosSubtermino (T "f" [V ("s",1),V ("s",2),V ("s",3)])
                  (V ("t",1)) []
(V ("t",1))
sustPosSubtermino (V ("s",1)) (V ("t",1)) [2]
*** Exception: No se ha definido bien la lista de posiciones
>>> sustPosSubtermino (T "f" [V ("s",1),V("s",2),V("s",3)])
                      (V ("t",1)) [2,2]
T "f" [V ("s",1),*** Exception: No se ha definido bien la
                     lista de posiciones
```

Su definición es,

```
sustPosSubtermino :: Termino -> Termino -> [Int] -> Termino
sustPosSubtermino _ t [] = t
sustPosSubtermino (V _) t [1] = t
sustPosSubtermino (V _) _ _ = error("No se ha definido bien la
                                    lista de posiciones")
sustPosSubtermino (T f xs) t (i:is) = (T f
    ((take (i-1) xs) ++
    (sustPosSubtermino (xs!!(i-1)) t is) :
    (drop i xs)))
```

Sustituciones

En esta sección vamos a introducir el concepto de sustituciones entre términos. De manera general, una sustitución será una aplicación desde un conjunto de las variables hacia el de los términos. Por ejemplo, sea $f(x)$ un término t y $\{x \mapsto i(x)\}$ la sustitución $\tilde{\sigma}$. Si aplicamos $\tilde{\sigma}$ a t obtenemos $\tilde{\sigma}(t) = f(i(x))$. Formalmente la definiremos como sigue,

Definición 2.9. *Sea Σ una signatura y X un conjunto infinito numerable de variables. Una $T(\Sigma, X)$–**sustitución**, es una función $\sigma : X \to T(\Sigma, X)$ tal que $\sigma(x) \neq x$ para un número finito de variables $x \in X$. El conjunto de variables donde $\sigma(x) \neq x$ se le llama **dominio de** σ. Lo denotaremos por Dom (σ).*

En definitiva, las sustituciones son de la forma

$$\sigma = \{x_1 \mapsto \sigma(x_1), \ldots, x_n \mapsto \sigma(x_n)\}$$

entendiendo que las restantes variables se aplican en ellas mismas.

La implementación del tipo de dato `Sustitucion` en Haskell es,

```
type Sustitucion = [(Nvariable,Termino)]
```

Por ejemplo, la sustitución $\sigma = \{x_2 \mapsto a, y_4 \mapsto z_5, v_1 \mapsto f(w_2)\}$ se representa por

```
[(("x",2),T "a" []),
 (("y",4),(V ("z",5))),
 (("v",1),T "f" [V ("w",2)])]
```

Para comprobar si una variable está en el dominio de una sustitución, definimos la función `enDominio`, tal que `(enDominio x s)` se verifica si `x` está en el dominio de `s`. Por ejemplo,

```
>>> let s = [(("x",2),T "a" []),(("y",4),(V ("z",5)))]
>>> enDominio ("x",2) s
True
>>> enDominio ("x",3) s
False
>>> enDominio ("y",2) s
False
>>> enDominio ("y",4) s
True
>>> enDominio ("z",5) s
False
```

Su definición es:

```
enDominio :: Nvariable -> Sustitucion -> Bool
enDominio v = any (\(x,_) -> v == x)
```

Con la función `(aplicaVar s v)`, calculamos el término obtenido tras aplicar una sustitución `s` a una variable `v`. Por ejemplo,

```
let s = [(("x",2),T "a" []),(("y",4),(V ("z",5)))]
>>> aplicaVar s ("x",2)
T "a" []
>>> aplicaVar s ("x",3)
V ("x",3)
>>> aplicaVar s ("y",2)
V ("y",2)
```

```
>>> aplicaVar s ("y",4)
V ("z",5)
>>> aplicaVar s ("z",5)
V ("z",5)
```

Su definición es:

```
aplicaVar :: Sustitucion -> Nvariable -> Termino
aplicaVar [] z = V z
aplicaVar ((x,y):s) z
  | x == z    = y
  | otherwise = aplicaVar s z
```

La definición de sustitución se puede extender para términos $t \in T(\Sigma, X)$. Sea la extensión $\tilde{\sigma}$ definida como,

$$\tilde{\sigma}(t) = \begin{cases} \tilde{\sigma}(x) = \sigma(x) & \text{si } t \in X \\ \tilde{\sigma}(f(t_1, \ldots, t_n)) = f(\tilde{\sigma}(t_1), \ldots, \tilde{\sigma}(t_n)) & \text{si } t = f(t_1, \ldots, t_n) \end{cases}$$

A partir de ahora, entenderemos σ como una sustitución extendida, a menos que se indique lo contrario.

Para la computación de esta definición usamos la función `(aplicaTerm s t)`, que es el término obtenido tras aplicar la sustitución `s` al término `t`. Por ejemplo,

```
>>> let s = [(("x",2),T "a" []),(("y",4),(V ("z",5)))]
>>> aplicaTerm s (T "s" [V ("x",2), T "m" [V ("y",4)]])
T "s" [T "a" [],T "m" [V ("z",5)]]
>>> aplicaTerm s (T "s" [V ("x",4), T "m" [V ("y",2)]])
T "s" [V ("x",4),T "m" [V ("y",2)]]
```

Su definición es:

```
aplicaTerm :: Sustitucion -> Termino -> Termino
aplicaTerm s (V x)    = aplicaVar s x
aplicaTerm s (T f ts) = T f (map (aplicaTerm s) ts)
```

Como la sustituciones son muy parecidas a las funciones, podemos plantearnos la composición entre ellas. Definiéndolas como $\sigma\tau(x) := \tilde{\sigma}(\tau(x))$. Veamos que $\sigma\tau(x)$ es una sustitución

Lema 2.1. La composición de sustituciones es una sustitución.

Demostración. $\sigma\tau(x)$ es una función de X a $T(\Sigma, X)$. Como $\sigma\tau(x) = x$ ocurre para todas las variables $x \in V - (\text{Dom}\,(\sigma) \cup \text{Dom}\,(\tau))$, por definición, es una sustitución. ∎

A continuación, introducimos un nuevo concepto imprescindible para la reescritura.

Definición 2.10. *Sea Σ una signatura y X un conjunto finito numerable de variables disjuntas de σ. Una **identidad** es un par $(s,t) \in T(\Sigma, X) \times T(\Sigma, X)$. Las denotaremos por $s \approx t$, donde s lo denominaremos el lado izquierdo y t el lado derecho.*

La idea de esta definición es similar a lo que entendemos por equivalencia. Por ejemplo, en la definición de grupos tenemos la propiedad asociativa, $(a + b) + c = a + (b + c)$. De la misma manera podemos representar la propiedad como una identidad de Σ_G, $f(f(a,b),c) \approx f(a,f(b,c))$.

Podemos relacionar este concepto con lo que hemos visto en el capítulo 1.

Definición 2.11. *Sea E un conjunto de identidades de Σ. La **relación de reducción** $\to_E \subseteq T(\Sigma, V) \times T(\Sigma, V)$ se define como,*

$s \to_E t$ *syss* $\exists (l,r) \in E, p \in Pos\,(s), \sigma$ *una sustitución tal que* $s|_p = \sigma(l)$ *y* $t = s[\sigma(r)]_p$.

Volviendo al ejemplo de grupos, su conjunto de identidades es,

$$G := \{f(x,f(y,z)) \approx f(f(x,y),z), f(e,x) \approx x, f(i(x),x) \approx e\}$$

El término $f(i(a),f(a,e))$, con a una constante, tiene una reducción en la posición ϵ con la sustitución $\sigma = \{x \mapsto i(a), y \mapsto a, z \mapsto e\}$, y es $f(f(i(a),a),e)$. Podemos afirmar nuevamente que $f(f(i(a),a),e) \to_G f(e,e)$, en la posición 1 y la sustitución $\sigma = \{x \mapsto a\}$.

Caracterización de $\overset{*}{\leftrightarrow}_E$

Como hemos visto en el capítulo 1, la clausura transitiva reflexiva de $\rightarrow_E$ la denotamos por $\overset{*}{\rightarrow}_E$, y la clausura simétrica transitiva reflexiva por $\overset{*}{\leftrightarrow}_E$. En la última parte de este capítulo, vamos a dar una caracterización de $\overset{*}{\leftrightarrow}_E$ que usaremos en capítulos posteriores.

Definición 2.12. *Sea $\equiv$ una relación binaria en $T(\Sigma, V)$*

1. *La relación $\equiv$ es **cerrada bajo sustituciones** syss $s \equiv t$ implica $\sigma(s) \equiv \sigma(t)$, para todo s, t y σ.*
2. *La relación $\equiv$ es **cerrada bajo Σ–operaciones** syss $s_1 \equiv t_1, \ldots, s_n \equiv t_n$ implica $f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)$, para todo $n \geq 0, f, s_1, \ldots s_n, t_1, \ldots t_n$.*
3. *La relación $\equiv$ es **compatible con Σ–operaciones** syss $s \equiv t$ implica $f(s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_n) \equiv f(s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n)$ para todo $n \geq 0, f, i = 1, \ldots, n$, y $s_1, \ldots, s_{i-1}, s, t, s_{i+1}, \ldots, s_n$.*
4. *La relación $\equiv$ es **compatible con el Σ–contexto** syss $s \equiv s'$ implica $t[s]_p \equiv t[s']_p$ para todo t y p.*

Una consecuencia directa a partir de la definición de $\rightarrow_E$ es,

Lema 2.2. Sea E un conjunto de Σ–identidades. La relación de reducción de $\rightarrow_E$ es cerrada bajo sustitución y compatible con las Σ–operaciones.

Demostración. Es trivial pues si sabemos que $s_i \equiv t_i \forall i$, entonces $\sigma(s_i) \equiv \sigma(t_i) \forall i$, y $f(s_1, \ldots, s_n) \equiv f(t_1, \ldots, t_n)$ por la propia definición de relación de reducción. ∎

Otro lema importante para la caracterización es,

Lema 2.3. Sea $\equiv$ una relación binaria en $T(\Sigma, V)$.

1. La relación $\equiv$ es compatible con las Σ–operaciones syss es compatible con el Σ–contexto.
2. Si $\equiv$ es reflexiva y transitiva, entonces es compatible con las Σ–operaciones syss es cerrado bajo las Σ–operaciones.

Demostración. La primera propiedad de derecha a izquierda es trivial por la definición de posición. De izquierda a derecha se realiza mediante una inducción de p, como para $p = \epsilon$ la afirmación es cierta, también lo será para $p = p_1$,

por tanto también para $p = p_1 p_2 \ldots$. La segunda propiedad de izquierda a derecha es cierta por la transitividad de $\equiv$. De derecha a izquierda se usa que la relación es reflexiva.

Teorema 2.1. *Sea E un conjunto de Σ–identidades. La relación $\overset{*}{\leftrightarrow}_E$ es la relación de equivalencia más pequeña en $T(\Sigma, V)$ que contiene E que es cerrada bajo sustituciones y Σ-operaciones.*

Demostración. $\overset{*}{\leftrightarrow}_E$ es una relación de equivalencia reflexiva y transitiva. Con el lema 2.2 tras aplicar inducción, se puede ver que es cerrada bajo sustituciones y compatible con Σ–operaciones. Por el lema 2.3, también es cerrada bajo Σ–operaciones.

Vamos a suponer que existe $\equiv$ relación de equivalencia en $T(\Sigma, v)$ que contiene a E, y es cerrada bajo sustituciones y Σ–operaciones. Nuestro objetivo es comprobar que $\overset{*}{\leftrightarrow} \subseteq \equiv$, para ello, demostraremos que si $s \to_E t$ entonces $s \equiv t$.

Partiendo de $s \to_E$, por la definición de relación de reducción, existe $(l, r) \in E$, p posición de s, y σ una sustitución tal que $s|_p = \sigma(l)$ y $t = s[\sigma(r)]_p$.

Como $\equiv$ contiene a E entonces $l \equiv r$, por tanto, $\sigma(l) \equiv \sigma(r)$ por ser cerrada bajo sustituciones. Al ser $\equiv$ reflexiva y cerrada bajo Σ–operaciones y compatible con Σ–operaciones, por el lema 2.3 también es compatible con Σ–contexto y por la definición de la propiedad, $s = s[\sigma(l)]|_p \equiv s[\sigma(r)]|_p = t$.

La demostración del teorema 2.1, nos indica una manera de calcular $\overset{*}{\leftrightarrow}_E$ a partir de E mediante las relaciones de reflexividad, simetría, transitividad, sustituciones, y Σ–operaciones. La aplicación de estas reglas en este contexto, es lo que llamamos lógica ecuacional. Las reglas se pueden escribir así,

Reflexiva $$\frac{}{E \vdash t \approx t}$$

Simétrica $$\frac{E \vdash s \approx t}{E \vdash t \approx s}$$

Transitiva $$\frac{E \vdash s \approx t \qquad E \vdash t \approx u}{E \vdash s \approx u}$$

Sustitución $$\frac{E \vdash s \approx t}{E \vdash \sigma(s) \approx \sigma(t)}$$

Σ–operaciones $$\frac{E \vdash s_1 \approx t_1 \qquad \ldots \qquad E \vdash s_n \approx t_n}{E \vdash f(s_1, \ldots, s_n) \approx f(t_1, \ldots, t_n)}$$

Capítulo 3

El problema de la unificación y normalización de términos

En este capítulo vamos definir los problemas de decisión en el razonamiento ecuacional. Nos centraremos en los sistemas de reescritura de términos y la unificación. Para ello consideraremos la siguiente definición.

Definición 3.1. *Dado un conjunto de identidades E y una ecuación $s \approx t$, diremos que la ecuación es,*

Válida	*en E syss $s\ s \approx_E t$*
Satisfactible	*en E syss hay una sustitución σ tal que $\sigma(s) \approx_E \sigma(t)$*

En la siguiente sección obtendremos varias propiedades interesantes sobre $\approx_E$.

Estudio de $\approx_E$

En el teorema 1.1, vimos que $\overset{*}{\leftrightarrow}$ era decidible si $\rightarrow_E$ si es convergente. De esta manera podíamos computar $\downarrow_E$. Por ello es importante saber si podemos decidir si un término implica a otro término. Esto se llama el problema del emparejamiento, dado dos términos s y l, ver si existe una sustitución $\sigma(l) = s$, y computar σ si existe.

Comprobaremos que el problema del emparejamiento y de la satisfactibili-

dad están muy relacionados.

Teorema 3.1. *Si E es finito y $\to_E$ es convergente, entonces $\approx_E$ es decidible.*

Demostración. Por el teorema 1.1, sabemos que $s \overset{*}{\leftrightarrow}_E t$ syss $s \downarrow_E = t \downarrow_E$. La forma normal del operador $\downarrow_E$ es computable, pues podemos decidir si un termino u es forma normal, y si no lo fuese, podemos computar un u' tal que $u \to_E u'$, y u' fuese una forma normal.

Para ver si u es una forma normal, tenemos que comprobar para todas las identidades de E y todas la posiciones $p \in Pos(u)$ si hay una sustitución σ tal que $u|_p = \sigma l$. Como el problema del emparejamiento es decidible y E es finito, u es una forma normal o podemos reducir u en un proceso iterativo finito a una formal normal. Este proceso termina pues $\to_E$ es terminante. ∎

Por este resultado, surge un interés en las reducciones convergentes. El problema de la palabra para E trata de decidir $s \approx t$ para arbitrarios $s, t, \in T(\Sigma, V)$. También tenemos el problema de las palabras básicas para E, que no es más que el problema de la palabra restringido a términos básicos s, t.

Ejemplo 3.1. Considerando la signatura $\Sigma = \{*, S, K\}$ donde $*$ es una función binaria y S y K son constantes. El conjunto de identidades considerado es,

$$E := \{((S \cdot x) \cdot y) \cdot z \approx (x \cdot z) \cdot (y \cdot z), (K \cdot x) \cdot y \approx x\}$$

Como se puede dar cualquier término a partir de uno básico, esto implica que el problema de la palabra para E sea indecidible.

Sistemas de reescritura de términos

En esta sección vamos a introducir la definición de regla y sistema de reescritura, así como una propiedad.

Definición 3.2. *Una **regla de reescritura** es una identidad $l \approx r$ tal que l no es una variable y $Var(l) \supseteq Var(r)$. En ese caso, escribiremos $l \to r$. Un sistema de reescritura de términos (SRT) es un conjunto de reglas de escritura. Una expresión reducible es una instancia del lado derecho de una regla de reescritura. Contraer la expresión reducible significa sustituirla por la instancia del lado derecho de la regla.*

Como cualquier SRT es un conjunto de identidades, $\rightarrow_R$ está bien definido.

A partir del teorema 3.1, deducimos el siguiente resultado,

Teorema 3.2. *Si R es un SRT convergente finito, $\approx_R$ es decidible: $s \approx_R t$ syss $s\downarrow_R = t\downarrow_R$.*

Veamos ahora una de las propiedades básicas de $\rightarrow_R$.

Definición 3.3. *Una relación en $T(\Sigma, V)$ es una **relación de reescritura** syss es compatible con las Σ-operaciones y cerrada bajo sustitución.*

Por el 2.3, sabemos que $\rightarrow_R$ es una relación de reescritura. Por inducción de las reglas de derivación, $\xrightarrow{*}_R$ y $\xrightarrow{+}_R$ también son relaciones de reescritura. Para $\xleftrightarrow{*}_R$ lo demuestra el teorema 2.1

Unificación sintáctica

En esta sección nos centraremos en estudiar la unificación. La unificación es el proceso de resolver la satisfactibilidad de un problema. Dado E, s y t, hay que encontrar una sustitución σ tal que $\sigma s \approx_E \sigma t$. Si s y t son básicos, el problema pasaría a ser un problema de palabras básicas. En esta sección estudiaremos el caso especial $E = \emptyset$ que es importante para algunos algoritmos simbólicos de computación. El que especialmente nos importa es un algoritmo que comprueba la confluencia a partir de pares críticos, que veremos mas adelante.

La sustitución o el unificador que busquemos va a depender del caso en el que nos encontremos. Por ejemplo $x \approx^? f(y)$ tiene varios unificadores $\{x \mapsto f(y)\}$, $\{x \mapsto f(a), y \mapsto a\}$. No siempre puede existir uno o varios unificadores por ejemplo, $f(x) \approx^? g(x)$ no tiene unificador. Para el caso en el que existan varios unificadores, nos gustaría encontrar el más general,

Definición 3.4. *Una sustitución σ es **mas general** que una sustitución σ' si hay una sustitución δ tal que $\sigma' = \delta\sigma$. En este caso escribimos $\sigma \lesssim \sigma'$. Diremos que σ' es una instancia de σ.*

En el ejemplo anterior, si $\sigma = \{x \mapsto f(y)\}$ y $\sigma' = x \mapsto f(a), y \mapsto a\}$ entonces $\sigma \lesssim \sigma'$ porque $\sigma' = \delta\sigma$ donde $\delta = \{y \mapsto a\}$.

Una propiedad inmediata de la definición es,

Lema 3.1. $\sigma \sim \sigma'$ syss existe una sustitución inyectiva p que sea biyectiva para las variables tal que $\sigma = p\sigma'$

Como nuestro objetivo es resolver conjuntos de ecuaciones, formalizaremos el problema de la unificación con la siguiente definición,

Definición 3.5. *Un **unificador** de un conjunto finito de ecuaciones* $S = \{s_1 =^? t_1, \ldots, s_n =^? t_n\}$ *es una sustitución* σ *tal que* $\sigma s_i = \sigma t_i$ *para* $i = 1, \ldots, n$. *Por* $U(S)$ *se denota el conjunto de todos los unificadores de* S. *Se dice que* S *es **unificable** si* $U(S) \neq \emptyset$. *El **problema de la unificación** de* S *consiste en calcular sus unificadores.*

Otra definición importante es,

Definición 3.6. *Una sustitución* σ *es el **unificador de máxima generalidad** (UMG) de* S *si* σ *es el menor elemento de* $U(S)$*; es decir,*

1. $\sigma \in U(s)$
2. $\forall \sigma' \in U(s) \sigma \lesssim \sigma'$

Por ejemplo, $\sigma' := \{x \mapsto z, y \mapsto z\}$ es un unificador de $x = f =^? y$, pero no es UMG, pues es menos general que $\sigma = x \mapsto y$.

Definición 3.7. *Una sustitución* σ *es **idempotente** si* $\sigma = \sigma\sigma$.

Lema 3.2. Una sustitución σ es idempotente syss $\text{Dom}(\sigma) \cap \text{VRan}(\sigma) = \emptyset$

El ultimo resultado de este capítulo se usará en la implementación del problema de la unificación.

Teorema 3.3. *Si un problema de unificación* S *tiene solución, entonces tiene un UMG idempotente.*

Demostración. El lema 3.1 nos dice que los UMG son únicos respecto a una sustitución p inyectiva y biyectiva para las variables. Por ello, incluso para los idempotentes UMG no serán únicos.

Unificación por transformación

La unificación puede ser representada como la aplicación iterada de transformaciones en un conjunto de ecuaciones. Por ejemplo el algoritmo que se-

guimos para resolver sistemas lineales,

Definición 3.8. *El problema de la unificación de* $S = \{x_1 =^? t_1, \ldots, x_n =^? t_n\}$ *está en su* ***forma resuelta****, si las variables* x_i *son distintas entre sí y ninguna* x_i *ocurre en* t_i*. En este caso definimos,* $\vec{S} := \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$.

Varias propiedades básicas de $\vec{S}$ son,

Lema 3.3. Si S está en un su forma resuelta entonces, $\sigma = \sigma\vec{S}$ para todo $\sigma \in U(S)$.

Demostración. Sea $S = \{x_1 =^? t_1, \ldots, x_n =^? t_n\}$. Probaremos por casos que $\forall x \in V.\sigma x = \sigma\vec{S}x$

Para el caso $x \in \{x_1, \ldots, x_n\}$, supongamos sin perdida de generalidad que $x = x_m$. Entonces $\sigma x = \sigma t_k = \sigma\vec{S}x$ pues $\sigma \in U(S)$.

Para el caso $x \notin \{x_1, \ldots, x_n\}$. Entonces $\sigma x = \sigma\vec{S}x$, pues $\vec{S}x = x$.

Lema 3.4. Si S está en su forma resuelta entonces $\vec{S}$ es un UMG idempotente de S.

Demostración. Usando el lema 3.2, deducimos que ninguno de los x_i ocurren en t_i. Por la misma razón tenemos que $\vec{S}x_i = t_i = \vec{S}t_i$, es decir $\vec{S} \in U(S)$. Finalmente $\vec{S}$ es un UMG por el lema 3.3, $\vec{S} \lesssim \sigma$ para todo $\sigma \in U(S)$.

Es decir, una vez que tengamos la forma resuelta de S, podemos extraer un UMG idempotente.

Vamos a dar una serie de reglas de transformación para conseguirlo.

Borrar	Elimina ecuaciones triviales
Descomponer	Reemplaza ecuaciones entre términos por ecuaciones entre sus subtérminos
Orientar	Mueve variables hacia el lado izquierdo
Resuelve	Afianza las soluciones, eliminando la variable resuelta en el resto del problema

Que formalmente significan,

Borrar	$\{t =^? t\} \sqcup S$	$\Longrightarrow$	S
Descomponer	$\{f(\overline{t_n}) =^? f(\overline{u_n})\} \sqcup S$	$\Longrightarrow$	$\{t_1 =^? u_1, \ldots, t_n =^? u_n\} \cup S$
Orientar	$\{t =^? x\} \sqcup S$	$\Longrightarrow$	$\{x =^? t\} \cup S$ si $t \notin V$
Resuelve	$\{x =^? t\} \sqcup S$	$\Longrightarrow$	$\{x =^? t\} \cup \{x \mapsto t\}(S)$ si $x \in \text{Var}(S) - \text{Var}(t)$

Donde $\sqcup$ denota la unión disjunta.

Definición 3.9. *La función* $\texttt{Unificar}(S)$ *devuelve un unificador de máxima generalidad si existe, y falla si no. Tiene la siguiente definición,*

$\texttt{Unificar(S)} \quad = \quad$ *Mientras exista un* T *tal que* $S \Longrightarrow T, S := T$.
Si S *es una forma resuelta entonces devuelve* $\vec{S}$*, si no, falla.*

Este algoritmo es no determinista, pues siempre se puede aplicar más de una regla de transformación. Por ello, el algoritmo escogerá una regla arbitraria. La terminación de `Unificar` dependerá de la terminación de $\Longrightarrow$.

Lema 3.5. `Unificar` termina para todos las entradas.

Lema 3.6. Si $S \Longrightarrow T$ entonces $U(S) = U(T)$

Lema 3.7. Si `Unificar`(S) devuelve una sustitución σ, σ es un UMG idempotente de S

Lema 3.8. Una ecuación $f(\sigma(\overline{s_m})) =^? g(\overline{t_n})$ donde $f \neq g$, no tiene solución.

Lema 3.9. Una ecuación $x =^t$, donde $x \in \text{Var}\,(t)$ y $x \neq t$, no tiene solución.

Lema 3.10. Si S es resoluble, $\texttt{Unificar}(S)$ no falla.

Implementación de la unificación y la reescritura de términos en Haskell

En esta sección vamos a implementar en el lenguaje de programación funcional Haskell los algoritmos de este capítulo. Usaremos algunas funciones del capítulo anterior, que se encuentran en la librería `Terminos.hs`.

Empezamos implementando los errores. Estos son causados por `UNIFICACION` (si el sistema no tiene solución) o `REGLA`(si durante la reescritura no se puede aplicar ninguna regla más).

```
data ERROR = UNIFICACION
           | REGLA
           deriving (Eq, Show)
```

Una ecuación es un par de términos

```
type Ecuacion = (Termino,Termino)
```

Un sistema de ecuaciones es una lista de ecuaciones

```
type Sistema = [Ecuacion]
```

Algoritmo de unificación

`(unificacion t1 t2)` es

- `(Right s)` si los términos `t1` y `t2` son unificables y `s` es un unificador del máxima generalidad de `t1` y `t2`.
- `(Left UNIFICACION)` si `t1` y `t2` no son unificables.

Por ejemplo,

```
>>> unificacion (T "f" [V ("x",0), T "g" [V ("z",0)]])
                  (T "f" [T "g" [V ("y",0)], V ("x",0)])
Right [(("z",0),V ("y",0)),(("x",0),T "g" [V ("y",0)])]
>>> unificacion (T "f" [V ("x",0),T "b" []])
                  (T "f" [T "a" [],V ("y",0)])
Right [(("y",0),T "b" []),(("x",0),T "a" [])]
>>> unificacion (T "f" [V ("x",0),V ("x",0)])
                  (T "f" [T "a" [],T "b" []])
Left UNIFICACION
>>> unificacion (T "f" [V ("x",0),T "g" [V ("y",0)]])
                  (T "f" [V ("y",0),V ("x",0)])
Left UNIFICACION
```

Su código es,

```
unificacion :: Termino -> Termino -> Either ERROR Sustitucion
unificacion t1 t2 = unificacionS [(t1,t2)] []
```

`(unificacionS es s)` es

- `(Right s')` si el sistema `s(es)`, obtenido aplicando la sustitución `s` a la ecuaciones de `es`, es unificable y `s'` es un unificador del máxima generalidad de `s(es)`.
- `(Left UNIFICACION)` si `s(es)` no es unificable.

Por ejemplo,

```
>>> unificacionS [(T "f" [V ("x",0), T "g" [V ("z",0)]],
                   T "f" [T "g" [V ("y",0)], V ("x",0)])]
                   []
Right [(("z",0),V ("y",0)),(("x",0),T "g" [V ("y",0)])]
>>> unificacionS [(T "f" [V ("x",0),T "b" []],
                   T "f" [T "a" [],V ("y",0)])]
                   []
Right [(("y",0),T "b" []),(("x",0),T "a" [])]
>>> unificacionS [(T "f" [V ("x",0),V ("x",0)],
                   T "f" [T "a" [],T "b" []])]
                   []
Left UNIFICACION
>>> unificacionS [(T "f" [V ("x",0),T "g" [V ("y",0)]],
                   T "f" [V ("y",0),V ("x",0)])]
                   []
Left UNIFICACION
```

Su código es,

```
unificacionS :: Sistema -> Sustitucion -> Either ERROR Sustitucion
unificacionS [] s = Right s
unificacionS ((V x,t):ts) s
  | V x == t  = unificacionS ts s
  | otherwise = reglaElimina x t ts s
unificacionS ((t,V x):ts) s =
  reglaElimina x t ts s
unificacionS ((T f ts1, T g ts2):ts) s
  | f == g    = unificacionS (zip ts1 ts2 ++ ts) s
  | otherwise = Left UNIFICACION
```

`(reglaElimina x t es s)` es

- `(Left UNIFICACION)`, si `x` ocurre en `t`.
- `(Right s')` es la solución del sistema obtenido al aplicarle a `es` la sustitución [x/t] en el entorno obtenido componiendo la sustitución `s` con [x/t], en caso contrario.

Por ejemplo,

```
>>> reglaElimina ("x",0) (T "f" [V ("x",0)]) [] []
Left UNIFICACION
>>> reglaElimina ("x",0) (T "f" [V ("x",1)]) [] []
Right [(("x",0),T "f" [V ("x",1)])]
>>> reglaElimina ("x",0) (T "f" [V ("x",1)]) [] [(("y",2),V ("x",0))]
Right [(("x",0),T "f" [V ("x",1)]),(("y",2),T "f" [V ("x",1)])]
>>> reglaElimina ("x",0) (T "f" [V ("x",1)]) [(V ("x",1),V ("x",0))] []
Left UNIFICACION
>>> reglaElimina ("x",0) (T "f" [V ("x",1)]) [(V ("x",1),V ("y",0))] []
Right [(("x",1),V ("y",0)),(("x",0),T "f" [V ("y",0)])]
>>> reglaElimina ("x",0) (T "f" [V ("x",1)]) [(V ("y",1),V ("x",0))] []
Right [(("y",1),T "f" [V ("x",1)]),(("x",0),T "f" [V ("x",1)])]
```

Su código es,

```
reglaElimina :: Nvariable -> Termino -> Sistema -> Sustitucion
             -> Either ERROR Sustitucion
reglaElimina x t es s
  | ocurre x t = Left UNIFICACION
  | otherwise  = unificacionS es' s'
  where es' = [(aplicaTerm [(x,t)] t1, aplicaTerm [(x,t)] t2)
                | (t1,t2) <- es]
        s'  = (x,t) : map (\(y,u) -> (y, aplicaTerm [(x,t)] u)) s
```

Equiparación de términos

`(equiparacion t1 t2)` es

- `(Right s)` si `t1` y es una instancia de `t1` y `s` es un equiparador del máxima generalidad de `t1` con `t2`.
- `(Left UNIFICACION)` en caso contrario.

Por ejemplo,

```
>>> let t1 = T "f" [V ("x",0), T "a" []]
>>> let t2 = T "f" [T "b" [],  T "a" []]
>>> equiparacion t1 t2
Right [(("x",0),T "b" [])]
>>> equiparacion t2 t1
Left UNIFICACION
>>> let t3 = T "f" [V ("x",0), V ("y",0)]
>>> equiparacion t1 t3
Left UNIFICACION
```

Su código es,

```
equiparacion :: Termino -> Termino -> Either ERROR Sustitucion
equiparacion t1 t2 = equiparacionS [(t1,t2)] []
```

`(equiparacionS es s)` es

- `(Right s')` si la derecha de las ecuaciones de `s(es)` y es una instancia de la izquierda y `s'` es un equiparador de máxima generalidad de la izquierda con la derecha
- `(Left UNIFICACION)` en caso contrario.

Por ejemplo,

```
>>> let t1 = T "f" [V ("x",0), T "a" []]
>>> let t2 = T "f" [T "b" [],  T "a" []]
>>> equiparacionS [(t1,t2)] []
Right [(("x",0),T "b" [])]
>>> equiparacionS [(t2,t1)] []
```

```
Left UNIFICACION
>>> let t3 = T "f" [V ("x",0), V ("y",0)]
>>> equiparacionS [(t1,t3)] []
Left UNIFICACION
```

Su código es,

```
equiparacionS :: Sistema -> Sustitucion -> Either ERROR Sustitucion
equiparacionS [] s = Right s
equiparacionS ((V x,t):es) s
    | not (enDominio x s)   = equiparacionS es ((x,t):s)
    | aplicaVar s x == t    = equiparacionS es s
    | otherwise             = Left UNIFICACION
equiparacionS ((_,V _):_) _ = Left UNIFICACION
equiparacionS ((T f ts1, T g ts2):es) s
    | f == g    = equiparacionS (zip ts1 ts2 ++ es) s
    | otherwise = Left UNIFICACION
```

Reescritura de términos

`(reescribe es t)` es

- `(Right s')`, donde `s'` es el término obtenido reescribiendo `t` con la primera regla de es que con la que se pueda reescribir.
- `(Left REGLA)` en caso contrario.

Por ejemplo,

```
>>> reescribe [(V ("x",1), V ("y",1)),
                (V ("x",2), V ("y",2))]
                (V("x",2))
Right (V ("y",2))
>>> reescribe [(V ("x",1), V ("y",1)),
                (V ("x",1), V ("y",2))]
                (V("x",2))
```

```
Left REGLA
>>> reescribe [(V ("x",2),T "f" [V ("x",3)]),
                 (V ("x",3),T "f" [V ("x",4)])]
                 (V("x",2))
Right (T "f" [V ("x",3)])
>>> reescribe [(V ("x",1),T "a" [])]
                 (T "f" [V ("x",1), T "b" []])
Right (T "f" [T "a" [],T "b" []])
```

Su código es,

```
reescribe :: Sistema -> Termino -> Either ERROR Termino
reescribe [] _ = Left REGLA
reescribe ((l,r):es) t
    | a == Left UNIFICACION = reescribe es t
    | b == t = reescribe es t
    | otherwise = Right b
    where a = equiparacion l r
          b = aplicaTerm (elimR a) t
          elimR (Right x) = x
```

`(formaNormal es t)` es la forma normal de `t` respecto de `es`. Por ejemplo,

```
>>> formaNormal [(V ("x",1), V ("y",1)),
                   (V ("x",2), V ("y",2))]
                   (V("x",2))
V ("y",2)
>>> formaNormal [(V ("x",1), V ("y",1)),
                   (V ("x",2), V ("y",2))]
                   (V("x",3))
V ("x",3)
>>> formaNormal [(V ("z",1), V ("a",1)),
                   (V ("x",1), T "g" [V ("z",1)])]
                   (T "f" [V ("x",1)])
T "f" [T "g" [V ("a",1)]]
```

Su código es,

```
formaNormal :: Sistema -> Termino -> Termino
formaNormal es (V x)
  | isRight a = elimR a
  | otherwise = V x
  where a = reescribe es (V x)
        elimR (Right r) = r
formaNormal es (T f ts)
  | a == Left REGLA = u
  | otherwise       = formaNormal es (elimR a)
  where u = T f (map (formaNormal es) ts)
        a = reescribe es u
        elimR (Right r) = r
```

Capítulo 4

Terminación

Como hemos podido comprobar en los anteriores capítulos, es importante que nuestros sistemas de reescritura tengan la propiedad de la terminación. Sin embargo, en la primera sección de este capítulo demostraremos que el problema de decidir si un sistema de reescritura es terminante, es indecidible. Aunque en casos mas restringidos si podremos decidir sobre la terminación del sistema de reescritura. En la segunda sección definiremos los órdenes de reducción, obteniendo una propiedad con ellos para verificar la terminación. En el resto daremos diferentes maneras de definir los órdenes de reducción.

El problema de decisión

La indecibilidad para el caso general

En esta sección vamos a introducir conceptos generales de las Ciencias de la Computación, aplicados a los sistemas de reescritura. Uno de los problemas generales de las Ciencias de la Computación es afirmar si un problema es decidible. Usaremos algunos de sus resultados sobre máquinas de Turing. Supondremos, sin perdida de generalidad, que el modelo es no determinista de una banda finita en ambas direcciones.

Definición 4.1. *Una **máquina de Turing** no determinista M viene dada por,*

1. *Un alfabeto* $\Gamma := \{s_0, \ldots, s_n\}$ *de símbolos, donde* s_0 *es considerado el espacio en blanco.*
2. *Un conjunto finito* $Q = \{q_0, \ldots, q_p\}$ *de estados.*
3. *Una relación de transición* $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{l, r\}$.

Una maquina de Turing se puede interpretar de la siguiente manera. Imaginemos que tenemos una tira de papel dividida en infinitos cuadrados en ambas direcciones, y en cada cuadrado se encuentra un símbolo del alfabeto. Además de la tira de papel, se encuentra un marcador en uno de los cuadrados. A partir de la relación de transición que escojamos, el marcador se moverá de un lado a otro y cambiará los símbolos del papel.

Un ejemplo de relación de transición es $\{q_1, s_1, q_2, s_2, l\}$. El marcador empieza por el cuadrado inicial de la tira de papel. Si el marcador posee el estado q_1 y en ese cuadrado de papel se encuentra s_1, procedemos a aplicar el algoritmo. Cambiamos el estado del marcador por q_2 y el símbolo del trozo de papel por s_1. A continuación movemos el marcador a la izquierda (si fuera l) o derecha (si fuera r).

Llamaremos a un estado de la computación de la máquina, configuración. En la configuración de una máquina se incluye, la tira de papel, la posición y el estado del marcador. Podemos expresar $K \vdash_M K'$ si la configuración de K' se puede obtener mediante K con una computación de la máquina de Turing M.

De aquí surge el problema de la parada. Nos interesa saber si dado una maquina de Turing M y una configuración de la máquina K, termina. Es decir, que no ocurre $K \vdash_M K_1 \vdash_M K_2 \ldots$.

Volviendo al tema de la reescritura, en nuestro es caso el problema es un poco más difícil. Como podemos aplicar toda regla en cada momento, no tenemos una configuración K como en el problema de la parada. Pero si planteamos el problema suponiendo que no partimos de una configuración inicial K, la pregunta pasa a ser; ¿qué configuraciones K hace que el problema termine o no? A esto se le denota por el problema de la parada uniforme.

Para extrapolar lo que ya sabemos sobre máquinas de Turing a la reescritura, vamos a codificar las configuraciones como términos con una signatura Σ_M.

Definición 4.2. *Sea M una máquina de Turing, definimos*

$$\Sigma_M := \{\overrightarrow{s_0}, \ldots, \overrightarrow{s_n}, \overleftarrow{s_0}, \ldots, \overleftarrow{s_n}\} \cup \{q_0, \ldots, q_p\} \cup \{\overrightarrow{l}, r\}$$

donde cada función es de aridad 1.

A continuación, enunciamos una definición que relaciona términos con máquinas de Turing.

Definición 4.3. *Sea x_0 una variable fija. Un **término de configuración** en Σ_M es cualquier término de la forma,*

$$\overrightarrow{l}(\overrightarrow{s_{i_k}}(\ldots \overrightarrow{s_{i_1}}(q(\overleftarrow{s_{j_1}}(\ldots \overrightarrow{s_{j_h}}(\overleftarrow{r}(x_0))\ldots)))\ldots))$$

donde $k, h \geq 0$, $\{i_1, \ldots, i_k, j_1, \ldots, j_h\} \subseteq \{0, \ldots, n\}$, y $q \in Q$.

Para entender mejor esta definición, debemos hacer las siguientes consideraciones. El marcador se encuentra en s_{j_1} con el estado q. Los elementos que se encuentran a la derecha del marcador son $s_{j_2}, \ldots s_{j_h}$, y los elementos que se encuentran a su derecha, son $s_{i_1}, \ldots, s_{i_k}$.

Como la tira es infinita, $\overrightarrow{l}, \overleftarrow{r}$ nos ayudan a controlar los espacios vacíos. Si durante la computación de la máquina de Turing hiciese falta un espacio en blanco, estas funciones se lo proporcionan.

Acabamos de transformar una máquina de Turing en un término. En la siguiente definición, adaptaremos un sistema de reescritura para una máquina de Turing.

Definición 4.4. *Un **sistema de reescritura** R_M consiste en las siguientes reglas de reescritura,*

- *Para cada transición $(q, s_i, q', s_j, r) \in \Delta$, R_M contiene la regla,*

$$q(\overleftarrow{s_i}(x)) \to \overrightarrow{s_j}(q'(x))$$

  Si $i = 0$, entonces R_M contiene la siguiente regla,

$$q(\overleftarrow{r}(x)) \to \overrightarrow{s_j}(q'(\overleftarrow{r}(x)))$$

- *Para cada transición $(q, s_i, q', s_j, l) \in \Delta$, R_M contiene la regla,*

$$\overrightarrow{l}(q(\overleftarrow{s_i}(x))) \to \overrightarrow{l}(q'(\overleftarrow{s_0}(\overleftarrow{s_j}(x))))$$

y para cada $s_k \in \Gamma$ *la regla,*

$$\overrightarrow{s_k}(q(\overleftarrow{s_i}(x))) \to q'(\overleftarrow{s_k}(\overleftarrow{s_j}(x)))$$

Si $i = 0$*, entonces* R_M *contiene una regla adicional,*

$$\overrightarrow{l}(q(\overleftarrow{r}(x))) \to \overrightarrow{l}(q'(\overleftarrow{s_0}(\overleftarrow{s_j}(\overleftarrow{r}(x)))))$$

y para cada $s_k \in \Gamma$ *la regla*

$$\overrightarrow{s_k}(q(\overleftarrow{r}(x))) \to q'(\overleftarrow{s_0}(\overleftarrow{s_j}(\overleftarrow{r}(x))))$$

Con estas reglas podemos llegar a la conclusión de que para todo par de términos de configuración t, t', que verifiquen $t \to_{R_M} t'$, implica $K_t \vdash K_{t'}$. E igual ocurre en sentido contrario, si tenemos dos configuraciones K, K' y un término de configuración t, si $K \vdash K'$ y $K \equiv K_t$, esto implica que existe un término de configuración t' tal que $K' \equiv K_{t'}$ y $t \to_{R_M} t'$.

De aquí podemos razonar una propiedad. Como el problema de la parada de las máquinas de Turing es indecidible, y acabamos de probar que una máquina de Turing es equivalente a un sistema de reescritura (con las reglas que hemos pedido), aseguramos que dado un sistema de reescritura finito R y un término t, el problema de ver si todas las reducciones son terminantes empezando por t es indecidible.

Sin embargo, el problema que acabamos de resolver no es el original que queríamos. El problema de la terminación pide que todas las reducciones desde todos los posibles términos sean terminantes. Puede darse el caso que tengamos una configuración de términos que termine, pero una reducción no terminante que empiece por un término que no este en la configuración inicial. Por tanto el problema no esta resuelto todavía. El siguiente lema asegura que este caso no puede ocurrir.

Lema 4.1. Sea t un término de Σ_M. Si existe una reducción $t \to_{R_M} t_1 \to_{R_M} t_2 \to \ldots$, entonces existe un término de configuración t' y una reducción infinita R_M empezando por t'.

Demostración. Vamos a considerar $\Sigma_M = \overrightarrow{\Gamma} \cup \overleftarrow{\Gamma} \cup Q\{\overleftarrow{r}, \overrightarrow{l}\}$, donde $\overrightarrow{\Gamma} := \{\overrightarrow{s_1}, \ldots, \overrightarrow{s_n}\}$ y $\overleftarrow{\Gamma} := \{\overleftarrow{s_1}, \ldots, \overleftarrow{s_n}\}$. Cualquier elemento w de Σ_M puede ser escrito como composición de funciones tal que, $w = u_1 v_1 u_2 v_2 \ldots u_q v_q u_{q+1}$, ya que u_i, v_j son funciones de aridad 1.

Como todas las reglas de reescritura de R_M contienen un símbolo de Q, cualquier reducción aplicada a w, se hace dentro de las funciones v.

Es decir, que si tomamos $w(x) \to_{R_M} w'(x)$, entonces existe un índice j, $1 \leq j \leq q$, y una palabra $v'_j \in \overrightarrow{\Gamma}^* Q \overleftarrow{\Gamma}^*$ tal que $w' = u_1 v_1 u_2 v_2 \dots u_j v'_j u_{j+1} \dots u_q v_1 u_{q+1}$, y que $\overrightarrow{l} v_j \overleftarrow{r}\,(x_0) \to_{R_M} \overrightarrow{l} v'_j \overleftarrow{r}$

Como q es finito, esto implica que la reducción infinita que empieza por $w(x)$ produce una reducción infinita empezando por $\overrightarrow{l} v_j \overleftarrow{r}\,(x_0)$. Como $\overrightarrow{l} v_j \overleftarrow{r}\,(x_0)$ es un término de configuración, queda demostrado el lema. ∎

Finalmente, por el lema 4.1, obtenemos el objetivo principal de esta sección, la indecibilidad para el problema de terminación de los sistemas de reescritura.

Sistemas de reescritura básicos hacia la derecha

En esta subsección vamos a analizar un SRT particular R, donde sus reglas hacia la derecha son términos básicos. A estos SRT les llamaremos básicos hacia la derecha, y probaremos que son terminantes.

Lema 4.2. Sea R un SRT finito básico hacia la derecha. Entonces son equivalentes,

1. R no termina.
2. Existe una regla $l \to r \in R$ y un término t de manera que $r \xrightarrow{+}_R t$ y t contiene a r como subtérmino.

Demostración. $(2 \Rightarrow 1)$ es cierto pues obtenemos una reducción finita; $r \xrightarrow{+}_R t = t[r]_p \xrightarrow{+}_R t[t]_p = t[t[r]_p]_p \xrightarrow{+}_R \dots$, donde p es la posición $t|_p = r$.

$(1 \Rightarrow 2)$ se demuestra mediante inducción por el cardinal de R.

Suponemos que $|R| > 0 >$ y consideramos una reducción infinita $t_1 \to t_2 \to t_3 \to \dots$. Tenemos que probar que esta cadena no termina. Sin perdida de generalidad al menos una reducción ocurre en la posición ϵ. Esto significa que existe un índice i, una regla $l \to r \in R$, y una sustitución σ, tal que $t_i = \sigma(l)$ y $t_{i+1} = \sigma(r) = r$. Luego hay una reducción infinita $r \to_R t_{i+2} \to_R t_{i+3} \to_R \dots$ que empieza por r.

Si la regla $l \to r$ no es usada en la reducción, entonces aplicamos la inducción a $R - \{l \to r\}$. En el caso en que sí sea usada, eso significa que existe j tal que r ocurre en t_{i+j} y por tanto, obtenemos la segunda proposición.

Luego si R termina entonces $\to_R$ es globalmente finita, ya que es una ramificación finita por el lema 4.2 y el lema 1.1. Por tanto todas las reducciones terminarían y, al ser finitas, lo hacen en un numero finito de pasos. Con esta idea, obtenemos el resultado clave de esta subsección.

Teorema 4.1. *Para un SRT finito básico hacia la derecha, el problema de la terminación es decidible.*

Órdenes de reducción

En esta sección relacionaremos la definición de orden bien fundado con el problema de la terminación.

En la sección previa, hemos probado que el problema de la terminación es indecidible. Sin embargo podemos dar varios métodos para resolver el problema. Estos métodos no son totalmente automatizados.

La idea básica para probar la terminación, se hace mediante un orden bien fundado. Supongamos que R es un sistema de reescritura finito, $>$ es un orden estricto bien fundado en $T(\Sigma, X)$. Relacionando los sistemas de reescritura con los órdenes, si R es terminante, diremos que $s \to_R t$ implica $s > t$. En vez de decidir $s > t$, tan solo tendremos que comprobar las reglas de R. Para considerar este nuevo acercamiento al problema necesitamos pedir ciertas propiedades a $>$.

Definición 4.5. *Sea Σ una signatura y V un conjunto numerable de variables. Un orden estricto $>$ es un* ***orden de reescritura*** *syss*

1. *Es compatible con las Sigma-operaciones: para todo $s_1, s_2 \in T(\Sigma, V)$, todo $n \geq 0$ y $f \in \Sigma^{(n)}$, $s_1 > s_2$ implica*

$$f(t_1, \ldots, t_{n-1}, s_1, t_{i+1}, \ldots, t_n) > f(t_1, \ldots, t_{n-1}, s_2, t_{i+1}, \ldots, t_n)$$

$\forall i, 1 \leq i \leq n$ y $\forall t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \in T(\Sigma, V)$.

2. *Es cerrado bajo sustituciones:* $\forall \sigma \in Sub\ (T(\Sigma, V))$, *si* $s_1 > s_2$ *entonces* $\sigma(s_1) > \sigma(s_2)$.

Un orden de reducción es un orden de reescritura bien fundado.

El siguiente teorema es de especial importancia para los órdenes de reducción.

Teorema 4.2. *Un sistema de reescritura* R *termina syss existe un orden de reducción* $>$ *que satisface* $l > r$ *para toda regla* $l \to r \in R$.

Demostración. ($\Rightarrow$) Suponiendo que R termina, entonces $\xrightarrow{+}_R$ es un orden de reducción ya que satisface $l \xrightarrow{x}_R r$ para todo $l \to r \in R$.

($\Leftarrow$) Como $>$ es un orden de reescritura, para $l > r$ se verifica $t[\sigma(l)]_p > t[\sigma(r)]_p$ para todo término t, sustitución σ y posición p. Como $>$ esta bien fundada, y para toda regla de R, $s_1 \to_R s_2$ implica $s_1 > s_2$, entonces no puede ocurrir una reducción infinita $s_1 \to_R s_2 \to_R s_3 \ldots$.

A partir de aquí podemos dar varios métodos para dar una repuesta al problema. En las secciones posteriores nos centraremos en los ordenes de simplificación, orden de camino lexicográfico y orden de camino recursivo, y daremos una implementación de ambos.

Órdenes de simplificación

En esta sección definiremos que son los órdenes de simplificación y daremos dos órdenes construidos a partir de estos a modo de ejemplo en las siguientes subsecciones. Empezamos por la definición,

Definición 4.6. *Sea* $>$ *un orden estricto en* $T(\Sigma, V)$ *es un* ***orden de simplificación****, syss es un orden de reescritura que para todo término* $t \in T(\Sigma, V)$ *y todas las posiciones* $p \in Pos\ (t) - \{\epsilon\}$ *entonces* $t > t|_p$.

La definición es equivalente a pedir que, para todo $n \leq 1$, todos los símbolos de funciones $f \in \Sigma^{(n)}$, todas las variables $x_1, \ldots, x_n \in V$, tenemos que $f(x_1, \ldots, x_i, \ldots, x_n) > x_i$. Usando esta nueva definición e introduciendo algunos conceptos de álgebra de homomorfismos, se puede demostrar que los

órdenes de simplificación son equivalentes a los ordenes de reducción para Σ finito.

Órdenes de caminos lexicográficos

La idea de los caminos recursivos reside en comparar las raíces de los términos, y en comparar recursivamente sus subtérminos. Estos subtérminos se pueden comparar mediante multiconjuntos (orden de caminos de multiconjuntos), tuplas (orden de caminos lexicográficos), o una mezcla de ambos (orden de caminos recursivos). Nos interesaremos por estos dos últimos.

Definición 4.7. *Sea Σ una signatura finita y $>$ un orden estricto de Σ. El **orden de caminos lexicográficos** $>_{ocl}$, se define como, $s >_{ocl} t$ syss ocurre alguno de los siguientes casos,*

- *(OCL1) $t \in Var(s)$ y $s \neq t$ ó*
- *(OCL2) $s = f(s_1, \ldots, s_m)$, $t = g(t_1, \ldots, t_n)$, y*
  - *(OCL2a) existe i, $1 \leq i \leq m$, con $s_i \geq_{lpo} t$ ó*
  - *(OCL2b) $f > g$ y $s >_{ocl} t_j$ para todo j en $1 \leq j \leq n$ ó*
  - *(OCL2c) $f = g$, $s >_{lpo} t_j$ para todo j en $1 \leq j \leq n$, y exista i, $1 \leq i \leq m$ tal que $s_1 = t_1, \ldots, s_{i-1} = t_{i-1}$ y $s_i >_{ocl} t_i$*

La definición es recursiva y esta bien definida.

A continuación implementaremos el orden de caminos lexicográfico en Haskell. Para realizar los ejemplos de esta función, primero implementaremos `ordenPorLista xs a b`, que es el resultado de comparar a y b, tal que $a > b$ syss a aparece antes en xs que b. Por ejemplo,

```
>>> ordenPorLista ["a","b","c"] "a" "c"
GT
>>> ordenPorLista ["a","b","c"] "c" "b"
LT
>>> ordenPorLista ["a","b","c"] "b" "b"
EQ
```

Su código es,

```
ordenPorLista :: Ord a => [a] -> a  -> a -> Ordering
ordenPorLista [] _ _ =
  error("Ninguno de los dos elementos se encuentran
         en la lista")
ordenPorLista (x:xs) a b
  | a == x = if a == b
             then EQ
             else GT
  | b == x = LT
  | otherwise = ordenPorLista xs a b
```

Definiremos `ordCamLex ord s t`, que es el resultado de comparar s y t términos, con el orden de caminos lexicográfico inducido por `ord`. Por ejemplo

```
>>> let ord = ordenPorLista ["i","f","e"]
>>> ordenCamLex ord (T "f" [V ("x",1), T "e" []]) (V ("x",1))
GT
>>> ordenCamLex ord (T "i" [T "e" []]) (T "e" [])
GT
>>> ordenCamLex ord (T "i" [T "f" [V("x",1),V("y",1)]])
                     (T "f" [T "i" [V("y",1)], T "i" [V("x",1)]])
GT
>>> ordenCamLex ord (T "f" [V("y",1),V("z",1)])
                     (T "f" [T "f" [V("x",1),V("y",1)], V("z",1)])
LT
```

Su código es,

```
 ordenCamLex:: ([Char] -> [Char] -> Ordering)
                -> Termino -> Termino -> Ordering
ordenCamLex _ s (V x)
  | s == (V x) = EQ
  | ocurre x s = GT --OCL1
  | otherwise = LT
```

```
ordenCamLex _ (V _) (T _ _) = LT
ordenCamLex ord s@(T f ss) t@(T g ts) --OCL2
  | all (\x -> ordenCamLex ord x t == LT) ss
    = case ord f g of
      GT -> if all (\x -> ordenCamLex ord s x == GT) ts
            then GT --OCL2b
            else LT
      EQ -> if all (\x -> ordenCamLex ord s x == GT) ts
            then ordLex (ordenCamLex ord) ss ts --OCL2c
            else LT
      LT -> LT
  | otherwise = GT --OCL1a
```

Órdenes de caminos recursivos

La implementación de la anterior sección se puede generalizar, a lo que llamaremos órdenes de caminos recursivos. Para mayor claridad crearemos la función `ordenTermino t s`, que es es el resultado de comparar el nombre delelemento de la posición vacía mediante el orden alfabético. Por ejemplo,

```
>>> ordenTermino (V ("a",2)) (V ("b",1))
LT
>>> ordenTermino (V ("x",2)) (T "f" [V ("b",1)])
GT
>>> ordenTermino (T "g" [V("x",2)]) (T "f" [V("b",1)])
GT
```

Su código es,

```
ordenTermino::  Termino -> Termino -> Ordering
ordenTermino (V (a,_)) (V (b,_)) = compare a b
ordenTermino (V (a,_)) (T b _) = compare a b
ordenTermino (T a _) (V (b,_)) = compare a b
ordenTermino (T a _) (T b _) = compare a b
```

La generalización se basa en añadir un argumento adicional a la función, éste se encarga de especificar que orden se aplica en el apartado (OCL2c). Por ejemplo,

```
>>> let stat f (ordenTermino) t s = ordLex ordenTermino t s
>>> let ord = ordenPorLista ["i","f","e"]
>>> ordenCamRec stat ord (T "f" [V ("x",1), T "e" []]) (V ("x",1))
GT
>>> ordenCamRec stat ord (T "i" [T "e" []]) (T "e" [])
GT
>>> ordenCamRec stat ord (T "i" [T "f" [V("x",1),V("y",1)]])
                          (T "f" [T "i" [V("y",1)], T "i" [V("x",1)]])
GT
>>> ordenCamRec stat ord (T "f" [V("y",1),V("z",1)])
                          (T "f" [T "f" [V("x",1),V("y",1)], V("z",1)])
LT
```

Su código es,

```
ordenCamRec:: ([Char] -> (Termino -> Termino -> Ordering)
                      -> [Termino] -> [Termino] -> Ordering)
           ->  ([Char] -> [Char] -> Ordering)
           -> Termino
           -> Termino
           -> Ordering
ordenCamRec _ _ s (V x)
  | s == (V x) = EQ
  | ocurre x s = GT --OCR1
  | otherwise = LT
ordenCamRec _ _ (V _) (T _ _) = LT
ordenCamRec est ord s@(T f ss) t@(T g ts) --OCR2
  | all (\x -> ordenCamRec est ord x t == LT) ss
    = case ord f g of
      GT -> if all (\x -> ordenCamRec est ord s x == GT) ts
            then GT --OCR2b
            else LT
      EQ -> if all (\x -> ordenCamRec est ord s x == GT) ts
```

```
            then est f (ordenCamRec est ord) ss ts --OCR2c
            else LT
      LT -> LT
  | otherwise = GT --OCR2a
```

Capítulo 5

Confluencia

En este capítulo estudiaremos el problema de determinar si un sistema de reescritura es confluente. En la primera sección vamos a demostrar que este problema es indecidible para el caso general. Sin embargo en las secciones posteriores, estudiaremos que ocurre para un sistema terminante. Para este caso el problema será decidible. Por último veremos que podemos decir para los sistemas que no terminan.

Estudio sobre el problema de decisión

En esta sección veremos la indecibilidad para comprobar si un sistema es confluente mediante el siguiente resultado,

Teorema 5.1. *El problema de decidir si un sistema de reescritura finito R es confluente, es indecidible.*

Demostración. El objetivo de esta demostración es reducir el problema de las palabras básicas para E (que sabemos que es indecidible), a un sistema de reescritura de términos.

Sea un conjunto de identidades E tal que $\mathrm{Var}(l) = \mathrm{Var}(r)$ para todo $l \approx r \in E$. Sea $R := E \cup E^{-1}$, entonces al tener $\rightarrow_R = \leftrightarrow_E$ es confluente. Además R es un sistema de reescritura (por $\mathrm{Var}(l) = \mathrm{Var}(r)$).

Dados dos términos básicos s y t, y una constante a, vamos a probar que

$R_{st} := R \cup \{a \to s, a \to t\}$ es confluente syss $s \approx_E t$.

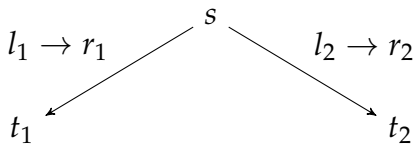
- ($\Rightarrow$) Si R_{st} es confluente, entonces ni s, ni t poseen una constante a, luego $s \downarrow_{R_{st}} t$. Esto significa que las reglas $a \to s, a \to t$ no pueden ser usadas. Por tanto $\to_R = \leftrightarrow_E$ y $s \approx_E t$.
- ($\Rightarrow$) Vamos a probar que para términos u, v, si $u \to_{R_{st}} v$ entonces $v^t \xrightarrow{*}_R u^t$, donde u^t denota el resultado de sustituir las constantes a en u por t. Supongamos que $u \to_{R_st} v$. Distinguimos que reglas se usan.
  - Si usamos $u \to_R v$, reemplazamos a por t para conseguir $u^t \to_R v^t$ y por tanto $v^t \to_R u^t$ al ser R simétrico.
  - Si usamos $a \to s$, entonces $u|_p = a$ y $v = u[s]_p$ para alguna posición p. Como $s \approx_E t$ entonces $s \xrightarrow{*}_R t$. Obtenemos $v \xrightarrow{*}_R u[t]_p$ que conlleva a $v^t \xrightarrow{*}_R (u[t]_p)^t$. Pero $(u[t]_p)^t = u^t[t^t]_p = u^t[t]_p = u^t$.
  - Si usamos $a \to t$ en la posición p, entonces $v^t = (u[t]_p)^t = u^t \xrightarrow{*}_R u^t$.

  Por tanto, si $u \xrightarrow{*}_{R_{st}} u_i$, para $i = 1, 2$, entonces $u_i \xrightarrow{*}_{R_{st}} u_i^t \xrightarrow{*}_R u^t$ y obtenemos $u_1 \downarrow_{R_{st}}$

Pares críticos

En esta sección estudiaremos la decibilidad para sistemas de reescritura finitos que sean localmente confluentes.

La necesidad de los **pares críticos** surge de la importancia a la hora de aplicar las reglas de un sistema de reescritura. Para entender mejor este concepto, plantearemos el siguiente ejemplo. Sea s un término, $R := \{l_1 \to r_1, l_2 \to r_2\}$ donde l_1, l_2 son subtérminos de s. Según apliquemos la primera regla o la segunda, obtendremos un nuevo término t.



Sea p_i las posiciones y σ_i las sustituciones tal que $s|_{p_i} = \sigma_i l_i$ y $t_i = s[\sigma_i r_i]_{p_i}, i = 1, 2$. Tenemos que estudiar como de relacionados están p_1 y p_2.

- **Caso 1**, p_1 y p_2 están en árboles separados. En este caso, se da la convergencia local. Por ejemplo, supongamos que partimos de la figura 5.1.

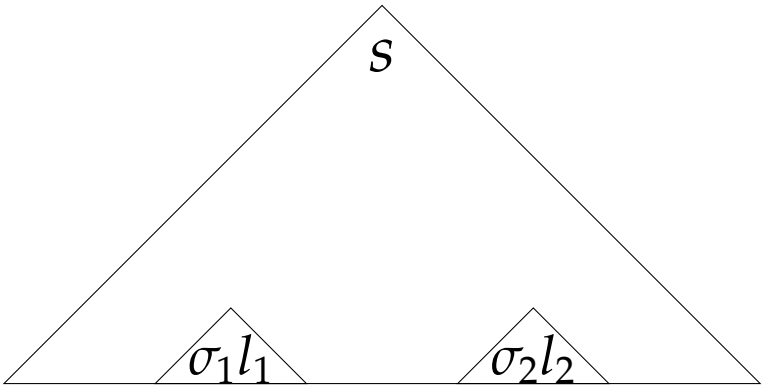


Figura 5.1: Término del caso 1 sin aplicar ninguna regla

Si aplicamos $l_1 \to r_1$ o $l_2 \to r_2$ obtenemos las figuras 5.2 y 5.3 respectivamente.

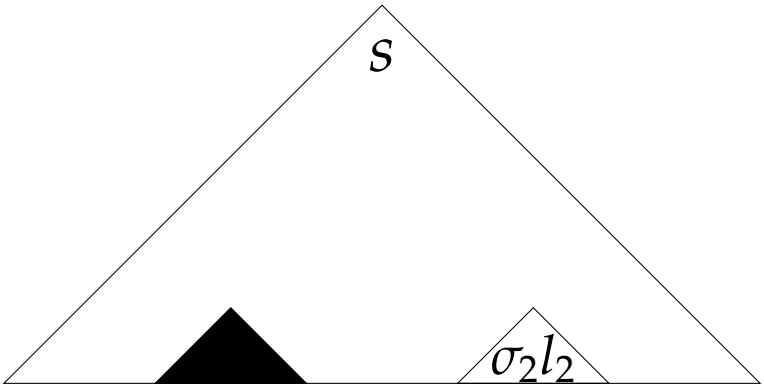


Figura 5.2: Término del caso 1 tras aplicar $l_1 \to r_1$

Y si usamos la regla que no se ha aplicado antes, para ambos términos llegamos a la figura 5.4.

- **Caso 2**. Si p_1 es un prefijo de p_2, tal y como se muestra en la figura 5.5
A partir del dibujo, podemos distinguir dos casos según como de separadas estén los términos $\sigma_1 l_1$ y $\sigma_2 l_2$.

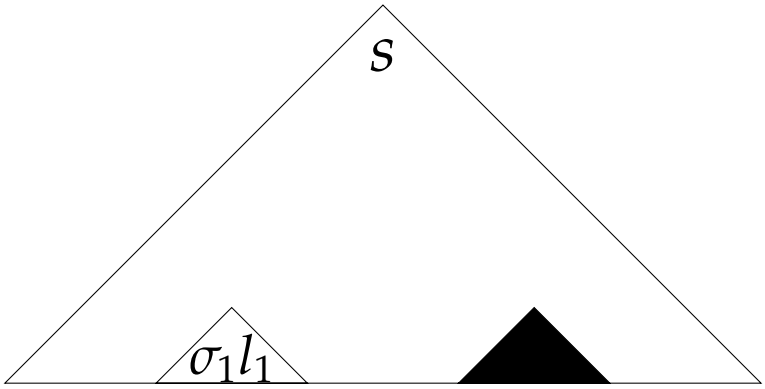


Figura 5.3: Término del caso 1 tras aplicar $l_2 \to r_2$

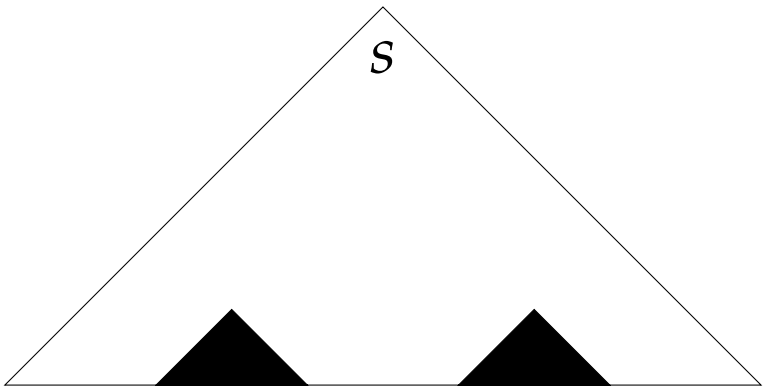


Figura 5.4: Término en el caso 1, donde no se produce solapamiento

- **Caso 2.1**. La reducción $\sigma_2 l_2$ no se solapa con l_1 pero está contenido en σ_1 (figura 5.6). A esta situación la llamaremos solapamiento no crítico, pues en esta situación ocurre una confluencia local.
- **Caso 2.2**. En este caso l_1 y l_2 se solapan (figura 5.7). A este caso lo llamaremos solapamiento crítico, y es la peor situación que podemos tener.

Definición 5.1. *Sea $l_i \to r_i$, $i = 1, 2$ dos reglas cuyas variables han sido renombradas tal que $Var(l_1, r_1) \cap Var(l_2, r_2) = \emptyset$. Sea $p \in Pos(l_1)$ tal que $l_1|_p$ no es una variable, y θ un umg de $l_1|_p =^? l_2$. Esto determinará el par crítico $\langle \theta r_1, (\theta l_1)[\theta r_2]_p \rangle$. Si dos reglas dan lugar a un par crítico, diremos que se* ***solapan****.*

A continuación enunciamos el Teorema de los Pares Críticos. Este resultado

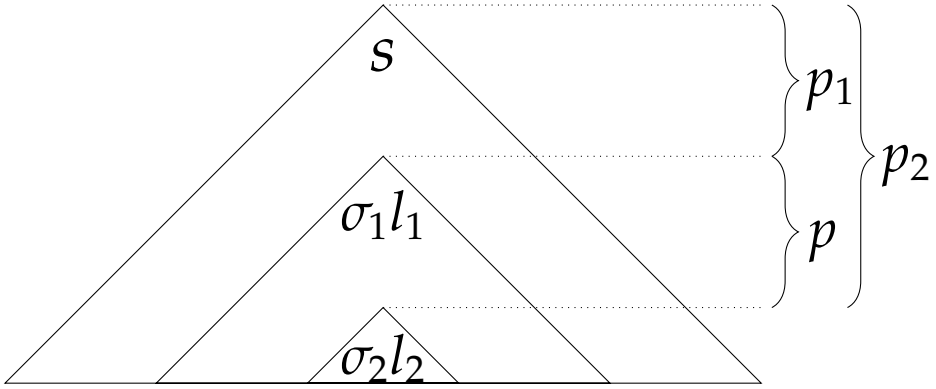


Figura 5.5: Situación del caso 2.

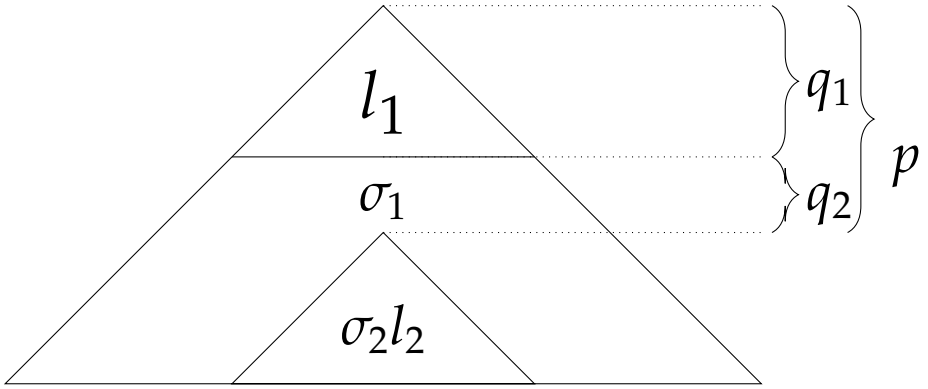


Figura 5.6: Caso 2.1.

así como la idea de la demostración, también se usará en la siguiente sección.

Teorema 5.2. *Un sistema de reescritura es localmente confluente syss todos sus pares críticos se vuelven a unir.*

Demostración. ($\Leftarrow$) Sabemos que si $s \to_R t$ para $i = 1, 2$, entonces es localmente confluente, ó $t_i = s[u_i]_p$, donde $\langle u_1, u_2 \rangle$ proviene de algún par crítico $\langle v_1, v_2 \rangle$, es decir $u_1 = \delta v_i$. Entonces $\xrightarrow{*} t$ para algún término, y por tanto $u_i \xrightarrow{\delta} t$ es también cierto. Esto implica $t_i \xrightarrow{s} [\delta t]_p$ para $i = 1, 2$.

($\Rightarrow$) Como el sistema de reescritura es localmente confluente, esto significa que cuando lleguemos a un par crítico este volverá a converger a un término. Luego todos sus pares críticos se vuelven a unir. ∎

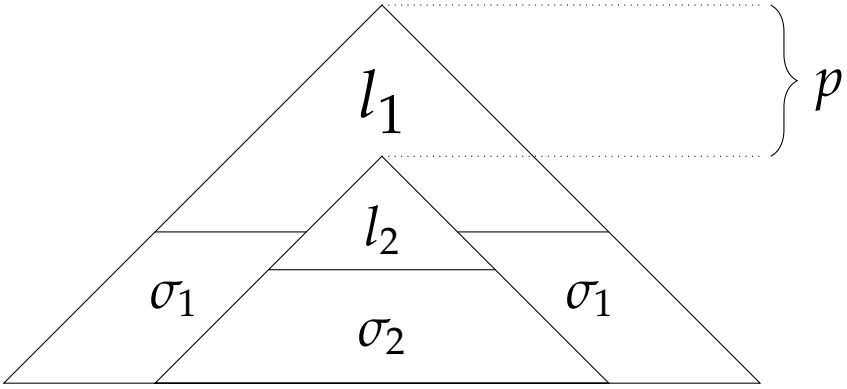


Figura 5.7: Caso 2.2.

Algunas propiedades directas de este teorema,

Corolario 5.1. Un sistema de términos terminante es confluente syss todos sus pares críticos se vuelven a unir

Pero si además pedimos que sea un sistema finito, obtenemos el siguiente resultado.

Corolario 5.2. La confluencia de un sistema de reescritura finito y terminante es decidible.

Demostración. El algoritmo que seguiremos será el siguiente:

Para cada par de reglas $l_1 \to r_1$ y $l_2 \to r_2$, y para cada $p \in \text{Pos}(l_1)$, tal que $l_1|_p$ no es una variable, se genera un par crítico unificando las variables disjuntas de $l_1|_p$ y l_2.

Para cada uno de esos pares críticos $\langle u_1, u_2 \rangle$ reducimos u_i a su forma normal $\tilde{u_i}$. Decimos que el sistema de reescritura es confluente syss $\tilde{u_1} = \tilde{u_2}$ para todos sus pares críticos.

Si se verifica $\tilde{u_1} = \tilde{u_2}$, entonces por el corolario 5.1, el sistema es confluente.

Si se da el caso de que existe algún par crítico no $\tilde{u_1} \neq \tilde{u_2}$, entonces se da la situación $\tilde{u_1} \overset{*}{\leftarrow} u_1 \leftarrow u \to u_2 \overset{*}{\to} \tilde{u_2}$, que demostraría que no es confluente. ∎

Implementación de los pares críticos

Se implementarán una función para calcular los pares críticos. Antes de poder definirla, se necesitarán varias funciones auxiliares. El código se puede encontrar en el fichero `ParesCriticos.hs`.

Vamos a necesitar la librería de términos.

```
import Terminos
```

- `(indiceMaximo t)` es el mayor índice que aparece en `t`. Por ejemplo,

  ```
  >>> indiceMaximo (V ("x",3))
  3
  >> indiceMaximo (T "w" [V("x",4), V("z", 200), V("y", 1)])
  200
  ```

  Su código es,

  ```
  indiceMaximo :: Termino -> Indice
  indiceMaximo (V (_,i)) = i
  indiceMaximo (T _ ts) = maximum (map indiceMaximo ts)
  ```

- `(renombraTermino n t)` es el termino resultante tras sumar a todos los índices de `t` el entero `n`. Por ejemplo,

  ```
  >>> renombraTermino 5 (V ("x",3))
  V ("x",8)
  >>> renombraTermino 3 (T "w" [V("x",4), V("z", 200), V("y", 1)])
  T "w" [V ("x",7),V ("z",203),V ("y",4)]
  ```

  Su código es,

  ```
  renombraTermino :: Int -> Termino -> Termino
  renombraTermino n (V (x,i)) = V(x,i+n)
  renombraTermino n (T f ts) = T f (map (renombraTermino n) ts)
  ```

- `(parCritico c (l1,r1) (l2,r2)` calcula el par crítico de la regla `l1` $\to$ `r1` y `l2` $\to$ `r2`. Por ejemplo,

```
>>> let c a = a
>>> parCritico c (V ("x",1), T "f" [V ("y",1)])
                 (V ("x",2),T "g" [V("y",2)])
[(T "f" [V ("y",1)],T "g" [V ("y",2)])]
>>> parCritico c (T "f" [V ("x",1)],T "f" [V ("y",1)])
                (V ("x",1), T "g" [V("y",2)])
[]
```

Su código es,

```
parCritico :: (Termino -> Termino) -> (Termino, Termino)
           -> (Termino, Termino) -> [(Termino, Termino)]
parCritico c (l1, r1) (l2, r2)
    | sigma1 == Left UNIFICACION = []
    | otherwise = [(sigma r1, sigma (c r2))]
    where sigma1 = unificacion l1 l2
          sigma = aplicaTerm (elimR(sigma1))
          elimR (Right a) = a
```

- `(paresCriticos rlista (l,r))` calcula la lista de todos los pares críticos formados al unificar `rlista` hacia la izquierda con un subtérmino `l`, donde `l` $\rightarrow$ `r` es una regla. Por ejemplo,

```
>>> paresCriticos [(V ("x",1), T "f" [V("y",1)])]
                  (V ("x",2), T "g" [V("y",2)])
[]
>>> paresCriticos [(T "f" [V ("x",1)], T "f" [V ("y",1)]),
                   (T "f"[V ("x",1)], T "g" [V ("x",2)]),
                   (T "g" [V ("x",2)],  T "f" [V ("y",1)])]
                   (V ("x",1), T "f" [V("y",1)])
[]
>>> paresCriticos [(T "f" [T "g" [V("x",1),V("y",1)],V("z",1)],
                    T "g" [V("x",1),V("z",1)]),
                   (T "g" [V("x",1),V("y",1)],V("x",1))]
                   (T "g" [V("x",1),V("y",1)],
                    T "g" [V("x",1),V("z",1)])
[(T "g" [V ("x",1),V ("z",3)],V ("x",1))]
```

Su código es,

```
paresCriticos :: [(Termino, Termino)] -> (Termino, Termino)
                 -> [(Termino, Termino)]
paresCriticos rlista (l,r) =
    cps (\x -> x) (renombraTermino m l) (renombraTermino m r)
    where cps _ (V _) _ = []
          cps c ter@(T f ts) r1 =
                concat(map (parCritico c (ter,r1)) rlista) ++
                 (pcint c f [] ts r1)(pcint c f [] ts r1)
          pcint _ _ _ [] _ = []
          pcint c f ts0 (t:ts1) r2 =
              (cps (\s -> c (T f (ts0++[s]++ts1))) t r2)
                           ++ (pcint c f (ts0++[t]) ts1 r2)
          m = maximum(map
               (\(l3,r3) ->
                maximum(indiceMaximo l3, indiceMaximo r3))
                rlista) + 1
```

- `(paresLista r)` es la lista de todos los pares críticos de `r`, incluyendo los triviales. Por ejemplo,

```
>>> paresLista [(V ("x",1), T "f" [V("y",1)])]
[]
>>> paresLista [(T "f" [T "g" [V("x",1),V("y",1)],V("z",1)],
                 T "g"[V("x",1),V("z",1)]),
                (T "g" [V("x",1),V("y",1)],V("x",1))]
[(T "g" [V ("x",1),V ("z",1)], T "g" [V ("x",1),V ("z",1)]),
 (T "g" [V ("x",1),V ("z",3)], T "f" [V ("x",1),V ("z",3)]),
 (V ("x",1),V ("x",1))]
```

  Su código es,

```
paresLista :: [(Termino, Termino)] -> [(Termino, Termino)]
paresLista r1 = paresLista2 r1 r1
```

- `(paresLista2 r1 r2)` es la lista de todos los pares críticos al unificar los términos a la izquierda de `r1` con un subtérmino a la izquierda con `r2`. Por ejemplo,

```
>>> paresLista2 [(V ("x",1), T "f" [V("y",1)])]
                [(V ("x",1), T "f"[V("y",1)])]
[]
>>> paresLista2 [(T "f" [T "g" [V("x",1),V("y",1)],V("z",1)],
                  T "g"[V("x",1),V("z",1)]),
                 (T "g" [V("x",1),V("y",1)],V("x",1))]
                [(V ("x",1), T "f" [V("y",1)])]
[]
>>> paresLista2 [(V ("x",1), T "f" [V("y",1)])]
                [(T "f" [T "g" [V("x",1),V("y",1)],V("z",1)],
                  T "g"[V("x",1),V("z",1)]),
                  (T "g" [V("x",1),V("y",1)],V("x",1))]
[(T "g" [V ("x",3),V ("z",3)],T "f" [V ("y",1)]),
 (T "g" [V ("x",3),V ("z",3)],
  T "f" [T "f" [V ("y",1)],V ("z",3)]),
 (V ("x",3),T "f" [V ("y",1)])]
```

Su código es,

```
paresLista2 :: [(Termino, Termino)] -> [(Termino, Termino)]
               -> [(Termino, Termino)]
paresLista2 r1 r2 = concat(map(paresCriticos r1) r2)
```

Ortogonalidad

En la sección anterior hemos analizado una técnica para determinar la confluencia para sistemas terminantes. En esta nos dedicaremos a analizar el problema para sistemas no terminantes. Para ello la estudiaremos desde otra perspectiva usando nuevamente los pares críticos.

Definición 5.2. *Una regla de reescritura $l \to r$ es **lineal por la izquierda** (respectivamente **lineal por la derecha**) si ninguna variable ocurre dos veces en l (respectivamente r). Si una regla es lineal por la izquierda y por la derecha diremos que es **lineal**.*

Y ampliamos la definición para sistemas de reescritura.

Definición 5.3. *Un sistema de reescritura es **lineal por la izquierda** (respectivamente **por la derecha**) si todas sus reglas son lineales por la izquierda (respectivamente por la derecha)*

Por último, antes de dar el principal resultado de la sección, definimos una restricción que nos servirá para demostrar la confluencia fuerte.

Definición 5.4. *Los términos s_1 y s_2 son **fuertemente unibles** respecto $\to$ si existen términos t_1 y t_2 tal que $s_1 \xrightarrow{=} t_1 \xleftarrow{*} s_2$ y $s_1 \xrightarrow{*} t_2 \xleftarrow{=} s_2$*

El siguiente resultado se conoce como el lema de confluencia fuerte.

Lema 5.1. Si R es lineal y cada par crítico de R es fuertemente unible, entonces R es fuertemente confluente.

Demostración. La prueba se hará reduciendo cada uno de los casos que estudiamos en la sección anterior.

- Caso 1. Es trivial.
- Caso 2.1. Se simplifica mediante la restricción de linealidad
- Caso 2.2. Como todos los pares críticos son fuertemente unible, la instancia de un par crítico es fuertemente unible

Si un sistema es fuertemente confluente, en particular, también es confluente. Por tanto acabamos de probar una condición suficiente para la confluencia.

Definición 5.5. *Un sistema de reescritura es **ortogonal** si es lineal por la izquierda y no tiene pares críticos*

Probar la confluencia para un sistema ortogonal posee el siguiente problema; al no poseer linealidad por la derecha, en el caso 2.1 es posible que se aplique mas de una vez la reducción $l_2 \to r_2$, y por tanto, lo único que podemos deducir es la confluencia local. Para evitar este problema, usaremos una relación de reescritura que posea la propiedad del diamante, que recordemos que el cumplimiento de esta propiedad implica la confluencia.

Volviendo al caso 2.1, para conseguir la propiedad del diamante tenemos que arreglar el problema de aplicar repetidas veces la reducción $l_2 \to r_2$. Para

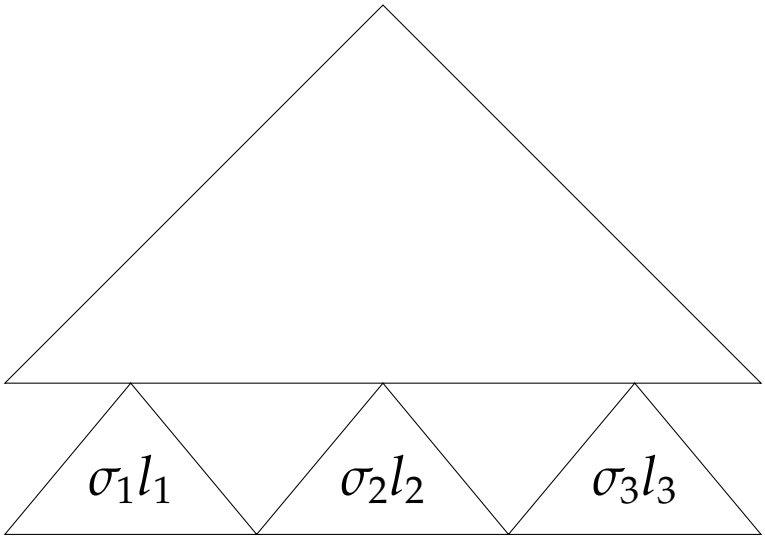


Figura 5.8:

ello vamos a considerar reducciones paralelas para los subárboles. Es decir, si tenemos la figura 5.8, consideramos la reducción $\rightrightarrows$ con la que obtenemos la situación 5.9.

Definición 5.6. *Diremos que un conjunto de posiciones es* ***paralelo*** *si $p||q$ para todo $p, q \in P$, donde P es un subconjunto de todas las posibles posiciones paralelas.*

Dado un término t_p para cada $p \in P$, definimos la notación,

$$s[t_p]_{p \in P} := s[t_{p_1}]_{p_1} \dots [t_{p_n}]_{p_n}$$

Definición 5.7. *Si para cada $p \in P$ tenemos la regla $l_p \to r_p in R$ y para cada sustitución σ_p tal que $s|_p = \sigma_p l_p$ llamaremos* ***reducción paralela*** *a $s \rightrightarrows_R^P s[\sigma_p r_p]_{p \in P}$*

A partir de esta definición, enunciamos el teorema de los sistemas ortogonales

Teorema 5.3. *Si R es ortogonal entonces $\rightrightarrows_R$ tiene la propiedad del diamante*

Demostración. Sea $s \rightrightarrows^{P_i} t_i, i = 0, 1$. Para esta prueba vamos a considerar una partición de $P_i = A_i \cup B_i \cup C$ donde

- $A_i := \{p \in P_i | \nexists q \in P_{1-i}, q \leq p\}$
- $B_i := \{p \in P_i | \exists q \in P_{1-i}, q < p\}$

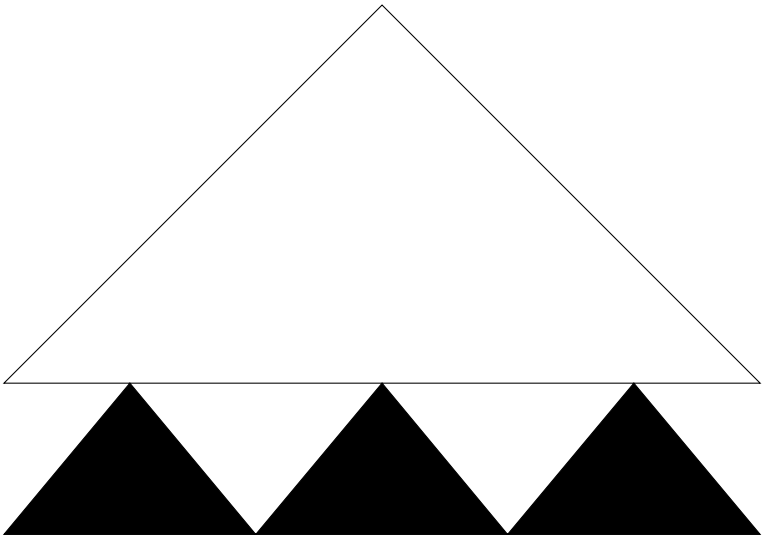


Figura 5.9:

- $P_0 \cap P_1$

Analizamos que elementos hay en cada partición; A_i contiene todas las posiciones contraídas que estén por encima de $s \rightrightarrows t_{1-i}$, B_i contiene las que están por debajo estrictamente de las contraídas en $s \rightrightarrows t_{1-i}$, y C un conjunto de posiciones contraídas en ambas reducciones.

Como no tenemos pares críticos, todos los b_i son el resultado de aplicar una regla de reducción hacia la derecha a a_{1-i}.

La reducción $t_{1-i} \rightrightarrows u$ tiene las mismas reglas que las posiciones B_i. Lo mismo ocurre para $s \rightrightarrows t_i$. Nuestro objetivo es llegar a que $t_{1-i} \rightrightarrows u$ también usa las posiciones de A_i, para llegar a formar el diamante.

Una reducción en la posición $a_i \in A_i$, no puede ser destruida por una reducción de algún $b_{1-i} \in B_{1-i}$ porque R es lineal por la izquierda. Por tanto la reducción $t_{1-i} \rightrightarrows u$ usa las mismas reglas en las posiciones A_i en la reducción $s \rightrightarrows t_i$ (que aplicadas en las posiciones de B_i).

Y en particular, tenemos el siguiente corolario.

Corolario 5.3. Todo sistema de reescritura ortogonal es confluente.

Capítulo 6

Completación

Durante los anteriores capítulos hemos estudiado la decibilidad para el problema de la terminación y confluencia. Hemos probado que ambos problemas son indecidibles para el caso general. En este capítulo vamos a estudiar un procedimiento, para que dado un sistema de reescritura, modificar sus elementos para lograr que sea terminante y confluente. Formalmente queremos construir un prodecimiento para el problema de la palabra para un sistema finito de identidades E.

Supongamos que tenemos un conjunto $E := \{x + 0 \equiv x, x + s(y) \equiv s(x + y)\}$. El sistema de reescritura generado por E es $R := \{x + 0 \to x, x + s(y) \to s(x + y)\}$. Para probar la termiación usamos el camino de orden lexicográfico inducido por $>$. La confluencia se verifica pues R no tiene pares críticos.

Para estudiar este conjunto hemos aplicado condiciones suficientes, es decir, podemos tener un sistema de identidades al que no podamos aplicar las propiedades que hemos estudiado, y sin embargo sea terminante y/o confluente. La idea detrás de los algorítmos es añadir reglas que no cambien la estructura equacional de E.

Por ejemplo, supongamos que tenemos un sistema E que es terminante pero no confluente. Eso quiere decir que existen pares críticos en el sistema de reescritura R asociado a E. Si añadimos alguna regla adiccional, podemos conseguir que esos pares críticos sean unibles.

Hay que tener en cuenta que al añadir reglas adiccionales se crearán nuevos

pares críticos. Este proceso continuará hasta que no existan más pares críticos.

Algoritmo de completación

Vamos a definir un proceso que empieza por un conjunto finito de identidades E, que intenta encontrar un sistema de reescritura convergente R que sea equivalente a E.

Algoritmo

Input: Un conjunto finito E de Σ-identidades y un orden de reducción ¿ en $T(\Sigma, V)$.

Output: Si el procedimiento termina; un sistema de reescritura R finito y convergente que es equivalente a E. En caso contrario devuelve un `Error`.

Inicialización: Si existe una identidad $(s \equiv t) \in E$ tal que $s \neq t, s \not> t$ y $t \not> s$, entonces termina con `Error`. En otro caso, $i := 0$ $R_0 := \{l \to r | (l \equiv r) \in E \cup E^{-1} \wedge l > r\}$.

`Mientras` $R_i \neq R_{i-1}$

$R_{i+1} := R_i$;

Para todo $\langle s, t \rangle \in$ `ParesCriticos(`R_i`)`:

$>$ Reducir s, t a forma normal s', t'

$>$ Si $s' \neq t'$, $s' \not> t'$ y $t' \not> s'$ terminar con `Error`

$>$ Si $s' > t'$ entonces $R_{i+1} := R_{i+1} \cup \{s' \to t'\}$

$>$ Si $t' > s'$ entonces $R_{i+1} := R_{i+1} \cup \{t' \to s'\}$

$i := i + 1$

Devuelve R_i

Primero eliminamos todas las identidades $t = t$ y calculamos todos los pares críticos y vamos recorriendo la lista. Cada término $\langle s, t \rangle$ se reduce a su

forma normal s' y t'. Si estos últimos son iguales, el par crítico es unible y se pasa al siguiente. En caso contrario, transformamos el par crítico en una regla de reescritura. Se comprueba la terminación de esa regla con $>$. Si esta última se verifica, se añade al sistema de reescritura. El proceso termina cuando el sistema no tenga más pares críticos unibles, ó produzca un `Error`.

El algoritmo puede producir varias situaciones; que devuelva un sistema de reescritura en la interacción donde todos los pares críticos son unibles, que produzca un `Error`, o que no pare pues se generen infinitas nuevas reglas.

Por último, vamos a demostrar que el resultado del algoritmo es una solución para el problema de la palabra.

Teorema 6.1. *Sea E un conjunto finito de identidades, y $>$ un orden de reducción.*

1. *Si el resultado del algoritmo aplicado a E y al orden $>$ termina y devuelve R_n, entonces R_n es un sistema de reescritura finito convergente equivalente a E. Además R_n es un procedimiento para el problema de la parada de E.*
2. *Si el resultado del algoritmo aplicado a E y al orden $>$ no termina, entonces $R_\infty := \cap_{i\geq 0} R_i$ es un sistema de reescritura infinito convergente que es equivalente a E.*

Demostración. (1) R_n es finito pues se construye mediante iteraciones finitas, donde cada interacción añade reglas $l \to r$ que satisfacen $l > r$. Por tanto por construcción R_n es terminante. Además es confluente pues todos sus pares críticos son unibles.

Por doble inclusión probaremos $\approx_{R_n} = \approx_E$. La primera inclusión se da pues $\approx_{R_0} = \approx_E$ es cierto y $R_0 \subseteq R_1 \subseteq \cdots \subseteq R_n$.

La segunda se demuestra mediante inducción para $\approx_{R_i} \subseteq \approx_E$ en i. El caso base es trivial. Para el paso de la inducción, usaremos que $s \approx_R t$ es cierto para cada par crítico $\langle s, t\rangle$ de R_i, y como la forma normal s', t' de s y t, verifica $s \approx_{R_i} s'$ y $t \approx_{R_i} t'$. Por tanto, tenemos $s' \approx_R t'$ que implica $s' \approx_E t'$ por la hipótesis de inducción, y que $\approx_{R_i} \subseteq \approx_E$ es decidible por 3.1.

(2) R_∞ es un sistema infinito donde en cada interacción se añade al menos una regla. La equivalencia de R_∞ y E, y la terminación de R_∞ se prueba de manera análoga a (1).

Para probar la confluencia de R_∞ vamos a probar que todos los pares críticos son unibles. Sea $\langle s, t\rangle$ un par crítico entre las reglas $l \to r$, $g \to d \in R_\infty$.

Como $R_\infty = \cup_{i \geq 0} R_i$ y $R_0 \subseteq R_1 \subseteq \ldots$, esto significa que existe $n \geq 0$ tal que $l \to r, g \to d \in R_n$, que implica que $\langle s, t \rangle \in \texttt{paresCriticos}(R_n)$. Este par es unible en R_n ó es unible en R_{n+1}, pues se le habrá añadido la correspondiente regla. Por tanto se da la convergencia.

Mejorando la completación

El anterior algoritmo suele generar un gran número de reglas. Solo hay que ver que para cada par crítico surgen varias. Además hay que hacer el calculo de los pares críticos en cada paso, lo que hace que el algoritmo sea muy costoso tanto para el tiempo como para la memoria.

Para mejorar el algoritmo de completación, necesitamos varios resultados que nos aseguren de que el proceso es correcto. En la siguiente sección hablaremos del algoritmo de Huet. Este algoritmo es algo mas complejo que el anterior y necesitaremos estos resultados para comprobar que efectivamente, el proceso funciona.

La idea se basa en extender las reglas de simplificación por un conjunto de reglas que nos resulten mas beneficiosas. Las reglas son las siguientes.

`Deducir`	$\dfrac{E, R}{E \cup \{s \approx t\}, R}$	Si $s \leftarrow_R u \to_R t$
`Orientar`	$\dfrac{E \cup \{s \approx' t\}, R}{E, R \cup \{s \to t\}}$	Si $s > t$
`Eliminar`	$\dfrac{E \cup \{s \approx s\}, R}{E, R}$	
`Simplificar identidad`	$\dfrac{E \cup \{s \approx' t\}, R}{E \cup \{u \approx t\}, R}$	Si $s \to_R u$
`D-simplificar regla`	$\dfrac{E, R \cup \{s \to t\}}{E, R \cup \{s \to u\}}$	Si $t \to_R u$
`I-simplificar regla`	$\dfrac{E, R \cup \{s \to t\}}{E \cup \{u \approx t\}, R}$	Si $s \stackrel{\sqsupset}{\to}_R u$

La notación $s \approx' t$ indica que es un par no ordenado, es decir $s \approx t \in E$ ó $t \approx s \in R$. Denotaremos $(E, R) \vdash_C (E', R')$ si (E, R) se puede transformar en (E', R') aplicando alguna regla anterior.

Las reglas generan un sistema terminante de reescritura terminante, pues todas están orientadas con el orden de reducción $>$, de lo que deducimos el siguiente resultado.

Lema 6.1. Si $R \subseteq >$ y $(E, R) \vdash_C (E', R')$ entonces $R' \subseteq >$.

Una consecuencia directa de este lema es dado un sistema de reescritura R del par (E, R) es terminante si este par se ha obtenido por uno inicial de la forma $(E_0, \emptyset)$ aplicando las reglas.

El siguiente resultado demuestra que las reglas no cambian la estructura ecuacional.

Lema 6.2. $(E_1, R_1) \vdash_C (E_2, R_2)$ implica $\approx_{E_1 \cup R_1} = \approx_{E_2 \cup R_2}$.

Demostración. La demostración es trivial para las tres primeras reglas, por lo que nos centraremos en las tres últimas.

- Para `Simplificar identidad`, partiendo de que $E_1 = E \cup \{s \approx' t\}$, $E_2 = E \cup \{u \approx t\}$, $R_1 = R = R_2$, y $s \to_R u$, que implica $u \approx_{E_1 \cup R_1} t$ y $\approx_{E_2 \cup R_2} \subseteq \approx_{E_1 \cup R_1}$. Para la otra inclusión partimos de $u \equiv t \in E_2$, $s \to_R u$ y $R = R_2$ implica $s \approx_{E_2 \cup R_2} t$ y por tanto, $\approx_{E_1 \cup R_1} \subseteq \approx_{E_2 \cup R_2}$.
- Para `D-simplificar regla`, tenemos $E_1 = E = E_2$, $R_1 = R \cup \{s \to t\}$, $R_2 = R \cup \{s \to u\}$ y $t \to_R u$. Con $s \to t \in R_1$, $t \to_R u$ y $R \subseteq R_1$ conseguimos $s \approx_{E_1 \cup R_1} u$, y con $s \to u \in R_2$, $t \to_R u$, y $R \subseteq R_2$ implica $s \approx_{E_2 \cup R_2} t$. Esto demuestra $\approx_{E_1 \cup R_1} = \approx_{E_2 \cup R_2}$.
- Para `I-simplificar regla` se demuestra de manera análoga

Usando la idea del procedimiento de completación que hemos comentado antes, formalizamos su definición.

Definición 6.1. *Un procedimiento de completación es un programa que acepta un conjunto finito de identidades E_0, un orden de reducción $>$, y usa las reglas para generar una secuencia (finita o infinita),*

$$(E_0, R_0) \vdash_C (E_1, R_1) \vdash_C (E_2, R_2) \vdash_C (E_3, R_3) \vdash_C \ldots$$

donde $R_0 := \emptyset$. *Además definimos* E_ω *como el conjunto de identidades persistentes y* R_ω *como el conjunto de reglas persistentes,*

$$E_\omega := \bigcup_{i \geq 0} \bigcap_{j \geq i} E_j$$

$$R_\omega := \bigcup_{i \geq 0} \bigcap_{j \geq i} R_j$$

Diremos que la secuencia es un éxito syss $E_\omega = \emptyset$ y R_ω es convergente y equivalente a E_0. Diremos que falla syss $E_\omega \neq \emptyset$. Un procedimiento de completación es correcto syss cada secuencia que no falle, sea un éxito.

Por último, la definición siguiente recoge la importancia de los pares críticos a la hora de analizar los procedimientos de completación.

Definición 6.2. *Una secuencia de completación se dirá justa syss*

$$\texttt{paresCriticos}(R_\omega) \subseteq \bigcup_{i \geq 0} E_i$$

Un procedimiento de completación es justo syss cada secuencia de éxito es justa.

Y enunciamos el resultado que nos ayudará a probar que el algoritmo de Huet resuelve el problema de completación.

Teorema 6.2. *Cada procedimiento de completación justo es correcto.*

Algoritmo de Huet

El algoritmo que vamos a enunciar es una extensión no trivial del algorítmo básico de completación.

Algoritmo de Huet

Input: Un conjunto finito E de Σ-identidades y un orden de reducción ¿ en $T(\Sigma, V)$.

Output: Si el procedimiento termina; un sistema de reescritura R infinito limitado y convergente que es equivalente a E. En caso contrario devuelve un `Error`.

Inicialización: $R_0 := \emptyset, E_O := E, i := 0$;

`Mientras` $E_i \neq \emptyset$ ó exista una regla no marcada en R_i

`Mientras` $E_i \neq \emptyset$

(a) Escoge una identidad $s \approx t \in E_i$

(b) Reduce s, t a una forma R_i-normal s', t'

(c) Si $s' = t'$ entonces,

$R_{i+1} := R_i$

$E_{i+1} := E_i - \{s \approx t\}$

$i := i + 1$

(d) Si no, si $s' \not> t'$ y $t' \not> s'$, entonces terminar con un output `Error`.

(e) Si no, sea l, r tal que $\{l, r\} = \{s', t'\}$ y $l > r$;

$R_{i+1} := \{g \to d' \mid g \to d \in R_i, g$ no puede reducirse con $l \to r$, y d' es una forma normal de d respecto $R_i \cup \{l \to r\}\}$

$E_{i+1} := (E_i - \{s \approx t\}) \cup \{g' \approx d \mid g \to d \in R_i$ y g puede ser reducido a g' con $l \to r\}$.

i := i + 1

Si existe una regla no marcada en R_i, sea $l \to r$ esa regla.

$R_{i+1} := R_i$

$E_{i+1} := \{s \approx t \mid \langle s, t\rangle$ es un par crítico de $l \to r$ ó una regla marcada en $R_i\}$

$i := i + 1$

Marca $l \to r$.

Devuelve R_i

Comprobamos que el algoritmo efectivamente realiza un procedimiento de completación sobre E

Lema 6.3. El algoritmo de completación de Huet es un procedimiento de completación.

Demostración. El objetivo de la prueba es comprobar que cada paso del algoritmo es una de las reglas de la anterior sección.

La computación de los pares críticos en el primer `Mientras` se puede realizar mediante `Deducir`, el paso (b) con `Simplificar identidad`, (c) con `Eliminar` y (e) con `Orienta`, `R-simplificar regla` y `L-simplificar regla`.

Debemos comprobar que si el algoritmo devuelve un `Error`, el procedimiento de completación falla

Lema 6.4. Una cadena del algoritmo de Huet falla syss termina con output `Fail`.

Demostración. Si el procedimiento de Huet termina con `Error`, entonces hay una identidad que no puede transformarse por la regla de simplificación y orientación, entonces $E_\omega \neq \emptyset$. Si el procedimiento no termina con `Error`, entonces no hay identidades y por ello, el bucle interior siempre termina. Por ello vamos a asociar cada iteracción del bucle como un multiconjunto de pares,

$$K_i := \{(\{s,t\},1)|s \approx t \in E_i\} \cup \{(\{l,r\},0)|l \to r \in R_i\}$$

Donde la primera componente esta ordenada usando el orden de multiconjuntos inducido por el orden de reducción $>$, y la segunda componente por el producto del orden lexicográfico.

Como los multiconjuntos K_i están ordenados usando una extensión de este orden de pares, este esta bien fundando y $K_i > K_{i+1}$. Por tanto el bucle termina y se demuestra el resultado.

Implementación del algoritmo de Huet

La implementación del algoritmo de Huet se realizará por partes.

- `(minRegla rl n r1 r2)` divide un conjunto de reglas en dos, donde uno tiene el tamaño de `r1`. Por ejemplo,

```
>>> minRegla (V("x",1),V("x",2)) 2 [(V("y",1),V("y",2))] []
((V ("x",1),V ("x",2)),[(V ("y",1),V ("y",2))])
>>> minRegla (V("x",1),V("x",2)) 1 [(V("y",1),V("y",2))] []
((V ("x",1),V ("x",2)),[(V ("y",1),V ("y",2))])
>>> minRegla (V ("x",1),V ("x",2)) 2
             [(V ("y",1),V ("y",2)),
              (T "f" [V("z",1),V("z",2)], V ("x",2))] []
((V ("x",1),V ("x",2)),
 [(T "f" [V ("z",1),V ("z",2)],V ("x",2)),(V ("y",1),V ("y",2))])
>>> minRegla (V ("x",1),V ("x",2)) 4 [(V ("y",1),V ("y",2)),
             (T "f" [V("z",1),V("z",2)], V ("x",2))] []
((V ("y",1),V ("y",2)),
 [(T "f" [V ("z",1),V ("z",2)],V ("x",2)),(V ("x",1),V ("x",2))])
```

Su código es,

```
minRegla :: (Termino, Termino) -> Int -> [(Termino, Termino)]
     -> [(Termino, Termino)]
     -> ((Termino, Termino), [(Termino, Termino)])
minRegla rl _ [] r2 = (rl,r2)
minRegla rl n ((l,r):r1) r2
    | m < n = minRegla (l,r) m r1 (rl:r2)
    | otherwise = minRegla rl n r1 ((l,r):r2)
    where m = longitudTerm l + longitudTerm r
```

- `(escogeRegla r)` calcula la regla que hay que aplicar durante el algoritmo de Huet. Por ejemplo,

```
>>> escogeRegla [(V("y",1),V("y",2))]
((V ("y",1),V ("y",2)),[])
>>> escogeRegla [(V ("y",1),V ("y",2)),
                 (T "f" [V("z",1),V("z",2)], V ("x",2))]
((V ("y",1),V ("y",2)),[(T "f" [V ("z",1),V ("z",2)],V ("x",2))])
```

Su código es,

```
escogeRegla :: [(Termino, Termino)]
            -> ((Termino, Termino), [(Termino, Termino)])
escogeRegla [] = error("No se puede aplicar ninguna regla")
escogeRegla ((l,r):r1) = minRegla (l,r)
                          (longitudTerm l + longitudTerm r)
                          r1 []
```

- `(anadeRegla (l,r) e s r)` es el paso (e) del algoritmo de Huet.

```
anadeRegla :: (Termino, Termino) -> [(Termino, Termino)]
              -> [Ecuacion] -> [Ecuacion]
              -> ([(Termino, Termino)], [(Termino, Termino)],
                  [(Termino, Termino)])
anadeRegla (l,r) e s r1 = (e2, (l,r):s1, r2)
    where (e1,s1) = simpl l r s r1 s e []
          (e2,r2) = simpl l r s r1 r1 e1 []
simpl
  :: Termino
     -> Termino
     -> [Ecuacion]
     -> [Ecuacion]
     -> [(Termino, Termino)]
     -> [(Termino, Termino)]
     -> [(Termino, Termino)]
     -> ([(Termino, Termino)], [(Termino, Termino)])
simpl _ _ _ _ [] a b = (a,b)
simpl l r s r1 ((g,d):u) e1 u1
    | g1 == g =
        simpl l r s r1 u e1 ((g, d1):u1)
    | otherwise = simpl l r r1 s u ((g1,d):e1) u1
       where g1 = formaNormal [(l,r)] g
             d1 = formaNormal ((l,r):(r1++s)) d
```

- `(orienta ord trip)` es el segundo bucle del algoritmo de Huet.

```
orienta :: (Termino -> Termino -> Ordering)
           -> ([(Termino, Termino)],  [(Termino, Termino)],
```

```
                [(Termino, Termino)])
           -> ([(Termino, Termino)], [(Termino, Termino)])
orienta _ ([],ss,rr) = (ss,rr)
orienta ord ((s,t):ee,ss,rr)
    | s1 == t1 = orienta ord (ee,ss,rr)
    | ord s1 t1 == GT = orienta ord (anadeRegla (s1,t1)
                                                ee ss rr)
    | ord t1 s1 == GT = orienta ord (anadeRegla (t1,s1)
                                                ee ss rr)
    | otherwise = error("Error con la funcion ord")
    where s1 = formaNormal (rr++ss) s
          t1 = formaNormal (rr++ss) t
```

- `(completa ord ee)` es el resultado de aplicar a ee, el algoritmo de completación de Huet. Por ejemplo,

```
>>> let ord = ordenCamLex (ordenPorLista ["a","b","c","x","y","z"])
>>> completa ord [(V("x",1),V("x",1))]
[]
>>> completa ord [(V ("y",1),V ("y",2)),
                  (T "f" [V("y",1),V("y",2)], V ("x",2))]
*** Exception: Error con la funcion ord
>>> completa ord [(T "f" [V("x",1),V("x",2)], V ("x",2))]
[(T "f" [V ("x",1),V ("x",2)],V ("x",2))]
```

Su código es,

```
completa :: (Termino -> Termino -> Ordering)
            -> [(Termino, Termino)] -> [(Termino, Termino)]
completa ord ee = compl ord (ee,[],[])
compl :: (Termino -> Termino -> Ordering)
         -> ([(Termino, Termino)], [(Termino, Termino)],
             [(Termino, Termino)])
         -> [(Termino, Termino)]
compl ord (ee,ss,tt) = case orienta ord (ee,ss,tt) of
                         ([],rr1)  -> rr1
                         (ss1,rr1) -> compl ord
                                      (paCr,ss2,rl:rr1)
      where (rl, ss2) = escogeRegla ss1
            paCr = (paresLista2 [rl] rr1)++
```

```
                    (paresLista2 rr1 [rl])++
                    (paresLista2 [rl] [rl])
```

Sistemas utilizados

Durante la realización de este Trabajo de Fin de Grado he usado los siguientes sistemas y paquetes.

- **Ubuntu 16.04 LTS.** Instalé *Ubuntu* en una partición que hice al disco duro de mi ordenador portátil. La instalación se realizó mediante la herramienta *LinuxLive USB Creator*.
- **LaTeX .** Descargué la distribución de LaTeX y *Tex Live* utilizando el *Gestor de Paquetes Synaptic*. Debo destacar dos paquetes fundamentales que me han sido de gran utilidad;
  - **Paquete AUCTex.** Anteriormente, solo había usado el editor, *TeXstudio* para la creación de documentos LaTeX, pero gracias a mi tutor José Antonio, me animé a probar *Emacs*. Mediante el comando `M-x list-packages` de *Emacs* me descargué este paquete desde el repositorio *MELPA*. Gracias a *AUCText* pude trabajar en *Emacs* y obtener diversos paquetes del editor que me ayudaron en la edición, como *flycheck* o *flyspell*.
  - **Paquete Tikz.** Usé este paquete para la realización de todas las figuras de este trabajo.
- **Haskell.** Usando el *Gestor de Paquetes Synaptic*, me descargué los paquetes *haskell-platform* y *GHC*. La versión de *Haskell* con la que he trabajado es la *2014.2.0.0* y la del compilador *GHC*, la *7.10.3-7*. Además para trabajar en *Emacs*, instalé el paquete *haskell-mode* y *doctest*. Este último automatizó la tarea de comprobar la veracidad de todos los ejemplos del código.
- **Git y GitHub.** Para control de versiones he usado *Git*. El editor *Emacs* me facilitó el aprendizaje de este sistema mediante el paquete *Magit*. Además todo el trabajo está subido a un repositorio *Github*[1]. La versión de *Magit*

[1] https://github.com/migpornar/SRTenHaskell

que utilizo es la *2016.12.01*.

Índice alfabético

Bibliografía

[1] Magit User Manual. Technical report.

[2] LaTeX wiki. Technical report, 2015.

[3] OCaml Documentation. Technical report, 2015.

[4] J.A. Alonso. Temas de programación funcional. Technical report, Univ. de Sevilla, 2015.

[5] J. Ataz. Creación de ficheros LATEX con GNU Emacs. Technical report.

[6] J. Ataz. Una introducción rápida a GNU Emacs. Technical report.

[7] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press Professional, 1990.

[8] T. Nipkow F. Baader. *Term Rewriting and All that*. Cambridge University Press, 1998.

[9] J. Fleuriot. Automated Reasoning course. Technical report, University of Edinburgh, 2016.

[10] Free Software Foundation. A Guided Tour of Emacs. Technical report.

[11] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[12] S. Hengel. Doctest Documentation. Technical report, Univ. de Sevilla.

[13] G. Hutton. *Programming in Haskell*. University of Nottingham, 2016.

[14] GitHub Inc. GitHub Help. Technical report.

[15] M. Lipovača. Learn You a Haskell for Great Good! Technical report.

[16] J. Ruiz. *Una teoría computacional acerca de la lógica ecuacional*. PhD thesis.

[17] K. Berry T. Martinsen, S. Gilmore. LATEX2e: An unofficial reference manual. Technical report, 2015.

[18] A. Borbón W. Mora. Insertar gráficos y figuras en documentos LaTeX. Technical report, Instituto Tecnológico de Costa Rica, 2009.

[19] Wikipedia. Rewriting. Technical report, Wikipedia, La Enciclopedia libre, 2016.