Apuntes de Conocimiento y Razonamiento Automatizado Curso 2019-2020

D. Castro, J.E. Morais

Universidad de Alcalá

Índice general

Capítulo 1

Resolución en el Cálculo Proposicional

El objetivo principal del presente Capítulo es la presentación del algoritmo de resolución, introducido por Robinson en la década de los sesenta del siglo pasado. En la sección 1.4 presentaremos el citado algoritmo, pero previamente situaremos el contexto de la Sección 1.1 y trataremos los tableros semánticos en 1.2.

1.1. Definiciones básicas

La parte sintáctica del Cálculo Proposicional describe y estudia las formas en las que distintas afirmaciones se combinan para formar nuevas afirmaciones. Las operaciones que combinan afirmaciones se denominan *conectivos*. Consideraremos los siguientes conectivos: $\neg$ (negación), $\vee$ (disyunción), $\wedge$ (conjunción), $\longrightarrow$ (implicación), $\leftrightarrow$ (bicondicional), $\oplus$ (disyunción excluyente), $|$ (NAND) y $\downarrow$ (NOR). De momento, no nos preocupa cuál es el significado de estos conectivos, aunque sus definiciones coinciden, obviamente, con las dadas en la asignatura de Estructuras Discretas.
La descripción de la sintaxis de cualquier lenguaje comienza con su alfabeto. El alfabeto del lenguaje del Cálculo Proposicional consta de los siguientes símbolos

- Conectivos: $\neg$, $\vee$, $\wedge$, $\longrightarrow$, $\leftrightarrow$, $\oplus$, $|$, $\downarrow$
- Paréntesis: (,)
- Letras proposicionales o átomos: $p, p_1, p_2, \ldots, q, q_1, q_2, \ldots,$

Una vez el alfabeto descrito, definimos lo que se entiende por proposición:

Definición 1.1 (Proposiciones o fórmulas)

i) Los átomos son proposiciones.

ii) Si A y B son proposiciones, entonces $\neg A$ es una proposición y $(A \ op \ B)$ es una proposición si op representa cualquier conectivo binario.

iii) Una cadena de símbolos es un proposición si, y sólo si, puede obtenerse empezando por letras proposicionales y aplicando repetidamente ii).

Definición 1.2 *Sea $\mathcal{P}$ el conjunto de letras proposicionales. Una* **asignación** *es una función $\nu : \mathcal{P} \to \{T, F\}$, esto es, ν asigna uno de los dos valores de verdad a cada letra proposicional.*

Dada una asignación, podemos extenderla a una función que asocie fórmulas del Cálculo Proposicional a valores de verdad, utilizando las definiciones de los significados de cada uno de los conectivos tal cual se presentaron en Estructuras Discretas. Así, por ejemplo, $\nu(p \vee q) = F$ si $\nu(p) = \nu(q) = F$ y $\nu(p \vee q) = T$ en cualquier otro caso. Lo hecho con la disyunción se puede hacer con cualquier otro conectivo.

Definición 1.3 *Dado un conjunto de fórmulas $A = \{A_1, \ldots, A_n\}$, una* **interpretación** *es la extensión de una asignación $\nu : \mathcal{P} \to \{T, F\}$ a A.*

El uso de la misma notación para asignaciones e interpretaciones no es ambiguo por cuanto cada asignación puede extenderse a exactamente una única interpretación.

Definición 1.4 *Dadas fórmulas A y B, son* **equivalentes** *si $\nu(A) = \nu(B)$ para toda interpretación ν. Se denota $A \equiv B$.*

Nota 1.5 *Para evitar posibles confusiones, señalemos que la notación $\equiv$ no se refiere, en ningún caso, a un conectivo aunque obviamente está intimamente relacionado con el conectivo $\leftrightarrow$:*

Teorema 1.6 *$A \equiv B$ si y sólo si $A \leftrightarrow B$ es cierta para toda interpretación.*

1.1.1. Satisfabilidad, validez y consecuencia

Definición 1.7 *Una fórmula A es* **satisfacible** *si $\nu(A) = T$ para alguna interpretación ν. En tal caso, diremos que ν es un modelo para A. Una fórmula A es* **válida**[1] *si $\nu(A) = T$ para toda interpretación ν. Lo denotaremos por $\vDash A$. Una fórmula A es* **falsificable** *si $\nu(A) = F$ para alguna interpretación ν (***no es válida***). Finalmente, diremos que una fórmula A es* **insatisfacible** *si $\nu(A) = F$ para toda interpretación ν (***no es satisfacible***).*

De las definiciones anteriores se deduce directamente el siguiente resultado:

Teorema 1.8 *Una fórmula A es válida si, y sólo si, $\neg A$ es insatisfacible. Una fórmula A es satisfacible si, y sólo si, $\neg A$ es falsificable.*

Definición 1.9 *Se dice que un conjunto de fórmulas $A = \{A_1, \ldots, A_n\}$ es* **(simultáneamente) satisfacible** *si $\nu(A_1) = T, \ldots, \nu(A_n) = T$ para alguna interpretación ν. En ese caso, diremos que ν es un modelo para A.*

Definición 1.10 *Dado un conjunto de fórmulas U, una fórmula A es* **consecuencia** *de U si es cierta para todo modelo de U. Se escribe $U \vDash A$.*

Nota 1.11 *Lo dicho para $\equiv$ y $\leftrightarrow$ es válido para $\vDash$ y $\longrightarrow$.*

Definición 1.12 *Una* **teoría** *es un conjunto cerrado bajo consecuencia lógica. Sus elementos son* **teoremas**. *Dado un conjunto de fórmulas U, el conjunto de sus consecuencias $\mathcal{T}(U)$ es la* **teoría de** *U, U son los* **axiomas** *de $\mathcal{T}(U)$ y $\mathcal{T}(U)$,*

Nota 1.13 *Una teoría es* **axiomatizable** *si admite un conjunto de axionas.*

1.2. Tableros semánticos

La construcción de tablas de verdad es un método poco eficiente para determinar la validez de una fórmula (o conjunto de fórmulas) en el cáculo proposicional. Un método relativamente eficiente es el de los *tableros semánticos*. La idea de este método es la búsqueda sistemática de un modelo para la fórmula. Tomemos el siguiente ejemplo, en el que, por simplicidad, los únicos operadores binarios que aparecen son la conjunción y la disyunción.

Ejemplo 1.14 *Estudiar la satisfabilidad de la fórmula $B = (p \vee q) \wedge (\neg p \wedge \neg q)$.*

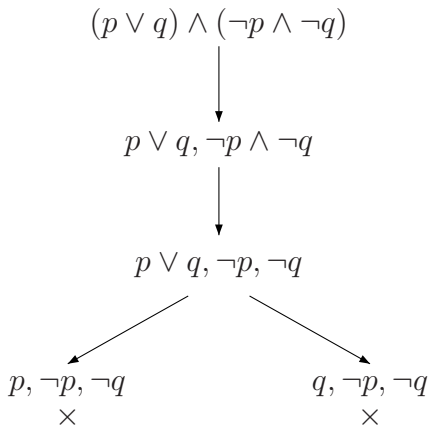
[1] Una fórmula válida también se conoce como tautología.

- $\nu(B) = T$ si, y sólo si, $\nu(p \vee q) = T$ y $\nu(\neg p \wedge \neg q) = T$.
- Por lo tanto, $\nu(B) = T$ si, y sólo si, $\nu(p \vee q) = \nu(\neg p) = \nu(\neg q) = T$.
- En definitiva, $\nu(B) = T$ si, y sólo si, una de las dos siguientes posibilidades se verifica

  1. $\nu(p) = \nu(\neg p) = \nu(\neg q) = T$.
  2. $\nu(q) = \nu(\neg p) = \nu(\neg q) = T$.

Como los conjuntos de literales finales $\{p, \neg p, \neg q\}$ y $\{q, \neg p, \neg q\}$ contienen pares complementarios (es decir, un literal y su negación) la fórmula es claramente insatisfacible.

La búsqueda sistemática de un modelo puede llevarse a cabo si usamos una estructura de datos adecuada. En el caso de los *tableros semánticos* se utilizan árboles: el nodo raíz se etiqueta con la fórmula original y el conjunto de fórmulas creado por la búsqueda etiquetan los nodos interiores. Las hojas se etiquetan con un conjunto de literales y una marca: $\times$ si el conjunto contiene pares complementarios y $\odot$ en el caso contrario. Para el ejemplo anterior, el tablero semántico sería:



Una presentación concisa de las reglas para crear tableros semánticos puede darse si las fórmulas se clasifican tomando en cuenta el operador principal. Si la fórmula es una negación, la clasificación tiene en cuenta tanto la negación como el operador principal. Se distinguen dos tipos de fórmulas: las $\alpha-$fórmulas son conjuntivas y se satisfacen si lo hacen cada una de las subfórmulas α_1 y α_2. Las $\beta-$fórmulas, que son disyuntivas, se satisfacen si una de sus subfórmulas β_1 o β_2 lo hace. Las reglas de construcción de los tableros en función del tipo de fórmula, se recogen en las tablas que siguen

α	α_1	α_2
$\neg\neg A_1$	A_1	
$A_1 \wedge A_2$	A_1	A_2
$\neg(A_1 \vee A_2)$	$\neg A_1$	$\neg A_2$
$\neg(A_1 \to A_2)$	A_1	$\neg A_2$
$\neg(A_1 \mid A_2)$	A_1	A_2
$A_1 \downarrow A_2$	$\neg A_1$	$\neg A_2$
$A_1 \leftrightarrow A_2$	$A_1 \to A_2$	$A_2 \to A_1$
$\neg(A_1 \oplus A_2)$	$A_1 \to A_2$	$A_2 \to A_1$

β	β_1	β_2
$\neg(B_1 \wedge B_2)$	$\neg B_1$	$\neg B_2$
$B_1 \vee B_2$	B_1	B_2
$B_1 \to B_2$	$\neg B_1$	B_2
$B_1 \mid B_2$	$\neg B_1$	$\neg B_2$
$\neg(B_1 \downarrow B_2)$	B_1	B_2
$\neg(B_1 \leftrightarrow B_2)$	$\neg(B_1 \leftarrow B_2)$	$\neg(B_2 \leftarrow B_1)$
$B_1 \oplus B_2$	$\neg(B_1 \leftarrow B_2)$	$\neg(B_2 \leftarrow B_1)$

Ahora, definimos tablero semántico y damos un algoritmo para la construcción del mismo.

Algoritmo 1.15 (Construcción de un tablero semántico)

Input: *Una fórmula A del Cálculo Proposicional.*

Output: *Un tablero semántico $\mathcal{T}$ para A con todas sus hojas marcadas.*

Un tablero semántico $\mathcal{T}$ para A es un árbol en el que cada nodo está etiquetado con un conjunto de fórmulas. El nodo raíz está etiquetado con A y, dado un nodo cualquiera l, etiquetada con un conjunto de fórmulas $U(l)$, aplicamos alguna de las siguientes reglas:

- *Si $U(l)$ es un conjunto de literales, comprobar si hay dos complementarios. Si es el caso, la rama se marca* **cerrada** $\times$*; en caso contrario, se marca* **abierta** $\odot$.
- *Si $U(l)$ no es un conjunto de literales, escogemos una fórmula en $U(l)$ que no sea un literal y se aplica alguna de las reglas siguientes:*
  - *Si es una α–fórmula, se crea un nuevo nodo hijo l', y se etiqueta con*

$$U(l') := (U(l) \setminus \{\alpha\}) \cup \{\alpha_1, \alpha_2\}$$

de acuerdo a las tablas anteriores.

- *Si es una β–fórmula, se crean dos nuevos nodos hijos l' y l'', y se etiquetan*

$$U(l') := (U(l) \setminus \{\beta\}) \cup \beta_1, \text{ y}$$
$$U(l'') := (U(l) \setminus \{\beta\}) \cup \beta_2,$$

de nuevo, de acuerdo a las tablas citadas.

Definición 1.16 *Un tablero cuya construcción ha terminado se dice completo. Un tablero completo se dice cerrado si todas las hojas están marcadas con $\times$ y abierto en caso contrario.*

Teorema 1.17 *La construcción de un tablero semántico termina.*

Robustez y completitud de los tableros semánticos

La construcción de tableros semánticos puede utilizarse para determinar la validez de una fórmula. Para ello, habrá que construir un tablero semántico para la negación de la fórmula. Se tiene:

Teorema 1.18 (Robustez y completitud) *Supongamos que $\mathcal{T}$ es un tablero completo para la fórmula A. Entonces, A es insatisfacible si, y sólo, si $\mathcal{T}$ es cerrado.*

Corolario 1.19 *A es satisfacible si, y sólo si, $\mathcal{T}$ es abierto.*

Corolario 1.20 *A es válida si, y sólo si, un tablero para $\neg A$ es cerrado.*

Nota 1.21 *La implicación directa de 1.20 establece la completitud de la construcción de tableros semánticos (dicho de otra forma, si la respuesta a la pregunta sobre la validez de una fórmula es negativa, tenemos la seguridad absoluta de que es así). La otra implicación establece la robustez de los tableros semánticos (en otras, palabras si la respuesta a la pregunta sobre la validez es positiva, se tiene la seguridad de que es así). En cualquier sistema lógico que pretenda establecer la validez de fórmulas es siempre más sencillo probar la robustez (basta ver que las reglas introducidas son correctas) que la completitud. Es difícil saber si hemos olvidado alguna regla que pueda ser necesaria para la completitud.*

Nota 1.22 *El corolario 1.20 nos dice cómo saber si una fórmula es válida o no. Basta construir un tablero para la negación de la fórmula. Lo que no dice es que si un tablero para una fórmula es abierto, esta sea válida. Piénsese en un tablero semántico para $p \vee q$, por ejemplo.*

Ejemplo 1.23 *1. Construir un tablero semántico para la fórmula*

$$(p \wedge q) \longrightarrow (\neg((p \longrightarrow q)) \vee r).$$

2. Estudiar la validez de la fórmula

$$((p \to q) \to r) \to (q \to r)$$

mediante el uso de tableros semánticos.

3. Probar la validez de la fórmula

$$(\neg p \wedge q \wedge (\neg p \to (q \to r)) \wedge ((q \wedge r) \to s)) \to s$$

mediante el uso de tableros semánticos.

1.3. Formas normales conjuntivas

Definamos, formalmente, el concepto de forma normal conjuntiva.

Definición 1.24 *Una fórmula se dice en forma normal conjuntiva si es una conjunción de disyunciones de literales.*

Ejemplo 1.25 *La fórmula* $(\neg p \vee q \vee r) \wedge (\neg q \vee r)$ *está en forma normal conjuntiva.*

Teorema 1.26 *Toda fórmula del Cálculo Proposicional puede transformarse en una equivalente en forma normal conjuntiva.*

Demostración .– Para demostrar el Teorema, llevamos a cabo las siguientes transformaciones en una fórmula cualquiera:

1. Eliminamos todos los operadores, salvo las conjunciones, disyunciones y negaciones, empleando las siguientes equivalencias:

$A \to B$	$\neg A \vee B$
$A \leftrightarrow B$	$(\neg A \vee B) \wedge (A \vee \neg B)$
$A \oplus B$	$(A \wedge \neg B) \vee (\neg A \wedge B)$
$A \mid B$	$\neg(A \wedge B)$
$A \downarrow B$	$\neg(A \vee B)$

2. Llevamos las negaciones junto a los átomos empleando las leyes de Morgan:

$\neg(A \wedge B)$	$\neg A \vee \neg B$
$\neg(A \vee B)$	$\neg A \wedge \neg B$

3. Eliminamos las dobles negaciones empleando $\neg\neg A \equiv A$.

4. Aplicamos las leyes distributivas:

$A \vee (B \wedge C)$	$(A \vee B) \wedge (A \vee C)$
$(A \wedge B) \vee C$	$(A \vee C) \wedge (A \vee C)$

■

Ejemplo 1.27 *Reducir a FNC las siguientes fórmulas:*

- $((\neg p \longrightarrow q) \wedge (q \longrightarrow r)) \wedge ((\neg q \longrightarrow \neg r) \longrightarrow p)$
- $F = ((p \oplus q) \longrightarrow (r \vee s)) \longrightarrow (r|s)$
- $(p \to q) \Leftrightarrow (p \to (r \vee q))$.

Definición 1.28 (Cláusula, cláusula unitaria, forma clausal) *Se dice cláusula a todo conjunto de literales (átomos o sus negaciones) que se suponen, implícitamente, en disyunción. Una cláusula es unitaria si contiene un único literal. Finalmente, se dice forma clausal a todo conjunto de cláusulas que se suponen, implícitamente, en conjunción.*

Corolario 1.29 *Toda fórmula del Cálculo Proposicional admite una forma clausal equivalente.*

Demostración .– El resultado se sigue de manera inmediata del Teorema precedente y de la definición de forma clausal. ∎

Ejemplo 1.30 *El conjunto de literales $\{p, \neg q, r\}$ es una cláusula que representa la fórmula*

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

El conjunto de cláusulas $\{\{p, q, \neg r\}, \{s, t, r\}\}$ representa a la fórmula

$$(p \vee q \vee (\neg r)) \wedge (s \vee t \vee r).$$

Normalmente, y con el fin de aligerar la notación, la forma clausal anterior suele escribirse $\{pq\bar{r}, str\}$.

Definición 1.31 (Cláusulas equivalentes) *Dadas dos cláusulas S y S', notaremos $S \approx S'$ siempre que S es satisfacible si y sólo si S' es satisfacible.*

Lema 1.32 *Supongamos que un literal l aparece en una cláusula de S, pero no l^c. Sea S' la forma clausal obtenida a partir de S eliminando todas las cláusulas en las que aparece l. Entonces, $S \approx S'$.*

Demostración .– Supongamos que S' es satisfacible, i.e. existe un modelo tal que $\nu(C') = T$ para toda $C' \in S'$. Extendiendo ν mediante $\nu(l) = T$, se sigue que $\nu(C) = T$ para toda $C \in S$.
Recíprocamente, si S es satisfacible, se sigue trivialmente que S' es satisfacible (basta observar que $S' \subseteq S$). ∎

Ejemplo 1.33 *La forma clausal* $\{pqr, \bar{p}q, r\}$ *representa la fórmula*

$$(p \vee q \vee r) \wedge (\neg p \vee q) \wedge r.$$

Dicha fórmula es satisfacible (basta tomar $\nu(r) = T$ *y* $\nu(q) = T$*). En este caso, podríamos tomar como forma clausal* S' $\{r\}$*, que es trivialmente satisfacible.*
Un modelo para S' *es, necesariamente,* $\nu(r) = T$ *que se extiende a un modelo de* S *añadiendo* $\nu(q) = T$ *(*$\nu(p)$ *puede tomar cualquier valor).*

Lema 1.34 *Sea* $C = \{l\} \in S$ *una cláusula unitaria y sea* S' *la forma clausal obtenida de* S *eliminando las cláusulas que contienen a* l *y eliminando* l^c *de las restantes. Entonces,* $S \approx S'$*.*

Demostración .– Sea ν un modelo para S. En particular, se tiene que $\nu(l) = T$. Sea ahora $C \in S$ una cláusula tal que $l^c \in C$ y sea $C' = C \setminus \{l^c\}$. Entonces $\nu(C') = T$ puesto que $\nu(C) = T$ y $\nu(l^c) = F$.
Recíprocamente, supongamos ahora que ν es un modelo para S'. Extendiendo ν mediante $\nu(l) = T$ obtenemos un modelo para S. ∎

Ejemplo 1.35 *Consideramos la forma clausal* $\{\bar{r}, pqr, pq\bar{r}\}$*. La fórmula asociada es satisfacible sin más que definir* $\nu(r) = F$*,* $\nu(p) = T$ *y* $\nu(q) = F$*. La forma clausal* $S' = \{pq\}$ *es satisfacible para igual valoración* ν*.*
Si partimos de $S' = \{pq\}$ *y un modelo para la misma, basta definir* $\nu(r) = F$ *para obtener un modelo para* S*.*

Definición 1.36 (Subsumir) *Dadas un par de cláusulas* C_1 *y* C_2*, diremos que* C_1 *subsume a* C_2 *si* $C_1 \subseteq C_2$*. Equivalentemente, diremos que* C_2 *está subsumida por* C_1*.*

Lema 1.37 *Sean* $C_1, C_2 \in S$ *dos cláusulas de* S *tales que* C_1 *subsume a* C_2*. Definiendo* $S' = S \setminus \{C_2\}$ *se tiene que* $S \approx S'$*.*

Demostración .– Cualquier asignación que haga cierta a C_1 hace cierta a C_2, por lo tanto $S \approx S'$. ∎

Ejemplo 1.38 *La clásula* $\{pq\}$ *subsume a* $\{pqr\}$*. Por lo tanto, si* $\{pqr, pq, s\}$ *es satisfacible, también lo es* $\{pq, s\}$ *(y recíprocamente).*

Lema 1.39 *Sea* C *un cláusula de* S *tal que* $l, l^c \in C$*. Definiendo* $S' = S \setminus \{C\}$ *se tiene* $S \equiv S'$*.*

Notación 1.40 *En lo que sigue, notaremos por* $\square$ *la cláusula vacía y* $\emptyset$ *el conjunto de cláusulas vacío.*

Lema 1.41 $\emptyset$ *es válido y* $\square$ *es insatisfacible.*

1.4. Resolución

Regla de Resolución: Sean C_1 y C_2 dos cláusulas de S tales que $l \in C_1$ y $l^c \in C_2$ [2]. Definimos la resolvente de C_1 y C_2 como la cláusula

$$Res(C_1, C_2) = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{l^c\}).$$

Teorema 1.42 *La resolvente C de dos cláusulas generatrices, digamos C_1 y C_2, es satisfacible si, y sólo si, ambas cláusulas son simultáneamente satisfacibles.*

Demostración .– Supongamos que C_1 y C_2 son dos cláusulas generatrices simultáneamente satisfacibles para una cierta asignación ν. Distinguimos los siguientes casos:

- $\nu(l) = T$: se sigue que $\nu(l^c) = F$. Por lo tanto, existe un literal en C_2, digamos $l' \neq l^c$, tal que $\nu(l') = T$. Puesto que $l' \in C$, se sigue que $\nu(C) = T$.
- $\nu(l^c) = T$: tenemos que $\nu(l) = F$. Por lo tanto, existe un literal $l'' \in C_1$, con $l'' \neq l$, tal que $\nu(l'') = T$. Puesto que $l'' \in C$, tenemos que $\nu(C) = T$.

Recíprocamente, supongamos que $\nu(C) = T$. Obviamente, $\nu(l''') = T$ para un cierto $l''' \in C$. Distinguimos los siguientes casos:

- Si $l''' \in C_1$, se tiene que definiendo $\nu(l) = F$, C_1 y C_2 son mutuamente satisfactorias.
- Si $l''' \in C_2$, extendemos ν mediante $\nu(l) = T$, con lo que se sigue igual conclusión.

■

Algoritmo 1.43 (Resolución) *El algoritmo de resolución toma como entrada un conjunto de cláusulas S y devuelve como información la satisfabilidad o insatisfabilidad de S.*
El algoritmo funciona como sigue: de entrada, define $S_0 = S$. Supuesto construido S_i, construye $S_{i+1} := S_i \cup \{Res(C_1, C_2)\}$ donde C_1 y C_2 son dos cláusulas generatrices (si estas existen). El algoritmo termina cuando no existen tales cláusulas generatrices o se deriva la cláusula vacía.

[2] A partir de ahora dos cláusulas así se dirán generatrices.

Definición 1.44 (Refutación) *Llamaremos refutación a toda derivación de* $\square$.

Ejemplo 1.45 *Consideremos el conjunto de cláusulas*

$$S = \{p, \bar{p}q, \bar{r}, \bar{p}\bar{q}r\}.$$

Enumeraremos las cláusulas como sigue:

$$C_1 = p,$$

$$C_2 = \bar{p}q,$$

$$C_3 = \bar{r},$$

$$C_4 = \bar{p}\bar{q}r.$$

Construimos las siguientes resolventes

$$C_5 = Res(C_3, C_4) = \bar{p}\bar{q},$$

$$C_6 = Res(C_5, C_2) = \bar{p}, \ y$$

$$C_7 = Res(C_6, C_1) = \square.$$

Puesto que hemos derivado la cláusula vacía, S es insatisfacible.

Ejemplo 1.46 Estudiar si de las premisas $F_1 = p \longrightarrow (q \longrightarrow r)$, $F_2 = (r \wedge s) \longrightarrow t$ y $F_3 = \neg w \longrightarrow (s \wedge \neg t)$ se puede concluir $F = p \longrightarrow (q \longrightarrow w)$. *La pregunta es equivalente a decidir si la fórmula*

$$(F_1 \wedge F_2 \wedge F_3) \longrightarrow F$$

es válida. Dicho de otro modo, decidir si la fórmula

$$F_1 \wedge F_2 \wedge F_3 \wedge \neg F$$

es insatisfacible. Reduciendo la fórmula inmediatamente anterior a FNC, esto es lo mismo que decidir si podemos derivar $\square$ *del siguiente conjunto de cláusulas:*

$$\{\bar{p}\bar{q}r, \bar{r}\bar{s}t, ws, w\bar{t}, p, q, \bar{w}\}.$$

Es fácil ver que esto es posible, luego la respuesta a la pregunta inicial es afirmativa.

Teorema 1.47 (Robustez) *Si, para un conjunto de cláusulas S, el procedimiento de resolución deriva* $\square$*, se sigue que S es insatisfacible.*

Demostración .– Por reducción al absurdo, supongamos que S es satisfacible y que el procedimiento de resolución deriva $\square$ en n pasos, esto es, el algoritmo genera una sucesión creciente de formas cláusales $S_0 \subset S_1 \subset \cdots \subset S_n$, con $S_0 = S$, tal que $\square \in S_n$ y S_i difiere de S_{i+1} en una resolvente. Obviamente podemos suponer que $\square \notin S_{n-1}$.
Sean $C_1, C_2 \in S_{n-1}$ dos cláusulas generatrices tales que $Res(C_1, C_2) = \square$. Puesto que S es satisfacible se sigue que C_1 y C_2 son simultáneamente satisfacibles. Por lo tanto, de acuerdo al Teorema 1.42, se tiene que el conjunto S_n sigue siendo satisfacible. Ahora bien, puesto que la cláusula $\square$ pertenece a S_n, llegamos a una contradicción. ∎

Definición 1.48 (Árbol semántico) *Sea S un conjunto de cláusulas cuyos átomos son $p_1, \ldots, p_n$. Un árbol semántico es un árbol en el que cada nodo de profundidad $i-1$ (con $i \geq 1$) tiene, exactamente, dos hijos cuyas correspondientes ramas están etiquetadas con p_i (izquierda) y $\bar{p}_i$ (derecha).*

Obviamente, cada rama de un árbol semántico determina una asignación definida de la siguiente manera: el valor de un átomo viene dado por la rama que se sigue partiendo de su nodo asociado, verdadero si la rama sigue por la izquierda y falso si sigue por la derecha.

Definición 1.49 (Rama cerrada, árbol cerrado) *Dada una forma clausal S y un árbol semántico para la misma, diremos que una rama b del árbol está cerrada si $\nu_b(S) = F$, donde ν_b es la asignación asociada a la rama b. En caso contrario, diremos que la rama está abierta. Si todas las ramas del árbol están cerradas, diremos que el árbol es cerrado.*

Teorema 1.50 *Un árbol semántico A para un conjunto de fórmulas S es cerrado si y sólo si S es insatisfacible.*

Demostración .– La demostración se sigue de las definiciones. ∎

Definición 1.51 (Nodo de fallo) *Sea S un conjunto de cláusulas y sea A un árbol semántico para S. Llamaremos nodo de fallo a todo nodo en una rama cerrada cuya asignación parcial falsifique una cierta cláusula de S y que, en la citada rama, sea el más cercano a la raíz de entre todos los que falsifican alguna cláusula de S.*

Definición 1.52 (Nodo de inferencia) *Sea S un conjunto de cláusulas y sea A un árbol semántico para S. Llamaremos nodo de inferencia a todo nodo de A cuyos hijos sean nodos de fallo.*

Teorema 1.53 (Completitud) *Si un conjunto de cláusulas S es insatisfacible, el procedimiento de resolución deriva $\square$.*

Demostración .– Puesto que el conjunto de cláusulas S es insatisfacible, cualquier árbol semántico para S es cerrado, i.e. cualquier rama es cerrada y, por lo tanto, tiene un nodo de fallo.
Veamos que, necesariamente, tiene que haber un nodo de inferencia. Supongamos, por reducción al absurdo, que no es así y sea n_0 un nodo de fallo. El nodo hermano de n_0, digamos n', no puede ser un nodo de fallo (de serlo, existiría un nodo de inferencia). Además, la rama que continua por n' debe tener un nodo de fallo con una profundidad mayor que la de n, digamos n_1. Por lo tanto podemos construir una secuencia de nodos $n_0, n_1, \ldots$ de profundidad creciente, lo cual es absurdo. Así pues, se sigue que todo árbol cerrado tiene un nodo de inferencia.
Los hijos de un nodo inferencia falsifican cláusulas. El hijo de la izquierda se corresponde con valor T para un átomo, por lo que la cláusula que falsifica, digamos C_1, ha de contener la negación de dicho átomo. Del mismo modo, el hijo de la derecha falsifica una cláusula, que denotaremos por C_2, que ha de contener al citado átomo. Así pues, al existir un nodo de inferencia existen dos cláusulas generatrices, C_1 y $C_2 \in S$. La resolvente $Res(C_1, C_2)$ se corresponde con un nuevo nodo de fallo de profundidad menor que substituye a C_1 y C_2. Por lo tanto, con cada resolvente que añadimos, el número de nodos de fallo decrece, como poco, en uno. Además, mientras el algoritmo de resolución no haya derivado la cláusula $\square$, podemos calcular al menos una resolvente. Se sigue que, tarde o temprano, el algoritmo derivará $\square$. ■

Capítulo 2

Resolución en el Cálculo de Predicados

En este segundo Capítulo realizaremos algo similar a lo hecho en el primero, pero dentro del Cálculo de Predicados: situaremos el contexto, presentaremos los tableros semánticos y, finalmente, la resolución.

2.1. Tableros semánticos para el Cálculo de Predicados

2.1.1. Definiciones básicas

En primer lugar, al igual que pasaba en el Cálculo Proposicional, para situarnos en el contexto del Cálculo de Predicados, necesitamos fijar la sintaxis del mismo. Es decir, debemos fijar el alfabeto y las reglas que, a partir del mismo, permiten construir las expresiones sintácticamente correctas (términos y fórmulas). El alfabeto está formado por

a) El conjunto de símbolos de variable $\mathcal{V} = \{X, Y, \ldots\}$.

b) El conjunto de símbolos de constante $\mathcal{C} = \{a, b, \ldots\}$.

c) El conjunto de símbolos de función de aridad n, $\mathcal{F} = \{f, g, h, \ldots\}$.

d) El conjunto de símbolos de predicado (o conjunto de símbolos de relación de aridad n), $\mathcal{P} = \{p, q, r, \ldots\}$.

e) Los conectivos $\{\neg, \vee, \wedge, \longrightarrow, \leftrightarrow, \oplus, |, \downarrow\}$.

f) Los cuantificadores $\forall$ y $\exists$.

g) Los paréntesis (y).

Nota 2.1 *Habitualmente, los conjuntos de variables y de símbolos de constante, función y predicado se considera que deben ser conjuntos numerables. Por otro lado, cada símbolo de función (o de predicado) tiene asociado un número natural n que se corresponde con su aridad. Podemos pensar, de este modo, los símbolos de constante como símbolos de función de aridad cero. En general, la aridad se deduce del contexto y no se suele explicitar, pero si se necesita escribiremos* f^n *o* p^n*.*

Los términos y fórmulas son las cadenas de símbolos que, formados siguiendo las reglas inductivas que damos en las dos definiciones siguientes, constituyen construcciones bien formadas del lenguaje del Cálculo de Predicados. Hablaremos de expresiones para englobar tanto a términos como a fórmulas.

Definición 2.2 *Se dicen* **términos** *a las cadenas que puede formarse aplicando inductivamente las siguientes reglas:*

i) Las variables y las constantes son términos.

ii) Si f *es una función* $n-$*aria y* $t_1, t_2, \ldots, t_n$ *son términos,* $f(t_1, \ldots, t_n)$ *es un término.*

El conjunto de términos lo denotaremos por $\mathcal{T}$*. Por otro lado, un término sin variables se dice* **básico**.

Definición 2.3 *Se dicen* **fórmulas** *a las cadenas que pueden formarse aplicando inductivamente las siguientes reglas:*

i) Toda **fórmula atómica** *o* **átomo**, *es decir,* $p(t_1, \ldots, t_n)$ *con* p *un símbolo de relación* $n-$*ario y* $t_1, \ldots, t_n$ *términos, es una fórmula.*

ii) Si A *y* B *son fórmulas, también lo son* $(\neg A)$ *y* $(A \text{ op } B)$*, donde op es cualquiera de los conectivos binarios*[1]*.*

iii) Si A *es una fórmula y* $X \in \mathcal{V}$*,* $(\forall X)A$ *y* $(\exists X)A$ *son fórmulas.*

Definición 2.4 *Se llama ocurrencia de una variable a cualquier aparición de una variable en una fórmula. En una fórmula del tipo* $(\forall X)A$ *o* $(\exists X)A$ *se dice que* A *es el radio de acción de la misma. Una ocurrencia de una variable se dice que es ligada si aparece dentro del radio de acción de un cuantificador universal o uno existencial. Una ocurrencia de una variable se dice libre si no es ligada. Una fórmula cerrada es aquella en la que no hay ocurrencias de variables libres.*

[1] Un átomo o la negación de un átomo suelen recibir el nombre de literal.

Clarificada la sintaxis del Cálculo de Predicados, nos ocupamos de la semántica del mismo. Si no atendemos a la semántica de las fórmulas, sería imposible asignar a las mismas ningún valor de verdad.

Definición 2.5 (Interpretación) *Supongamos un conjunto de fórmulas U. Sean $\{p_1, \ldots, p_j\}$ el conjunto de símbolos de predicado, $\{f_1, \ldots, f_k\}$ el conjunto de símbolos de función y $\{a_1, \ldots, a_l\}$ las constantes de U. Llamaremos interpretación de U a toda tupla*

$$I = (D, \{R_1, \ldots, R_j\}, \{F_1, \ldots, F_k\}, \{d_1, \ldots, d_l\}),$$

donde: D es un dominio no vacío, cada símbolo de predicado p_i, de aridad n_i, está asociado con una relación R_i, de aridad n_i; cada símbolo de función f_i, de aridad n_i, está asociado con una función F_i definida sobre D, de aridad n_i y, finalmente, cada constante a_i está asociada con una constante del dominio d_i.

Ejemplo 2.6 *Consideremos la fórmula $(\forall X)\ p(a, X)$. Las siguientes son distintas interpretaciones numéricas:*

$$\mathcal{I}_1 = (\mathbb{N}, \{\leq\}, \{0\}),\ \ \mathcal{I}_2 = (\mathbb{N}, \{\leq\}, \{1\}),\ \ \mathcal{I}_3 = (\mathbb{Z}, \{\leq\}, \{0\}).$$

Una interpretación no numérica para la fórmula es la siguiente: si $\mathcal{S}$ denota el conjunto de cadenas binarias, $\mathcal{I}_4 = (\mathcal{S}, subcadena, \lambda)$, donde subcadena es la relación binaria "ser subcadena de" y λ representa la cadena vacía.

Definición 2.7 (Asignación) *Sea $\mathcal{I}$ una interpretación. Una asignación $\sigma_{\mathcal{I}} : \mathcal{V} \longrightarrow \mathcal{D}$ es una función que asocia a cada variable de la fórmula un elemento del dominio de $\mathcal{I}$. Para explicitar que las variables X_i se corresponden con elementos del dominio d_i, escribiremos $\nu_{\sigma_{\mathcal{I}}[X_i \leftarrow d_i]}$.*

Dadas una interpretación y una asignación, podemos dar a una fórmula un valor de verdad

Definición 2.8 *Sea A una fórmula, $\mathcal{I}$ una interpretación y $\sigma_{\mathcal{I}}$ una asignación. El valor de la fórmula para $\sigma_{\mathcal{I}}$, denotado por $\nu_{\sigma_{\mathcal{I}}}(A)$ se define por:*

- *Si $A = p_k(t_1, \ldots, t_n)$ es un átomo, $\nu_{\sigma_{\mathcal{I}}}(A) = T$, si $(t_1, \ldots, t_n) \in R_k$, si R_k es la relación asociada por $\mathcal{I}$ a p_k. Naturalmente, para evaluar los términos, los símbolos de constante se sustituyen por las correspondientes constantes según $\mathcal{I}$, los símbolos de función por las correspondientes funciones y las variables por las imágenes correspondientes según $\sigma_{\mathcal{I}}$.*

- $\nu_{\sigma_\mathcal{I}}(\neg A) = T$ *si y sólo si* $\nu_{\sigma_\mathcal{I}}(A) = F$
- $\nu_{\sigma_\mathcal{I}}(A \; op \; B) = T$ *si y sólo si* $\nu_{\sigma_\mathcal{I}}(A) \; op \; \nu_{\sigma_\mathcal{I}}(B) = T$.
- $\nu_{\sigma_\mathcal{I}}(\forall X A) = T$ *si y sólo si* $\nu_{\sigma_{\mathcal{I}[X \leftarrow d]}}(A) = T$ *para todo* $d \in D$.
- $\nu_{\sigma_\mathcal{I}}(\exists X A) = T$ *si y sólo si* $\nu_{\sigma_{\mathcal{I}[X \leftarrow d]}}(A) = T$ *para algún* $d \in D$.

Es trivial que, con las definiciones anteriores, se tenga el resultado que sigue.

Teorema 2.9 *Si A es una fórmula cerrada, su valor no depende de la asignación.*

Por lo tanto, podemos hablar del valor de una fórmula cerrada para una interpretación dada. Este valor lo denotaremos por $\nu_\mathcal{I}(A)$. Además, se tiene:

Teorema 2.10 *Si $A' = A(X_1, \ldots, X_n)$ es una fórmula no cerrada en las que las variables $X_1, \ldots, X_n$ tienen ocurrencias libres, se tiene:*

- $\nu_{\sigma_\mathcal{I}}(A') = T$ *para alguna asignación* $\sigma_\mathcal{I}$ *si y sólo si* $\nu_{\sigma_\mathcal{I}}(\exists X_1 \cdots \exists X_n F) = T$.
- $\nu_{\sigma_\mathcal{I}}(A') = T$ *para toda asignación* $\sigma_\mathcal{I}$ *si y sólo si* $\nu_{\sigma_\mathcal{I}}(\forall X_1 \cdots \forall X_n F) = T$

Definición 2.11 *Se dice que una fórmula cerrada A es verdadera para la interpretación $\mathcal{I}$ o que $\mathcal{I}$ es un* **modelo** *para A si $\nu_\mathcal{I}(A) = T$. Se escribe $\mathcal{I} \vDash A$.*

Ejemplo 2.12 *Las interpretaciones $\mathcal{I}_1$ y $\mathcal{I}_4$ del Ejemplo 2.6 son modelos para la fórmula. Las otras dos, no.*

Definición 2.13 *Una fórmula cerrada A se dice* **satisfacibe** *si para alguna interpretación $\mathcal{I}$, $\mathcal{I} \vDash A$. Si toda interpretación es un modelo para A se dice que la fórmula es* **válida** *y se nota $\vDash A$. La fórmula A se dice* **insatisfacible** *si no es satisfacible y se dice* **falsificable** *si no es válida.*

Ejemplo 2.14 *Veamos algunos ejemplos de fórmulas cerradas y estudiémoslas desde el punto de vista de la satisfabilidad:*

- *La fórmula $(\forall X p(X)) \longrightarrow p(a)$ es válida.*
- *La fórmula $\forall X \forall Y (p(X, Y) \longrightarrow p(Y, X))$ es satisfacible, pero no válida.*
- *La fórmula $\exists X \exists Y (p(X) \wedge \neg p(Y))$ es satisfacible, pero no válida. Obsérvese que cualquier modelo para la fórmula ha de contener, necesariamente, más de un elemento.*

- *La fórmula satisfacible $\forall X p(a, X)$ postula la existencia de algún elemento especial en el dominio de cualquier modelo. Por ejemplo, el cero para los naturales.*

Definición 2.15 *Dos fórmulas cerradas A_1 y A_2, se dicen* **lógicamente equivalentes** *si $\nu_{\mathcal{I}}(A_1) = \nu_{\mathcal{I}}(A_2)$ para cualquier interpretación $\mathcal{I}$. Se escribe $A \equiv B$*

Ejemplo 2.16 *Algunas equivalencias de interés son las siguientes:*

$$\begin{array}{rcl} \neg(\exists p(X)) & \equiv & \forall \neg p(X) \\ \neg(\forall p(X)) & \equiv & \exists \neg p(X) \\ \exists X(p(X) \vee q(X)) & \equiv & ((\exists X p(X)) \vee (\exists q(X))) \\ \forall X(p(X) \wedge q(X)) & \equiv & ((\forall X p(X)) \wedge (\forall q(X))) \end{array}$$

Las dos últimas equivalencias expresan que el cuantificador existencial conmuta con la disyunción y el cuantificador universal con las conjunción. No es cierto, sin embargo, que la conjunción conmute con el cuantificador existencial y que la disyunción conmute con el cuantificador universal. A este respecto, son válidas las fórmulas que siguen:

$$\begin{array}{rcl} ((\forall X p(X)) \vee (\forall X q(X))) & \longrightarrow & (\forall X(p(X) \vee q(X))) \\ (\exists X(p(X) \wedge q(X))) & \longrightarrow & ((\exists X p(X)) \wedge (\forall X q(X))) \end{array}$$

2.1.2. Construcción de tableros semánticos

Recordemos que en el Cálculo Proposicional, la construcción de tableros semánticos está basada en la búsqueda de modelos para la correspondiente fórmula. La misma idea puede aplicarse en el caso del Cálculo de Predicados. Sin embargo, hay una diferencia sustancial con respecto al Cálculo Proposicional: la validez en el Cálculo de Predicados es indecidible.
En lo que sigue, consideramos tableros semánticos para fórmulas en las que los términos que aparecen son únicamente constantes y variables.
Junto a las $\alpha-$reglas y $\beta-$reglas definidas para la construcción de tableros en el caso del Cálculo Proposicional, debemos añadir reglas que nos permitan lidiar con los cuantificadores. Estas reglas se conocen como $\gamma-$reglas y $\delta-$reglas:

γ	$\gamma(a)$
$\forall X A(X)$	$A(a)$
$\neg\exists X A(X)$	$\neg A(a)$

δ	$\delta(a)$
$\exists X A(X)$	$A(a)$
$\neg\forall X A(X)$	$\neg A(a)$

Antes de dar el algoritmo de construcción de tableros semánticos, veremos varios ejemplos que nos ayudarán a entender las dificultades de extender las ideas usadas en el caso del Cálculo Proposicional. Asimismo, trataremos de analizar qué hacer para evitarlas.

Ejemplo 2.17 *El primer ejemplo que consideramos es la negación de la fórmula válida*

$$\forall X(p(X) \longrightarrow q(X)) \longrightarrow (\forall X p(X) \longrightarrow \forall X q(X)).$$

Si aplicamos dos veces $\alpha-$reglas de la forma $\neg(A \longrightarrow B)$, se obtiene el siguiente conjunto de fórmulas:

$$\forall X(p(X) \longrightarrow q(X)), \forall X p(X), \neg\forall X q(X).$$

La fórmula $\neg\forall X q(X)$ es equivalente a $\exists X \neg q(X)$. Por lo tanto, en cualquier modelo, algún elemento del dominio no debe verificar q. En vez de escoger un dominio específico, tomamos una letra genérica del conjunto de constantes $a \in \mathcal{A}$ y especificamos la fórmula. Así, tenemos un nuevo nodo en el tablero:

$$\forall X(p(X) \longrightarrow q(X)), \forall X p(X), \neg q(a).$$

Las dos primeras fórmulas están cuantificadas universalmente. Así pues, podemos instanciarlas en cualquier elemento. En particular, en a.
De esta forma, llegamos a

$$p(a) \longrightarrow q(a), p(a), \neg q(a).$$

Aplicando una $\beta-$ regla, llegamos a un tablero cerrado. Habríamos probado que la negación de la fórmula es insatisfacible y que, por lo tanto, la fórmula es válida.

Ejemplo 2.18 *Ahora consideremos la negación de la siguiente fórmula que es satisfacible pero no válida:*

$$\forall X(p(X) \vee q(X)) \longrightarrow (\forall X p(X) \vee \forall X q(X)).$$

Construyamos un tablero de la misma forma que antes:

$$\begin{array}{c}
\neg(\forall X(p(X) \vee q(X)) \longrightarrow (\forall X p(X) \vee \forall X q(X))) \\
\downarrow \\
\forall X(p(X) \vee q(X)), \neg(\forall X p(X) \vee \forall X q(X)) \\
\downarrow \\
\forall X(p(X) \vee q(X)), \neg\forall X p(X), \neg\forall X q(X) \\
\downarrow \\
\forall X(p(X) \vee q(X)), \neg\forall X p(X), \neg q(a) \\
\downarrow \\
\forall X(p(X) \vee q(X)), \neg p(a), \neg q(a) \\
\downarrow \\
p(a) \vee q(a), \neg p(a), \neg q(a)
\end{array}$$

Aplicando una $\beta-$ regla se llega a un tablero cerrado. De este modo, habríamos probado que la fórmula inicial es válida. ¿Qué es lo que hemos hecho mal? En primer lugar, no deberíamos haber utilizado la misma letra para instanciar $\neg\forall X p(X)$ y $\neg\forall X q(X)$. Haciéndolo, asumimos que en un cierto dominio hay un mismo elemento que no verifica las relaciones denotadas por p y q simultáneamente. Así, debemos introducir un nuevo símbolo de constante para instanciar $\neg\forall X p(X)$ una vez usada la constante a para $\neg\forall X q(X)$. De este modo la quinta línea en el tablero sería:

$$\forall X(p(X) \vee q(X)), \neg p(b), \neg q(a).$$

Por lo tanto, cada vez que instanciemos una fórmula existencial debemos usar un símbolo de constante distinto a los ya utilizados, lo que sugiere la necesidad de tomar nota de los símbolos de constante utilizados. Si ahora instanciamos la fórmula cuantificada universalmente, se llega a

$$p(a) \vee q(a), \neg p(b), \neg q(a).$$

Ahora podríamos instanciar la fórmula $\forall X(p(X) \vee q(X))$ en b por estar cuantificada universalmente, pero ya la hemos utilizado y "borrado" del tablero. Para prevenir esta circunstancia, ninguna fórmula universal se borra del tablero. Siguiendo este criterio, la sexta línea del tablero quedaría

$$\forall X(p(X) \vee q(X)), p(a) \vee q(a), \neg p(b), \neg q(a).$$

Ahora podemos instanciar la fórmula universal en b y se llega a

$$\forall X(p(X) \vee q(X)), p(a) \vee q(a), p(b) \vee q(b), \neg p(b), \neg q(a).$$

Es fácil ver que, aplicando $\beta-$reglas se llega a una única rama abierta en el tablero que se corresponde con el modelo $(\{a, b\}, \{P, Q\})$ con $a \in P$, $a \notin Q$,

$b \notin P$ y $b \in Q$. Jamás hubiésemos llegado a este modelo sin instanciar la fórmula universal en b. Por lo tanto, debemos instanciar todas las fórmulas universales en todos los símbolos de constante que se hayan ido añadiendo en la construcción del tablero.

Con las enseñanzas extraídas de los ejemplos, podemos dar un algoritmo para la construcción de tableros semánticos en el Cálculo de Predicados.

Algoritmo 2.19 (Construcción de un tablero semántico)

Input*: Una fórmula A del cálculo de predicados.*

Output*: Un tablero semántico $\mathcal{T}$ para A: sus ramas son finitas o infinitas con hojas marcadas como abiertas o cerradas.*

Un tablero semántico $\mathcal{T}$ para A es un árbol en el que cada nodo está etiquetado con un par $W(n) = (U(n), C(n))$, donde $U(n)$ es un conjunto de fórmulas y $C(n)$ contiene los símbolos de constante que aparecen en $U(n)$. Inicialmente, el árbol consta de un sólo nodo, la raíz, etiquetada por el conjunto de fórmulas $\{A\}$ y por el conjunto $\{a_1, \ldots, a_k\}$ que es el conjunto de símbolos de constante que aparecen en A. En el caso de que la fórmula A no contenga ningún símbolo de constante, escogemos uno arbitrario, digamos a, y etiquetamos el nodo raíz con $(\{A\}, \{a\})$. El tablero se construye inductivamente aplicando alguna de las siguientes reglas a una hoja etiquetada por $W(l) = (U(l), C(l))$:

- *Si $U(l)$ es un conjunto de* **literales** *y existe un par que se contradicen, la rama se marca* **cerrada** $\times$*; en caso contrario se marca* **abierta** $\odot$*.*
- *Si $U(l)$ no es un conjunto de literales, escogemos una fórmula $F \in U(l)$ que no sea literal y se aplica alguna de las reglas:*
  - *Si $F \in U(l)$ es una α–fórmula, se crea un nuevo nodo hijo l', y se etiqueta con*

    $$W(l') = (U(l'), C(l')) = ((U(l) \setminus \{F\}) \cup \{\alpha_1, \alpha_2\}, C(l)).$$

    En el caso de que F sea de la forma $\neg\neg F_1$, no hay α_2.
  - *Si $F \in U(l)$ es una β–fórmula, se crean dos nuevos nodos hijos l' y l'', y se etiquetan*

    $$W(l') = (U(l'), C(l') = ((U(l) \setminus F\}) \cup \{\beta_1\}, C(l)),$$

    $$W(l'') = (U(l''), C(l'') = ((U(l) \setminus F\}) \cup \{\beta_2\}, C(l)),$$

- *Si $F \in U(l)$ es una δ–fórmula, se crea un nuevo nodo hijo l', y se etiqueta con*

$$W(l') = (U(l'), C(l')) = (U(l) \setminus \{F\}) \cup \{\delta(b)\}, C(l) \cup \{b\}).$$

  de acuerdo a las tablas anteriores y siendo b una constante que no aparezca en $U(l)$.

- *Si $F \in U(l)$ es una γ–fórmula y $b \in C(l)$, se crea un nuevo nodo hijo l' y se etiqueta*

$$W(l') = (U(l'), C(l')) = (U(l) \cup \{\gamma(b)\}, C(l)).$$

- *Si todas las reglas que se puedan aplicar a un nodo no generan un hijo distinto, y no es cerrado, se marca como abierto $\odot$.*

Nota 2.20 *Es importante señalar que el algoritmo no sirve, en general, para buscar un modelo para la fórmula dada. Es obvio que sólo podremos exhibir un modelo finito y es posible que haya fórmulas satisfacibles que no admitan modelos finitos. Un ejemplo de esta situación está dado en el Teorema que sigue.*

Teorema 2.21 *La fórmula $A_1 \wedge A_2 \wedge A_3$, donde*

$$A_1 = \forall X \exists Y p(X, Y)$$

$$A_2 = \forall X \neg p(X, X)$$

$$A_3 = \forall X \forall Y \forall Z (p(X, Y) \wedge p(Y, Z)) \rightarrow p(X, Z)$$

no admite un modelo finito.

Demostración .– Supongamos, por reducción al absurdo, que la fórmula sí admite un modelo con un dominio finito D y consideremos $d_1 \in D$.
Por A_1, existe $d_2 \in D$ t.q. $p(d_1, d_2)$. Repitiendo el proceso, construimos una secuencia de elementos de D $d_1, \ldots, d_n, \ldots$, tales que $p(d_i, d_{i+1})$.
Por A_3, se tendría que, para todo $i < j$,con $i, j \in \mathbb{N}$, $p(d_i, d_j)$.
Como D es finito, existen $i, j \in \mathbb{N}$ t.q. $d_i = d_j$, por lo tanto, gracias a A_2, tenemos $\neg p(d_i, d_j)$. Lo que contradice que $p(d_i, d_j)$. ∎

Finalizamos la Sección estableciendo la robustez y completitud de los tableros semánticos.

Teorema 2.22 *Sea A una fórmula válida. La construcción del tablero para $\neg A$ cierra.*

Teorema 2.23 *Sea A una fórmula y T un tablero para A. Si T es cerrado, A es insatisfacible.*

Ejemplo 2.24 *Veamos algunos ejemplos de construcción de tableros semánticos aplicados a la decisión sobre la satisfabilidad de fórmulas del Cálculo de Predicados.*

- *Probar la validez de la fórmula*

$$(\forall X(p(X) \longrightarrow q(X))) \longrightarrow ((\forall p(X)) \longrightarrow (\exists X q(X)))$$

- *Utilizar tableros semánticos para probar la validez de la fórmula:*

$$(\forall X(p(X) \wedge q(X))) \leftrightarrow (\forall X p(X) \wedge \forall X q(X)).$$

- *Estudiar la validez de la fórmula*

$$(\forall X \forall Y(p(X,Y) \longrightarrow q(Y))) \longrightarrow ((\forall X \forall Y p(X,Y)) \wedge (\exists Y q(Y)))$$

2.2. Skolenización

Recordemos que un término se dice básico si no contiene variables. Del mismo modo, se tiene:

Definición 2.25 *Una fórmula se dice* **básica** *o* **base** *si no contiene variables ni cuantificadores. Una fórmula A' se dice que es una instancia base de una fórmula A si, y sólo si, A' puede obtenerse de A sustituyendo las variables de A por términos básicos.*

Si en el Cálculo Proposicional la aplicación de la resolución exigía que las fórmulas estuviesen dadas en forma clausal, la resolución en el Cálculo de Predicados requiere algo similar.

Definición 2.26 (Forma normal conjuntiva prenexa) *Se dice que una fórmula está en forma normal conjuntiva prenexa si es de la forma*

$$Q_1 X_1 \cdots Q_n X_n M,$$

donde cada Q_i, $1 \leq i \leq n$ es un cuantificador y M es una fórmula libre de cuantificadores en forma normal conjuntiva. Normalmente, $Q_1 X_1 \cdots Q_n X_n$ recibe el nombre de prefijo y M el de matriz.

Definición 2.27 (Forma clausal, cláusula base) *Se dice que una fórmula cerrada está en forma clausal si está en forma normal conjuntiva prenexa y todos sus cuantificadores son universales. Una cláusula es una disyunción de literales y, una cláusula base es una instancia base de una cláusula.*

Ejemplo 2.28 *La fórmula siguiente está en forma clausal:*

$$\forall X \forall Y ([p(f(X)) \vee \neg p(g(Y)) \vee q(Y)] \wedge [\neg q(Y) \vee \neg q(g(Y)) \vee q(X)].$$

Para abreviar la notación, se suele escribir la forma clausal anterior como sigue:

$$\{\{p(f(X)), \neg p(g(Y)), q(Y)\}, \{\neg q(Y), \neg q(g(Y)), q(X)\}\}.$$

Teorema 2.29 (Skolem) *Sea A una fórmula cerrada. Entonces, existe una fórmula A' en forma clausal tal que $A \approx A'$.*

Nota 2.30 *La notación $A \approx A'$ no quiere decir que las fórmula A y A' sean lógicamente equivalentes. Sólo quiere decir que A es satisfacible si, y sólo, si A' es satisfacible.*
Por otro lado, es relativamente sencillo transformar una fórmula cualquiera en otra equivalente en forma normal conjuntiva prenexa. La falta de equivalencia lógica en el Teorema de Skolem se produce a la hora de eliminar los cuantificadores existenciales. Esta eliminación se lleva a cabo introduciendo nuevos símbolos de función. Por ejemplo, si consideramos la fórmula $A = \forall X \exists Y p(X, Y)$, se elimina el cuantificador existencial transformando la fórmula en $A' = \forall X p(X, f(X))$. Estamos expresando casi la misma idea: la fórmula A dice que para todo X, se puede producir un valor Y de tal forma que el predicado sea cierto. En la fórmula A', se explicita una forma de hacer esta elección. De ahí que las fórmulas no sean lógicamente equivalentes, pero sí que se comporten igual si de satisfabilidad hablamos. Es fácil ver que $\mathcal{I} = \{(\mathbb{Z}, \{\ >\})\}$ es un modelo para A. Sin embargo, no cualquier función $F(X)$ sirve para extender el modelo a un modelo para A'. Si tomamos $F(X) = X + 1$, la interpretación $\mathcal{I}' = \{(\mathbb{Z}, \{\ >\}, \{F(X) = X + 1\}\}$ no es un modelo para A'. Sustituyendo $F(X)$ por $G(X) = X - 1$ en $\mathcal{I}'$ sí se obtiene un modelo para A'.

Demostración del Teorema 2.29 Para demostrar el Teorema, llevaremos a cabo transformaciones análogas a las del Teorema 1.26. Más concretamente,

1. Renombramos las variables de manera que ninguna variable esté en el radio de acción de dos cuantificadores.

2. Eliminamos todos los operadores, salvo las conjunciones, disyunciones y negaciones, empleando las mismas equivalencias que en el Teorema 1.26.

3. Llevamos las negaciones junto a los átomos empleando las leyes de Morgan y las equivalencias:

$\neg\forall X A(X)$	$\exists X \neg A(X)$
$\neg\exists X A(X)$	$\forall X \neg A(X)$

4. Eliminamos las dobles negaciones empleando $\neg\neg A \equiv A$.

5. Extraemos los cuantificadores de la matriz. Repetidamente, escogemos un cuantificador que no esté en el radio de acción de otro cuantificador aún en la matriz y lo extraemos de la matriz empleando las siguientes equivalencias:

$$A \; op \; QXB(X) \equiv QX(A \; op \; B(X)) \text{ y}$$
$$QXA(X) \; op \; B \equiv QX(A(X) \; op \; B),$$

donde Q es cualquier cuantificador y op es $\wedge$ o $\vee$.

6. Aplicamos las leyes distributivas para transformar la matriz a forma normal conjuntiva.

7. Eliminamos los cuantificadores existenciales como sigue: si un cierto cuantificador $\exists X$ no está precedido por un cuantificador universal, consideramos un nuevo símbolo de función 0-ario (símbolo de constante), digamos a, eliminamos $\exists X$ y sustituimos todas las apariciones de X por a. Si $\exists X$ está precedido por n cuantificadores universales, digamos $\forall X_1 \cdots \forall X_n$, consideramos un nuevo símbolo de función $n-$aria, digamos f, eliminamos $\exists X$ y sustituimos cada aparición de X por $f(X_1, \ldots, X_n)$. Los nuevos símbolo introducidos, a o f según el caso, se llaman funciones de Skolem.

La única transformación que merece especial atención es la última. Supongamos que reemplazamos un cuantificador existencial por una función de Skolem. Debemos probar que $A \approx A'$. Para ello, sea $\mathcal{I}$ un modelo para la fórmula $\forall X_1 \ldots \forall X_n \exists X p(X_1, \ldots, X_n, X)$ y sea A' la fórmula que introduce la correspondiente función de Skolem $\forall X_1 \ldots \forall X_n p(X_1, \ldots, X_n, f(X_1, \ldots, X_n))$. Extendemos $\mathcal{I}$ a una interpretación $\mathcal{I}'$ de A' sin más que asociar a f, la función F definida en el dominio mediante: dados $c_1, \ldots, c_n$, se define $F(c_1, \ldots, c_n) =$

c_{n+1}, donde c_{n+1} es tal que $(c_1, \ldots, c_{n+1}) \in P$, siendo P la relación asociada al predicado p. Dicho c_{n+1} siempre existe al ser $\mathcal{I}$ un modelo para A. Además, resulta claro que $\mathcal{I}'$ es un modelo para A'.
Probemos, ahora, que si A' es satisfacible, A es satisfacible. Sea $\mathcal{I}'$ un modelo para A' y F la función asociada a f. La interpretación $\mathcal{I}$, obtenida de $\mathcal{I}'$, eliminando el símbolo f es un modelo para A. Dado cualquier $(c_1, \ldots, c_n)$ en el dominio, existe otro c en el dominio, basta tomar $c = F(c_1, \ldots, c_n)$, tal que $p(c_1, \ldots, c_n, c)$. ■

Ejemplo 2.31 *Skolenizar cada una de las siguientes fórmulas del Cálculo de Predicados*

1. $\forall X(p(X) \longrightarrow q(X)) \longrightarrow (\forall X p(X) \longrightarrow \forall X q(X))$
2. $\forall X \exists Y(p(X,Y) \longrightarrow q(Y)) \longrightarrow (\exists X p(X) \wedge \forall X \exists Y r(X,Y))$
3. $\exists X \forall Y p(X,Y) \longrightarrow \forall Y \exists X p(X,Y)$

2.3. Modelos de Herbrand

¿Cómo podemos aplicar la resolución vista en el Cálculo Proposicional para establecer un algoritmo para la satisfabilidad en el Cálculo de Predicados? Una idea simple sería la de transformar la fórmula a forma clausal. Después, podríamos generar instancias base de las cláusulas y aplicar a las mismas el proceso de resolución descrito en la Sección 1.4. Sin embargo, las preguntas son obvias: ¿cómo escojo tales instancias? La pregunta no es fácil de resolver y más si tenemos en cuenta que el conjunto de instancias base es infinito. El Teorema de Herbrand (Teorema 2.37) que veremos a continuación prueba que, de hecho, bastaría considerar un número finito de instancias base. Sin embargo, la pregunta sobre la forma de elección de esas instancias persite. De hecho, no existe forma eficiente de generar las instancias base de las que habla el Teorema de Herbrand. En los años sesenta del siglo pasado, Robinson demostró, sin embargo, que la resolución es a menudo un procedimiento práctico cuando se aplica a cláusulas que no son básicas. De eso nos ocuparemos en la Sección 2.5.

Definición 2.32 (Universo de Herbrand) *Sea S un conjunto de cláusulas, A el conjunto de símbolos de constante en S y F el conjunto de símbolos de función en S. El* **universo de Herbrand** *de S, H_S se define como:*

$$\begin{aligned} a &\in H_S && \text{si } a \in A \\ f(t_1, \ldots, t_n) &\in H_S && \text{si } f \in F, t_j \in H_S. \end{aligned}$$

En el caso de que el conjunto de cláusulas no contenga símbolos de constante, se considera que el conjunto A está formado por un símbolo de constante cualquiera, digamos $A = \{a\}$.

El universo de Herbrand es, justamente, el conjunto de términos base que se pueden formar a partir de los símbolos en S. Obviamente, si S contiene un símbolo de función, el universo de Herbrand es infinito.

Definición 2.33 (Base de Herbrand) *Sea H_S el universo de Herbrand para un conjunto de cláusulas S. Se dice* **base de Herbrand** *de S, B_S, al conjunto de átomos que pueden formarse usando símbolos de predicado en S y términos de H_S.*

Definición 2.34 (Interpretación de Herbrand) *Sea S un conjunto de cláusulas. Se llama* **interpretación de Herbrand** *para S es una interpretación para S cuyo dominio es el universo de Herbrand de S y de tal forma que los símbolos de constante y de función se asignan a sí mismos. Un modelo de Herbrand para S es una interpretación de Herbrand que satisface S. Puede identificarse con el subconjunto de la base de Herbrand para los que $\nu(p(t_1, \ldots, t_n)) = T$.*

Ejemplo 2.35 *Para ilustrar las definiciones anteriores, sea*

$$S = \{\{\neg p(a, f(X, Y))\}, \{p(b, f(X, Y))\}\}.$$

El universo de Herbrand de S es

$$H_S = \{a, b, f(a, a), f(a, b), f(b, a), f(b, b), f(a, f(a, a)), f(f(a, a), a), \ldots\}.$$

La base de Herbrand de S es

$$B_S = \{p(c, f(t_1, t_2)) : c \in \{a, b\} \text{ y } t_1, t_2 \in H_S\}.$$

Un modelo de Herbrand para la fórmula viene definido por el subconjunto de la base de Herbrand dado por

$$\{p(b, d) : p(b, d) \in B_S\}$$

Teorema 2.36 *Sea S un conjunto de cláusulas. Si S posee un modelo, posee un modelo de Herbrand.*

Teorema 2.37 (Herbrand) *Un conjunto de clásulas es insatisfacible si y sólo si un subconjunto finito de instancias base de las mismas es insatisfacible.*

2.3.1. Resolución base

Definición 2.38 *Dadas dos cláusulas base C_1 y C_2 tales que $l \in C_1$ y $l^c \in C_2$ (cláusulas contradictorias/generatrices), se define la resolvente de C_1 y C_2 como la cláusula*

$$Res(C_1, C_2) = (C_1 \setminus \{l\}) \cup (C_2 \setminus \{l^c\})$$

Teorema 2.39 *La resolvente de dos cláusulas base C_1 y C_2 es satisfacible si y sólo si C_1 y C_2 son simultáneamente satisfacibles.*

Demostración .– Sean C_1 y C_2 cláusulas simultáneamente satisfacibles con $l \in C_1$ y $l^c \in C_2$. Por el Teorema 2.36, poseen un modelo de Herbrand $\mathcal{H}$. Sea B el subconjunto de la base de Herbrand que define $\mathcal{H}$, es decir,

$$B = \{p(c_1, \ldots, c_n) : \nu_{\mathcal{H}}(p(c_1, \ldots, c_n)) = T\}$$

para términos base c_i. Dos literales base no pueden ser, al mismo tiempo, elementos de B. Por lo tanto, si $l \in B$, para que se satisfaga C_2, debe existir un literal $l' \in C_2$ tal que $l' \in B$. Por construcción, $l' \in C$, luego $\nu_{\mathcal{H}}(l') = T$ y, en consecuencia, $\mathcal{H}$ es un modelo para C. Si $l^c \in B$, el argumento es similar. Recíprocamente, si C es satisfacible, es satisfacible para una interpretación de Herbrand $\mathcal{H}$ que estará definida por un subconjunto B de su base de Herbrand. Para algún literal $l' \in C$ y $l' \in B$. Por construcción de la resolvente, $l' \in C_1$ o $l' \in C_2$ y $l, l^c \notin C$. Supongamos que $l' \in C_1$. Podemos extender $\mathcal{H}$ a un modelo $\mathcal{H}'$ para C_1 y C_2, definiendo $B' = B \cup \{l^c\}$. Si $l' \in C_2$, el argumento es idéntico.

■

2.4. Sustitución y unificación

Definición 2.40 (Sustitución) *Llamaremos sustitución a todo conjunto de la forma:*

$$\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\},$$

donde cada X_i es una variable y cada t_i es un término distinto de la variable X_i ,para $1 \leq i \leq n$.

Definición 2.41 *Se llama expresión a todo término, cláusula o conjunto de cláusulas. Dada una expresión E y una sustitución θ, llamaremos instancia $E\theta$ de E al resultado de aplicar θ a E, es decir, a la expresión obtenida a partir de E sustituyendo simultáneamente cada aparición de X_i en E por t_i.*

Ejemplo 2.42 *Sea $E = p(X) \vee q(f(Y))$ y $\theta = \{X \leftarrow Y, Y \leftarrow f(a)\}$. Entonces $E\theta = p(Y) \vee q(f(f(a)))$.*

Definición 2.43 (Composición de substituciones) *Sean*

$$\theta = \{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\} \text{ y}$$

$$\sigma = \{Y_1 \leftarrow s_1, \ldots, Y_m \leftarrow s_m\}$$

dos sustituciones cualesquiera. Si $\mathcal{X}$ e $\mathcal{Y}$ son sus respectivos conjuntos de variables, definimos la composición de θ con σ, denotada $\theta\sigma$, mediante:

$$\theta\sigma := \{X_i \leftarrow t_i\sigma : X_i \in \mathcal{X}, X_i \neq t_i\sigma\} \cup \{Y_j \leftarrow s_j : Y_j \in \mathcal{Y},\ Y_j \notin \mathcal{X}\}.$$

Dicho en palabras: se aplica la sustitución σ a los términos t_i de θ (siempre y cuando no resulte $X_i \leftarrow X_i$) y se añaden las sustituciones de σ en las que no aparecen variables que ya estén en θ.

Ejemplo 2.44 *Sean*

$$\theta = \{X \leftarrow f(Y), Y \leftarrow f(a), Z \leftarrow U\}$$

$$\sigma = \{Y \leftarrow g(a), U \leftarrow Z, V \leftarrow f(f(a))\}$$

y sea $E = p(U, V, X, Y, Z)$. Entonces,

$$\theta\sigma = \{X \leftarrow f(g(a)), Y \leftarrow f(a), U \leftarrow Z, V \leftarrow f(f(a))\}$$

$$E(\theta\sigma) = p(Z, f(f(a)), f(g(a)), f(a), Z).$$

Es muy fácil ver que $E(\sigma\theta) = (E\theta)\sigma$.

Teorema 2.45 *Si θ y σ son sustituciones y E es una expresión, se tiene $E(\sigma\theta) = (E\theta)\sigma$. Por otro lado, la composición de sustituciones es asociativa.*

La clave para la resolución es la conocida como unificación. Por ejemplo, las cláusulas $p(f(X), g(Y))$ y $\neg p(f(f(a)), g(Z))$ no se pueden resolver pues no son contradictorias. Ahora bien, la sustitución $\{X \leftarrow f(a), Z \leftarrow Y\}$ produce dos literales complementarios: $p(f(f(a)), g(Z))$ y $\neg p(f(f(a)), g(Z))$. A pesar de que estos dos literales no son básicos, podríamos aplicarles resolución como veremos en la próxima Sección. En este caso, la sustitución convierte al primer átomo y a la negación del segundo en idénticos. A este tipo de sustituciones se les dice unificaciones.

Definición 2.46 (Unificación, unificación más general) *Dado un conjunto de átomos, una unificación es una sustitución que hace los átomos idénticos. Una unificación más general*[2] *es una unificación θ tal que para cualquier otra unificación σ, existe una tercera μ satisfaciendo $\sigma = \theta\mu$.*

No todos los átomos son unificables. Evidentemente, dos átomos cuyos símbolos de predicado son diferentes no son unificables. Tampoco son unificables, por ejemplo, $p(X)$ y $p(f(X))$.
Como para que dos átomos sean unificables sus símbolos de predicado deben ser iguales (es decir, sus símbolos y su aridad son iguales), la unificabilidad de dos átomos se describe mediantes ecuaciones de términos. Así, la unificabilidad de los predicados $p(f(X), g(Y))$ y $p(f(f(a)), g(Z))$ se expresa por las ecuaciones

$$\begin{aligned} f(X) &= f(f(a)) \\ g(Y) &= g(Z) \end{aligned}$$

En lo que resta de Sección, consideraremos ecuaciones de términos.

Definición 2.47 (Forma resuelta) *Diremos que un conjunto de ecuaciones de términos está en forma resuelta si todas las ecuaciones son de la forma $X_i = t_i$, con X_i variable, y cada variable que aparece en la parte izquierda de las ecuaciones, no aparece en término alguno. Una ecuación en forma resuelta define, de manera obvia, una sustitución.*

Algoritmo 2.48 (Unificación)

Input*: Un conjunto de ecuaciones de términos.*

Output*: Un conjunto en forma resuelta (determinando una unificación más general) o "no es posible la unificación".*

[2] A partir de este momento se usará, cuando convenga, la abreviatura umg por unificación más general.

El algoritmo simplemente aplica las siguientes reglas hasta que el conjunto está en forma resuelta o se determina la imposibilidad de obtener una tal forma para el conjunto de ecuaciones dado:

1. *Transforma las ecuaciones de la forma $t = X$, con t término y X variable, en ecuaciones de la forma $X = t$.*

2. *Elimina toda ecuación de la forma $X = X$, con X variable.*

3. *Dada una ecuación de la forma $t = t'$ con t y t' no variables, lleva a cabo las siguientes tareas: para, indicando la imposibilidad de la forma resuelta, si los símbolos de función más exteriores son distintos o tienen distinta aridad. En caso contrario, si los símbolos de función más exteriores son iguales y tienen la misma aridad, i.e., si $t = f(t_1, \ldots, t_n)$ y $t' = f(t'_1, \ldots, t'_n)$, substituye la ecuación $t = t'$ por las ecuaciones $t_1 = t'_1, \ldots, t_n = t'_n$.*

4. *Dada una ecuación de la forma $X = t$, tal que X aparece en algún otro término, para indicando la imposibilidad de la forma resuelta si X aparece en t. En caso contrario, si X no aparece en t, substituye X por t allá donde aparezca X.*

Teorema 2.49 *El algoritmo anterior siempre termina. Además, si determina que la unificación no es posible, es que no hay unificación posible. Si termina devolviendo un conjunto de ecuaciones en forma resuelta $\{X_1 = t_1, \ldots, X_n = t_n\}$, la sustitución que define*

$$\{X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n\}$$

es una umg del conjunto de ecuaciones.

Demostración .– Claramente el algoritmo termina puesto que cualquiera de las cuatro reglas dadas sólo puede aplicarse un número finito de veces (la única que quizás merece un poco más de atención es la cuarta, pero es fácil ver que, cada vez que se aplica, el número de apariciones de una cierta variable disminuye).
Si el algoritmo termina devolviendo la imposibilidad de unificación, lo hace usando las reglas 3 o 4 y, en esos casos, la unificación es imposible.
Falta por ver que cuando el algoritmo termina con éxito, las ecuaciones que devuelve son umg. Para ello, veremos, en primer lugar que cualquiera de las reglas del algoritmo preservan el conjunto de unificadores. Esto es obvio en el caso de las reglas 1 y 2. Veamos qué ocurre para la regla 3 aplicada a

$t = f(t_1, \ldots, t_n)$ y $t' = f(t'_1, \ldots, t'_n)$. Si $t\sigma = t'\sigma$ (σ unifica t y t'), es obvio que $t_i\sigma = t'_i\sigma$ para todo i. Recíprocamente, si $t_i\sigma = t'_i\sigma$ para todo i es obvio que $t\sigma = t'\sigma$.
Sea ahora $t_1 = t_2$ una ecuación a la que se aplica la regla 4 a $X = t$ para transformarla en $u_1 = u_2$. Tras aplicar la regla, $X = t$ permanece en el conjunto de ecuaciones. Luego cualquier unificador σ del conjunto de ecuaciones modificado verifica $X\sigma = t\sigma$. Entonces,

$$u_i\sigma = (t_i\{X \leftarrow t\})\sigma = t_i(\{X \leftarrow t\}\sigma) = t_i\sigma.$$

Así pues, si σ unifica t_1 y t_2, también unifica u_1 y u_2.
Después de esto, es trivial que la sustitución que da el algoritmo es umg. Al ser el conjunto de términos en forma resuelta una unificación, concluimos que ésta es, forzosamente, la más general. ■

Ejemplo 2.50 *Estudiar si es posible unificar los siguientes conjuntos de fórmulas. En caso de que sea posible la unificación, determinar una umg.*

1. $\{p(g(Y), f(X, h(X), Y)), p(X, f(g(Z), W, Z))\}$
2. $\{p(a, X, f(g(Y))), p(Y, f(Z), f(Z)))\}$
3. $\{p(f(X, X), g(Y), Z)), p(f(a, V), g(b), h(W)))\}$

2.5. Resolución

Una vez estudiada la unificación, podemos definir la resolución aplicada a cláusulas que no son base, usando la unificación como parte integral de la regla de resolución.
Antes de definir la regla de resolución, necesitamos introducir la siguiente notación. Si $L = \{l_1, \ldots, l_n\}$ es un conjunto de literales, se denota por L^c el conjunto de literales $L^c = \{l_1^c, \ldots, l_n^c\}$.

Definición 2.51 (Regla de Resolución) *Sean C_1 y C_2 dos cláusulas* **sin variables en común**, *de tal forma que existen subconjuntos de literales $L_1 = \{l_1^1, \ldots, l_n^1\} \subseteq C_1$ y $L_2 = \{l_1^2, \ldots, l_m^2\} \subseteq C_2$ satisfaciendo que L_1 y L_2^c pueden unificarse por una u.m.g. σ (en ese caso diremos que C_1 y C_2 se contradicen o que son generatrices). Se define la resolvente de C_1 y C_2 mediante*

$$Res(C_1, C_2) = (C_1\sigma \setminus L_1\sigma) \cup (C_2\sigma \setminus L_2\sigma\})$$

Obsérvese que la regla de resolución requiere que las cláusulas no tengan variables en común. Esto siempre se puede conseguir renombrando las variables en una de las cláusulas antes de usarla en la regla de resolución. Como las variables en las cláusulas están cuantificadas universalmente, esto no cambia la satisfabilidad.

Ejemplo 2.52 *Consideremos las cláusulas* $p(f(X))$ *y* $\neg p(X)$*. Para aplicar resolución, renombramos la variable de la segunda cláusula:* $\neg p(X_1)$*. Un unificador más general para las cláusulas es* $\{X_1 \leftarrow f(X)\}$ *y aplicando resolución a* $p(f(X))$ *y* $\neg p(f(X))$*, derivamos* $\square$*.*

Algoritmo 2.53

Input*: Una forma clausal* S*.*

Output*: La forma clausal* S *es satisfacible o insatisfacible.*

Comenzamos con el conjunto $S_0 = S$*. Supongamos que hemos construido* S_i*. Si existen dos cláusulas generatrices* $C_1, C_2 \in S_i$ *en* S_i*, sea* $C = Res(C_1, C_2)$*. Si* $C = \square$*, se acaba el algoritmo y* S *es insatisfacible. En caso contrario,*

$$S_{i+1} \leftarrow S_i \cup Res(C_1, C_2)$$

Si $S_{i+1} = S_i$ *para todos los pares posibles de cláusulas contradictorias, se termina el proceso y* S *es satisfacible.*

Ejemplo 2.54 *En el ejemplo, las líneas* 1 *a* 7 *especifican un conjunto de cláusulas. En el resto de líneas se deriva la cláusula vacía mediante resolución. Cada línea contiene la resolvente, la unificación más general y las cláusulas padre.*

1.- $\neg p(X) \vee q(X) \vee r(X, f(X))$		
2.- $\neg p(X) \vee q(X) \vee s(f(X))$		
3.- $t(a)$		
4.- $p(a)$		
5.- $\neg r(a, Y) \vee t(Y)$		
6.- $\neg t(X) \vee \neg q(X)$		
7.- $\neg t(X) \vee \neg s(X)$		
8.- $\neg q(a)$	$\{X \leftarrow a\}$	$3, 6$
9.- $q(a) \vee s(f(a))$	$\{X \leftarrow a\}$	$2, 4$
10.- $s(f(a))$		$8, 9$
11.- $q(a) \vee r(a, f(a))$	$\{X \leftarrow a\}$	$1, 4$
12.- $r(a, f(a))$		$8, 11$
13.- $t(f(a))$	$\{Y \leftarrow f(a)\}$	$5, 12$
14.- $\neg s(f(a))$	$\{X \leftarrow f(a)\}$	$7, 13$
15.- $\square$		$10, 14$

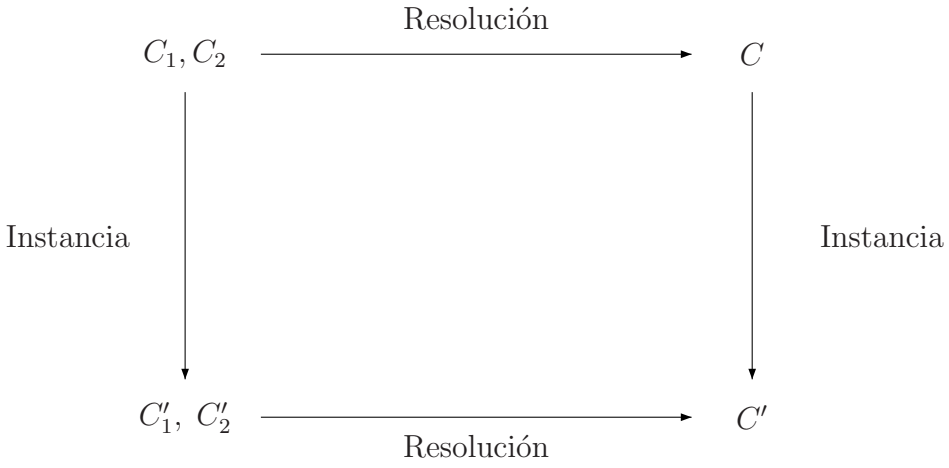
Ejemplo 2.55 *Estudiar si podemos derivar la cláusula vacía usando resolución aplicada a la forma clausal*

$$\{\neg p(X,Y) \vee p(Y,X), \neg p(X,Y) \vee \neg p(Y,Z) \vee p(X,Z), p(X,f(X)), \neg p(X,X)\}$$

2.5.1. Robustez y completitud de la resolución

Teorema 2.56 (Lema de levantamiento) *Supongamos que C_1' y C_2' son dos instancias básicas de C_1 y C_2 respectivamente y sea C' una resolvente básica de C_1' y C_2'. Entonces, existe una resolvente C de C_1 y C_2 para la cual C' es instancia básica.*

La situación descrita por el Teorema puede visualizarse así:



Demostración del Teorema 2.56.- Primero, se renombran las variables para que C_1 y C_2 no tengan variables comunes. Sean $l \in C_1'$ y $l^c \in C_2'$ los literales generadores de la resolución básica. Puesto que C_1' es una instancia base de C_1 y $l \in C_1'$, debe existir un conjunto de literales $L_1 \subseteq C_1$ tal que l es una instancia base de cada literal en L_1. De manera similar, existe un subconjunto $L_2 \subseteq C_2$ para l^c. Sean λ_1 y λ_2 unificaciones más generales para L_1 y L_2 respectivamente y sea $\lambda = \lambda_1 \cup \lambda_2$. λ es una sustitución puesto que L_1 y L_2 no tienen variables en común. $C_1\lambda$ y $C_2\lambda$ son cláusulas contradictorias para $L_1\lambda$ y $L_2\lambda$. Sea σ su umg y C su resolvente:

$$C = ((C_1\lambda)\sigma \setminus \{((L_1\lambda)\sigma)\}) \cup ((C_2\lambda)\sigma \setminus \{((L_2\lambda)\sigma)\}) =$$

$$= (C_1(\lambda\sigma) \setminus \{L_1(\lambda\sigma)\}) \cup (C_2(\lambda\sigma) \setminus \{L_2(\lambda\sigma)\}).$$

Si $\lambda\sigma$ es umg para L_1 y L_2^c, C será una resolvente de C_1 y C_2. Pero λ define una instanciación, luego se reduce a ecuaciones de la forma $X \leftarrow t$ para variables distintas y σ es umg, luego $\lambda\sigma$ es un conjunto reducido de ecuaciones, todas ellas necesarias para unificar $L_1\lambda$ y L_2^c. Por lo tanto, $\lambda\sigma$ es umg.

Ahora, puesto que C_1' y C_2' son instancias base de C_1 y C_2:

$$C_1' = C_1\lambda\sigma\theta_1 \text{ y } C_2' = C_2\lambda\sigma\theta_2$$

para ciertas sustituciones θ_1 y θ_2. Sea $\theta = \theta_1 \cup \theta_2$. Entonces, $C' = C\theta$, por lo que C' es una instancia básica de C. ■

Usando el lema del levantamiento, se puede probar que

Teorema 2.57 *Sean C_1 y C_2 dos clásulas que permiten resolución. Entonces, C_1 y C_2 son simultáneamente satisfacibles si, y sólo si, $Res(C_1, C_2)$ es satisfacible para cualquier resolución.*

Demostración .– Supongamos, en primer lugar, que C_1 y C_2 son simultáneamente satisfacibles, que son contradictorias para una umg σ y que $C = Res(C_1, C_2)$ para la citada unificación. Entonces, existe una interpretación $\mathcal{I}$ tal que $\nu_{\mathcal{I}}(C_i) = T$[3]. En consecuencia, existen instancias base $C_i' = C_i\lambda_i$ tal que $\nu_{\mathcal{I}}(C_i') = T$. Puesto que σ es una umg, hay sustituciones θ_i tales que $\lambda_i = \sigma\theta_i$. Entonces, $C_i' = C_i\lambda = C_i(\sigma\theta_i) = (C_i\sigma)\theta_i$, lo que demuestra que $C_i\sigma$ es satisfacible para la misma interpretación.
Supongamos que las cláusulas de C_1 y C_2 que sirven para resolver se unifican en l y l^c respectivamente. Sólo uno de entre l o l^c puede ser satisfacible en la interpretación $\mathcal{I}$. Supongamos, sin restricción de generalidad, que $\nu_{\mathcal{I}}(l) = T$. Puesto que $C_2\sigma$ es satisfacible, debe existir un literal $l' \in C_2$ distinto de l^c y tal que $\nu_{\mathcal{I}}(l'\sigma) = T$. Por la construcción de la resolvente, $l' \in C$ y, por lo tanto, $\nu_{\mathcal{I}}(C) = T$.
Ahora, supongamos que $Res(C_1, C_2)$ es satisfacible para cualquier resolución. Por reducción al absurdo, asumamos que C_1 y C_2 no son simultáneamente satisfacibles. Entonces, tienen que existir instancias base C_1' y C_2' de C_1 y C_2 mutuamente insatisfacibles y generatrices. La resolvente C' de C_1' y C_2' es insatisfacible. Por el lema del levantamiento, existe una cláusula, C que es una resolvente de C_1 y C_2 tal que C' es una instancia básica suya. En consecuencia, C es insatisfacible contradiciendo la hipótesis de que toda resolvente de C_1 y C_2 es satisfacible.

■

[3]Aunque no se diga explícitamente, a lo largo de la demostración, i hace referencia tanto a 1 como a 2.

Una vez establecido este resultado y usando los argumentos de su demostración, la prueba de la robustez y completitud de la resolución es la misma que la del caso de la resolución en el Cálculo Proposicional, salvo que cuando se define nodo de fallo en un árbol semántico para el conjunto de clásulas S, se define cambiando falsificando de una cláusula por falsificación de alguna *instancia base* de una cláusula. Del mismo modo, rama cerrada quiere decir rama que falsifica una instancia base de S.

Teorema 2.58 (Robustez) *Si la resolución deriva la cláusula vacía, el conjunto de cláusulas es insatisfacible*

Teorema 2.59 (Completitud) *Si el conjunto de cláusulas es insatisfacible, el proceso de resolución puede derivar la cláusula vacía* $\square$.

Ejemplo 2.60 *¿De las fórmulas*

$$\forall X \forall Y (\neg p(X) \vee r(X,Y)),$$

$$\neg(\forall X \forall Y (q(Y) \rightarrow r(X,Y)) \text{ y}$$
$$\forall X \exists Y (p(X) \vee s(X,Y))),$$

se deduce $\exists X \exists Y r(X,Y)$*? Usar resolución para responder a la pregunta.*

Capítulo 3

Resolución SLD

3.1. Introducción

La mayor parte de los axiomas en cualquier teoría tienen la forma

Si premisa-1 y premisa-2 y ... entonces conclusión.

Formalmente

$$A = \forall X_1 \cdots \forall X_m (B_1 \wedge \ldots \wedge B_m \longrightarrow B),$$

donde los B_i y B son átomos. Es decir, si ocurre $B_1, \ldots, B_n$, entonces se tiene B. En forma clausal, el axioma es $A = \neg B_1 \vee \ldots \vee \neg B_n \vee B$. Para probar que una fórmula $G = G_1 \wedge \ldots \wedge G_k$, con los G_i átomos, es consecuencia lógica de un conjunto de axiomas, se añade la negación de G al conjunto de axiomas y se trata de construir una refutación por resolución. Típicamente $\neg G$ se denomina *cláusula objetivo*.
Como presumiblemente los axiomas se han escogido para que sean satisfacibles, carece de sentido realizar resolución entre los axiomas por lo que la resolución se realiza entre la cláusula objetivo con alguno de los axiomas. Como $\neg G$ es una disyunción de literales negativos sólo puede resolverse con el único literal positivo del axioma. Se obtiene, por lo tanto, una cláusula que sólo contiene literales negativos. Si entre los axiomas se encuentran axiomas sin antecedentes, esto es, axiomas cuya forma clausal es un átomo positivo, eventualmente, podríamos producir resolventes que cada vez contendrían menos literales y, finalmente, la cláusula vacía.
Si escribimos los axiomas usando $\leftarrow$, podemos interpretar la fórmula

$$A = \forall X_1 \cdots \forall X_n (B \leftarrow B_1 \wedge \ldots \wedge B_m),$$

como un procedimiento: para computar B, computar $B_1, \ldots, B_n$. Esta es la razón por la que se suelen escribir de esta forma los axiomas en Programación Lógica.

3.2. Resolución SLD

Definición 3.1 (Cláusulas de Horn) *Se dice* **cláusula de Horn** *a toda fórmula del tipo $A \leftarrow B_1, \ldots, B_n$ con, a lo más, un literal positivo. Si existe ese literal, A, se le dice* **cabeza** *y a los literales B_i, $1 \leq i \leq n$, se les dice* **cuerpo** *de la cláusula. En particular, llamaremos* **hechos** *a las cláusulas del tipo $A \leftarrow$ y* **cláusulas objetivo** *a las del tipo $\leftarrow B_1, \ldots, B_n$.*

Definición 3.2 *Un conjunto de cláusulas cuya cabeza tiene la misma letra de predicado se llamará* **procedimiento**. *Un conjunto de procedimientos se dice* **programa lógico**.

Ejemplo 3.3 *En el siguiente programa lógico las cláusulas C_3 a C_{14} forman una base de datos*[1].

$$C_1 = p(X,Y) \leftarrow q(X,Z), r(Z,Y)$$
$$C_2 = p(X,Y) \leftarrow s(X,Z), q(Z,W), r(W,Y)$$

$$\begin{array}{lll} C_3 = s(a,b) & C_4 = s(b,a) & C_5 = s(c,d) \\ C_6 = s(d,c) & C_7 = q(b,c) & C_8 = q(c,b) \\ C_9 = r(a,e) & C_{10} = r(b,e) & C_{11} = r(c,f) \\ C_{12} = r(d,f) & C_{13} = r(a,g) & C_{14} = r(b,g) \end{array}$$

Definición 3.4 *Dado un programa lógico P y una cláusula objetivo $\leftarrow G$, llamaremos* **respuesta correcta** *a toda sustitución θ tal que $P \vdash \forall(\neg G\theta)$.*

Ejemplo 3.5 *Sea P el conjunto de axiomas de la aritmética en $\mathbb{N}$ y $G = \neg(X = Y + 13)$. Entonces, $\theta = \{Y \leftarrow X - 13\}$ es una respuesta correcta pues*

$$P \vDash \forall(X = X - 13 + 13)$$

Obsérvese, por lo tanto, que una respuesta correcta no tiene que ser necesariamente una sustitución básica.

[1] Se suele llamar base de datos a un conjunto de hechos.

Supongamos que queremos saber si $B = \exists(B_1 \wedge \ldots \wedge B_n)$ es consecuencia lógica de un programa P, donde $\exists$ denota la clausura existencial[2] de la conjunción. Se tendrá $P \vDash B$ si, y sólo si, existe una sustitución base σ tal que:

$$P \vDash (B_1 \wedge \ldots \wedge B_n)\sigma.$$

Supongamos ahora que σ puede escribirse como composición de sustituciones $\sigma = \theta\lambda$ para cualquier sustitución básica λ. De hecho, esta composición siempre existe: en el peor de los casos, $\sigma = \theta$ y λ la sustitución vacía. Se tiene, por lo tanto que

$$P \vDash \forall((B_1 \wedge \ldots \wedge B_n)\theta).$$

Así pues, "B es consecuencia lógica de P" es equivalente a que θ sea una respuesta correcta para la cláusula objetivo definida por $G = \neg(B_1 \wedge \ldots \wedge B_n)$. Como

$$\forall G = \forall\neg(B_1 \wedge \ldots \wedge B_n) = \neg\exists(B_1 \wedge \ldots \wedge B_n) = \neg B,$$

se tiene que $P \vDash B$ si, y sólo si, $\nvDash P \wedge G$.

Ejemplo 3.6 *Pongamos que queremos construir una refutación de la cláusula objetivo* $\leftarrow p(d, Y)$ *con el programa dado en el Ejemplo 3.3. A cada paso, señalamos la cláusula objetivo escogida, la cláusula del programa con la que se hace resolución, la unificación para la resolución y la resolvente.*

- *P1:* $\leftarrow p(d, Y)$, C_2, $\{X \leftarrow d, Y_1 \leftarrow Y\}$, $\leftarrow s(d, Z), q(Z, W), r(W, Y)$.
- *P2:* $\leftarrow s(d, Z)$, C_6, $\{Z \leftarrow c\}$, $\leftarrow q(c, W), r(W, Y)$.
- *P3:* $\leftarrow q(c, W)$, C_8, $\{W \leftarrow b\}$, $\leftarrow r(b, Y)$.
- *P4:* $\leftarrow r(b, Y)$, C_{10}, $\{Y \leftarrow e\}$, $\square$.

La composición de las unificaciones es $\{X \leftarrow d, Z \leftarrow c\, W \leftarrow b, Y \leftarrow e\}$, *por lo que la sustitución* $\{Y \leftarrow e\}$ *es una respuesta correcta, es decir,* $P \vDash p(d, e)$. *En consecuencia,* $P \vDash \leftarrow p(d, Y) \equiv \neg p(d, Y)$.

En el ejemplo inmediatamente anterior hemos ido escogiendo literales de las diferentes cláusulas objetivo tomando siempre la situada más a la izquierda. Naturalmente, no es esta la única forma de hacerlo. Si no se fija una regla para escoger estas cláusulas, el algoritmo consecuente es no determinista.

[2]Se dice clausura existencial de una fórmula A con variables libres $X_1, \ldots, X_n$ a $\exists X_1 \cdots \exists X_n A$. Análogamente se define clausura universal.

Definición 3.7 (Regla de computación) *Se dice regla de computación a una función que, cuando se aplica a una cláusula objetivo, escoge uno y sólo uno de los átomos de la misma*[3].

Una vez introducidas las ideas anteriores, estamos en condiciones de definir lo que se entiende por derivación SLD.

Definición 3.8 *Sea P un programa, R una regla de computación y G una cláusula objetivo. Una derivación por SLD-resolución es una secuencia de pasos entre cláusulas objetivo y cláusulas del programa. La primera cláusula objetivo es $G_0 = G$ y, supuesta construida G_i, se construye G_{i+1} seleccionando $A_k \in G_i$ (mediante R), escogiendo $C_i \in P$ tal que su cabeza unifica con A_k gracias a una u.m.g. θ_i y resolviendo:*

$$G_i = \leftarrow A_1, \ldots, A_{k-1}, A_k, A_{k+1}, \ldots, A_n$$

$$C_i = A \leftarrow B_1, \ldots, B_l$$

$$A_k\theta_i = A\theta_i$$

$$G_{i+1} = \leftarrow (A_1, \ldots, A_{k-1}, B_1, \ldots, B_l, A_{k+1}, \ldots, A_n)\theta_i.$$

Se distinguen los siguientes tipos de derivación:

- *Infinita*: Se produce cuando en cualquier objetivo G_i de la secuencia, el átomo A_i escogido por la regla de computación unifica con (una variante) una cláusula del programa P.
- *Finita*: la derivación termina en un número finito de pasos. Atendiendo al último objetivo de la secuencia, G_n, las derivaciones pueden ser:
  - De éxito, si $G_n = \square$, es decir, una derivación de éxito es una refutación.
  - De fallo, en cualquier otro caso.

3.2.1. Robustez y completitud

Teorema 3.9 (Robustez) *Sea P un programa, R una regla de computación y G una cláusula objetivo. Supongamos que existe una refutación SLD de G. Sea $\theta = \theta_1 \cdots \theta_l$ la sucesión de unificadores y sea σ la restricción de θ a las variables de G. Entonces, σ es una respuesta corrrecta para G.*

[3]Por ejemplo, Prolog elige siempre el átomo que se encuentra más a la izquierda.

La refutación SLD no es completa para conjuntos de cláusulas en general. El conjunto de cláusulas

$$\{p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$$

es insatisfacible, pero no hay forma de derivar la clásula vacía si se exige que una de las cláusulas con la que resolver sea una de las dadas. Sin embargo, sí es completa para cláusulas de Horn.

Teorema 3.10 (Completitud) *Sea P un programa, R una regla de computación y G una cláusula objetivo. Sea σ una respuesta correcta. Entonces, existe una refutación SLD de G a partir de P tal que σ es la restricción de la sucesión de unificadores $\theta = \theta_1 \cdots \theta_n$ a las variables de G.*

3.3. Árboles SLD

En la Definición 3.8, no se especifica en qué orden elegir las cláusulas del programa con la que hacer resolución con el átomo elegido por la regla de computación. Cualquier forma de especificar cómo hacer esta elección se dice *regla de ordenación*.
Por otro lado, podemos describir todas las derivaciones SLD posibles en un árbol que, lógicamente, se llamará árbol SLD.

Definición 3.11 (Árbol SLD) *Sea un programa P y G una cláusula objetivo. Un árbol SLD, para unas ciertas reglas de computación y de ordenación, es un árbol definido por:*

1. *El nodo raíz de es el objetivo inicial.*
2. *Si $G_i \equiv\leftarrow A_1 \wedge \ldots \wedge A_k \wedge \ldots \wedge A_n$ es un nodo y A_k es el átomo seleccionado por la regla de computación, para cada cláusula $C_j \equiv A \leftarrow B_1 \wedge \ldots \wedge B_m$ de P cuya cabeza A unifica con A_k, escogidas en el orden dado por la regla de ordenación, con umg θ, el resolvente*

   $$G_{ij} \equiv\leftarrow (A_1 \wedge \ldots A_{k-1} \wedge B_1 \wedge \ldots \wedge B_m \wedge A_{k+1} \wedge \ldots \wedge A_n)\theta$$

   es un nodo del árbol hijo de G_i.

Obsérvese que las hojas del árbol se corresponden con cláusulas que no pueden resolverse con ninguna cláusula del programa. Estas hojas o bien son la cláusula vacía o nodos de fallo. Por otro lado, cada rama del árbol es una derivación SLD. Las ramas que se corresponden con derivaciones infinitas se

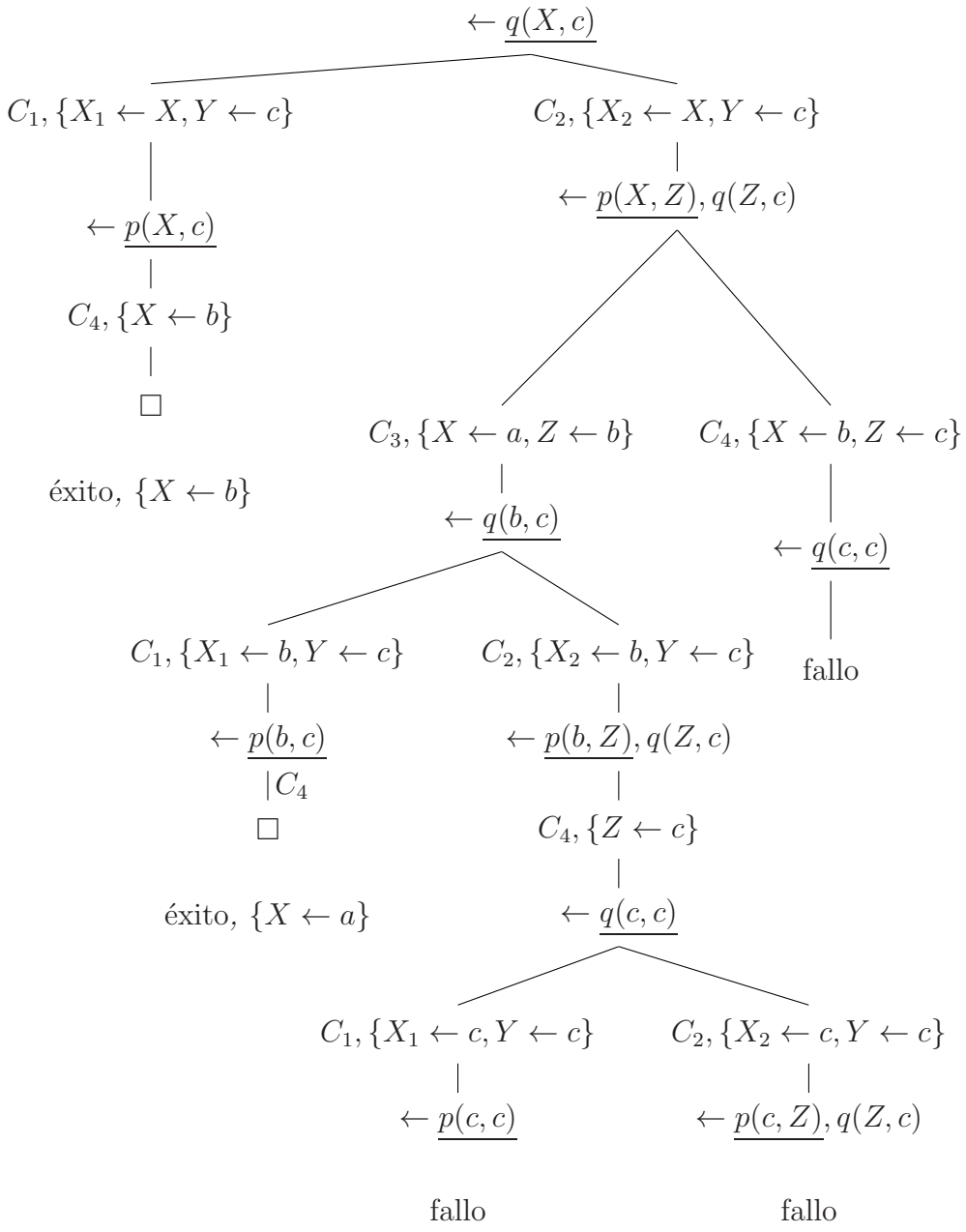
dicen ramas infinitas, las que se corresponden con refutaciones se dicen ramas de éxito y las que se corresponden con derivaciones de fallo, se dicen ramas de fallo. Como veremos a continuación, los árboles dependen de las reglas de computación y también de la regla de ordenación.

Ejemplo 3.12 *Consideremos el siguiente programa lógico:*

- $C_1 = q(X, Y) \leftarrow p(X, Y)$
- $C_2 = q(X, Y) \leftarrow p(X, Z), q(Z, Y)$
- $C_3 = p(a, b)$
- $C_4 = p(b, c)$

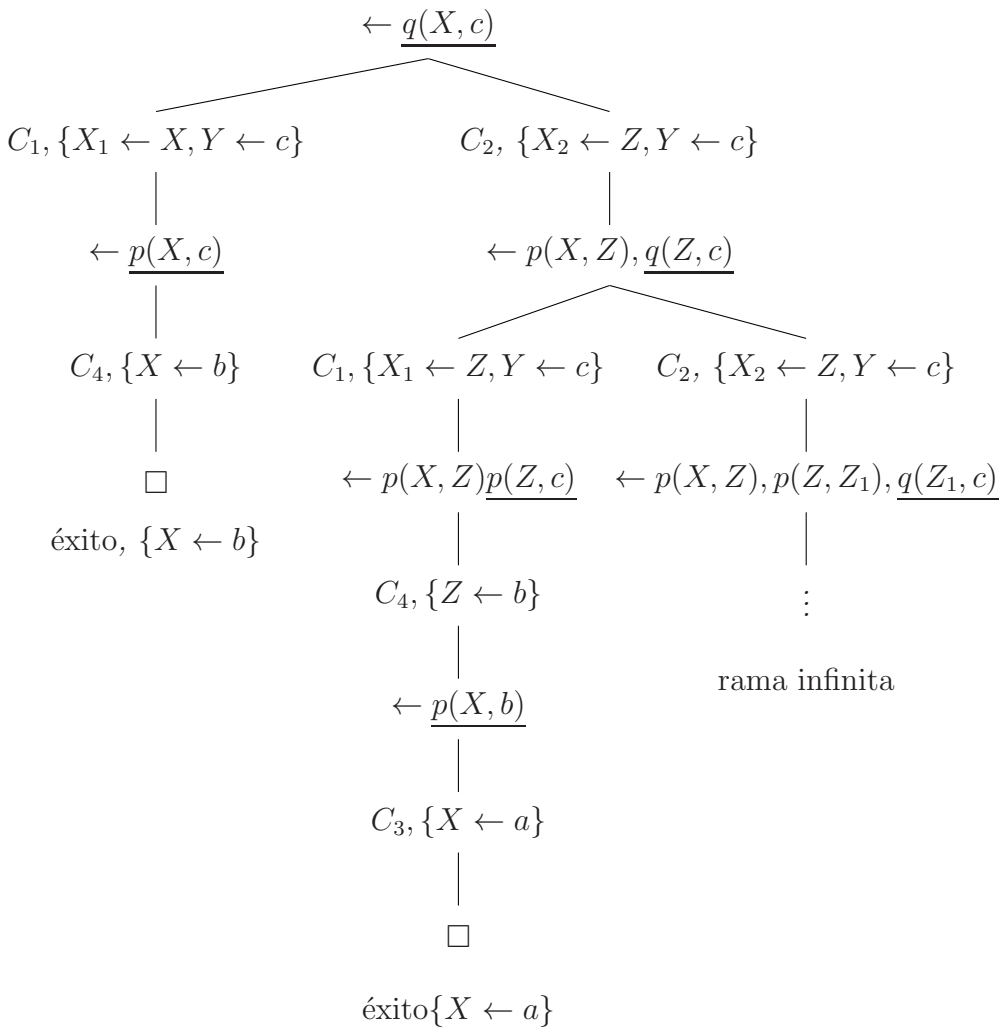
y tomemos como objetivo $\leftarrow q(X, c)$.

Primero construimos el árbol SLD que se corresponde con la regla de computación que elige el átomo más a la izquierda y la regla de ordenación que elige la primera cláusula posible (es decir, la que está más arriba). Es decir, usamos las reglas de computación y de ordenación de Prolog.



Cada arista del árbol está etiquetada con la cláusula del programa con la que hacemos resolución y la correspondiente umg. Además, subrayamos el átomo de la clásula objetivo escogido. Por otro lado, no completamos el árbol: en la rama más a la izquierda se repite parte del árbol ya construido, por lo que la marcamos como rama de fallo.

Ahora, construimos el árbol cambiando la regla de computación: elegimos el átomo en la cláusula objetivo más a la derecha



El Teorema sobre la completitud de la resolución SLD establece que la resolución SLD es completa independientemente de la regla de computación, pero el Teorema sólo dice que una tal refutación existe. La elección de la regla de búsqueda en el árbol determinará si la refutación se encuentra o no. El uso de la búsqueda en profundidad garantiza que siempre se encuentra una rama de éxito si existe. Por otro lado, la búsqueda en profundidad puede hacer que se escoja una derivación infinita, por lo que podríamos no

encontrar la refutación existente (si elegimos en el segundo caso del ejemplo anterior como regla de ordenación la de elegir el que esté más abajo en la lista, nos encontraríamos en esa situación). Sin embargo, la información del árbol que se debe almacenar crece exponencialmente con la profundidad, por lo que es impracticable. Es esta la razón por la Prolog se utiliza búsqueda en profundidad con backtracking.

Ejemplo 3.13 *1. Sea P el programa lógico dado por:*

$$C_1 = p(X,Y) \leftarrow q(Z,X), r(Z,V), q(V,Y).$$
$$C_2 = p(X,Y) \leftarrow q(Z,Y), r(Z,V), q(V,X).$$

$$C_3 = q(a,c). \qquad C_4 = q(a,d). \qquad C_5 = r(a,b).$$
$$C_6 = r(b,a). \qquad C_7 = q(b,e).$$

Computar las respuestas correctas para la pregunta $\leftarrow p(c,Y)$ usando resolución SLD a la Prolog.

2. Las listas se pueden codificar, empleando el Cálculo de Predicados, mediante una función . de aridad 2, y una constante nil, la lista vacía, de manera que: una lista con un elemento se representa mediante $.(a, nil)$, con dos elementos $.(a, .(b, nil)), \ldots$ y así sucesivamente. Definir un procedimiento lógico append de aridad 3 que tome valor cierto si el tercer elemento es la concatenación de los dos primeros. Una vez definido, calcular todas las respuestas correctas a

$$\leftarrow append(.(a, .(b, nil)), .(c, nil), Z)$$

usando resolución SLD a la Prolog.

3. Con las mismas definiciones que en el ejemplo anterior, calcular todas las respuestas correctas a

$$\leftarrow append(Y, Z, .(a, .(b, .(c, nil))))$$

usando resolución SLD a la Prolog.

Capítulo 4

$\lambda-$cálculo

Nota Preliminar. Las notas de este Capítulo están tomadas, en su mayor parte, de L.C. Paulson: "Foundations of Functional Programming". Las notas de Paulson se encuentran en

`http://www.cl.cam.ac.uk/~lp15/papers/Notes/Founds-FP.pdf`

4.1. λ-términos

Definición 4.1 *Los* **términos** *del $\lambda-$cálculo se construyen recursivamente a partir de un conjunto de variables $x, y, z, \ldots$ y pueden tener una de las siguientes formas*

x	*variable*
$(\lambda x.M)$	*abstracción, donde M es un término*
(MN)	*aplicación, donde M y N son términos*

Por lo tanto, usamos letras mayúsculas $L, M, N, \ldots$ para designar términos. Si dos términos son idénticos, se escribe $M \equiv N$. La igualdad entre $\lambda-$términos es algo más complicada de definir y se tratará más adelante.
En un término de la forma $(\lambda x.M)$, la variable x se dice ligada y a M se le dice cuerpo. Toda ocurrencia de x en M está ligada por la abstracción. Una ocurrencia de una varible es *libre* si no está ligada por ninguna abstracción. Por ejemplo, x está ligada e y es libre en el término $(\lambda z.(\lambda x.(yx)))$.

La abstracción $(\lambda x.M)$ representa a la función f tal que $f(x) = M$. Por lo tanto, la aplicación de f a N debería corresponderse con el resultado de sustituir por N todas las ocurrencias libres de x en M.

Ejemplos típicos de de abstracciones son:

$(\lambda x.x)$ La función *identidad*. Se denota **I**.

$(\lambda y.x)$ Una función *constante*.

El concepto de sustitución requiere explicación.

Definición 4.2 *Sea M un $\lambda-$término. El conjunto de variables ligadas de M, denotado por $BV(M)$ está dado por:*

$$\begin{aligned} BV(x) &= \emptyset \\ BV(\lambda x.M) &= BV(M) \bigcup \{x\} \\ BV(MN) &= BV(M) \bigcup BV(N) \end{aligned}$$

Definición 4.3 *Sea M un $\lambda-$término. El conjunto de variables libres de M, denotado por $FV(M)$ está dado por:*

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) \setminus \{x\} \\ FV(MN) &= FV(M) \bigcup FV(N) \end{aligned}$$

Definición 4.4 (Sustitución) *Sea M un $\lambda-$término. La sustitución por L de todas las apariciones libres de y en M, denotada por $M[L/y]$, está definida por:*

$$x[L/y] \equiv \begin{cases} L & \text{si } x \equiv y \\ x & \text{en caso contrario.} \end{cases}$$

$$((\lambda x.M)[L/y]) \equiv \begin{cases} (\lambda x.M) & \text{si } x \equiv y \\ (\lambda x.M[L/y]) & \text{en caso contrario.} \end{cases}$$

$$(MN)[L/y] \equiv (M[L/y]N[L/y])$$

Nota 4.5 *Las notaciones introducidas en las definiciones precedentes no forman parte del lenguaje del $\lambda-$cálculo. Pertenecen al metalenguaje: nos permiten decir cosas sobre el $\lambda-$cálculo.*

La sustitución no debe "inmiscuirse" en la ligazón de las variables. Consideremos, por ejemplo, el término $(\lambda x.(\lambda y.x))$. Debería representar la función que, aplicada a un argumento N, devuelve la función constante $(\lambda y.N)$. Sin embargo, esto no ocurre si $N \equiv y$. En ese caso, el reemplazar x por y en $(\lambda y.x)$, $(\lambda y.x)[y/x]]$, da lugar a la función identidad. La ocurrencia libre de x en la abstracción se transforma en una ocurrencia ligada de y. Si se permitiese esto, como veremos más adelante (Ejemplo 4.7), el λ-cálculo sería inconsistente. La sutitución $M[N/x]$ es correcta siempre y cuando las variables ligadas de M tengan intersección vacía con las variables libres de N, es decir,

$$BV(M) \bigcap FV(N) = \emptyset.$$

Obsérvese que podemos garantizar que esta condición se verifica renombrando las variables.

4.1.1. Conversión y normalización

La idea de que las $\lambda-$abstracciones representan funciones se expresa formalmente a través de reglas de conversión que las manipulan. Existen tres tipos de conversiones: α, β y η conversiones.

La $\alpha-$conversión expresa el hecho de que el nombre de las variables es irrelevante:

$$(\lambda x.M) \rightarrow_\alpha (\lambda y.M[y/x]).$$

Esta conversión es válida siempre y cuando y no aparezca en M. Generalmente, se ignora la diferencia entre términos que se pueden hacer idénticos mediante $\alpha-$conversiones. Por ejemplo, $(\lambda x.(xz)) \equiv (\lambda y.(yz))$, es decir, ambos términos se consideran idénticos.

Las $\beta-$ conversiones tratan de la sustitución:

$$((\lambda x.M)N) \rightarrow_\beta M[N/x].$$

Una $\beta-$conversión sustituye el argumento, N, en el cuerpo de la abstracción, M. Esta conversión sólo es correcta, naturalmente, si $BV(M) \bigcap FV(N) = \emptyset$. Por ejemplo,

$$((\lambda z.(zy))(\lambda x.x)) \rightarrow_\beta ((\lambda x.x)y) \rightarrow_\beta y.$$

Finalmente, la $\eta-$conversión es $(\lambda x.Mx) \rightarrow_\eta M$. Esta reducción funciona si $x \notin FV(M)$, es decir, cuando M no depende de x. Esta regla codifica

la siguiente propiedad de las funciones: dos funciones son iguales si devuelven iguales resultados para iguales argumentos. Es claro que $(\lambda x.Mx)$ y M verifican esta propiedad.

Diremos que N es una reducción de M, denotado $M \to N$, si $M \to_\beta N$ o $M \to_\eta N$. Esta reducción puede consistir en la aplicación de una conversión a un subtérmino de M para llegar a N. Necesitamos, pues, reglas de inferencia para las reducciones:

$$\frac{M \to M'}{(\lambda x.M) \to (\lambda x.M')} \quad \frac{M \to M'}{(MN) \to (M'N)} \quad \frac{M \to M'}{(NM) \to (NM')}$$

Si un término no admite reducción se dice que está en *forma normal*. Así, los términos $(\lambda xy.x)$ o $((xy)z)$ están forma normal. *Normalizar* un término significa aplicarle sucesivas reducciones hasta llegar a forma normal. Un término posee forma normal si puede reducirse a un término en formal normal. Por ejemplo, $((\lambda x.x)y)$ no está en forma normal, pero una $\beta-$conversión lo convierte en su forma normal y.

No todo término admite forma normal. El término habitualmente denotado por Ω, $((\lambda x.xx)(\lambda x.xx))$ se reduce a sí mismo mediante una $\beta-$conversión. Por lo tanto, no admite forma normal.

4.1.2. Funciones de Curry

En el $\lambda-$cálculo sólo hay funciones de una variable. De todas formas, las funciones multivariadas pueden expresarse mediante una función cuyo resultado es otra función. Consideremos, por ejemplo, que L es un término en el que sólo aparecen libres las variables x e y y queremos formalizar la función de dos variables $f(x, y) = L$. La abstracción $(\lambda y.L)$ contiene a x como variable libre. La abstracción $(\lambda x.(\lambda y.L))$, que no contiene ni a x ni a y como variables libres, representa la función dada. Aplicada a argumentos M y N se tiene:

$$(((\lambda x.(\lambda y.L))M)N) \to_\beta ((\lambda y.L[M/x])N) \to_\beta (L[M/x][N/y])$$

Por supuesto, en la reducción anterior se ha de respetar la condición que hace correctas las $\beta-$conversiones.

Esta técnica se conoce como *currificación* por Haskell B. Curry. Una función construida usando lambdas anidados se llama *currificada*. Claramente, la estrategia funciona para un número arbitrario de argumentos. Este tipo de funciones son claves en la programación funcional: aplicadas a unos pocos de sus primeros argumentos, devuelven funciones que pueden ser útiles por sí mismas.

Sin embargo, cuando hay muchos argumentos, la escritura de estas funciones puede ser muy complicada. Para simplificar escribiremos:

$$(\lambda x_1.(\lambda x_2 \ldots (\lambda x_n.M)\ldots)) \quad \text{como} \quad (\lambda x_1 x_2 \ldots x_n.M) \quad \text{y}$$

$$(\ldots(M_1 M_2)\ldots) \quad \text{como} \quad (M_1 M_2 \ldots M_n)$$

Asimismo, se eliminan los paréntesis más exteriores y los que delimitan el cuerpo de una abstracción. Siguiendo estas reglas,

$$(\lambda x(x(\lambda y.(yx)))) \text{ puede escribirse como } \lambda x.x(\lambda y.yx).$$

Es importante entender el uso de los paréntesis. La reducción

$$\lambda z.(\lambda x.M)N \to_\beta \lambda z.M[N/x]$$

es correcta pero un término similar tal que $\lambda z.z(\lambda x.M)N$ no admite más reducciones que las que se apliquen a M y N ya que la abstracción no se aplica a nada.
Por otro lado, conviene notar que $\lambda x.MN$ es abreviatura de $\lambda x.(MN)$ y no de $(\lambda x.M)N$. Del mismo modo, xyz es abreviatura de $(xy)z$ y no de $x(yz)$.

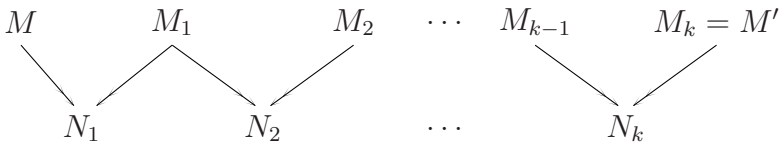
4.2. Igualdad y normalización

El $\lambda-$cálculo es una teoría de ecuaciones: consiste en reglas que permiten probar que dos términos son iguales. La propiedad esencial es que dos términos son iguales si se pueden reducir a un mismo término. Veamos esto con más detenimiento.
Tal cual lo hemos definido, $M \to N$ significa que M reduce a N mediante una única reducción. Sin embargo, hemos visto ejemplos en lo que hemos utilizado varios pasos de reducción. Se escribe $M \twoheadrightarrow N$ si

$$M \to M_1 \to M_2 \to \ldots \to M_k \equiv N \ \ (k \geq 0)$$

Por ejemplo, $((\lambda z.(zy))(\lambda x.x)) \twoheadrightarrow y$. También se utiliza la notación $\to^*$

Informalmente, $M = M'$ si M puede transformarse mediante ninguna o un número finito de reducciones u expansiones en M'. (Una *expansión* es lo inverso de la reducción). Gráficamente:



Por lo tanto, si $M \twoheadrightarrow L$ y $M' \twoheadrightarrow L$, $M = M'$. Por ejemplo, se verifica que $a((\lambda y.by)c) = (\lambda x.ax)(bc)$ ya que ambos términos se reducen a la forma normal $a(bc)$. Nótese que $=$ es la menor relación de equivalencia que contiene a $\rightarrow$.
El que dos términos son iguales, intuitivamente, debe significar que tienen el mismo valor. La igualdad, tal cual está definida, satisface las propiedades esperables. En primer lugar, es una relación de equivalencia:

$$M = M \quad \frac{M = N}{N = M} \quad \frac{L = M \;\; M = N}{L = N}$$

En segundo lugar, la igualdad respeta las leyes de construcción de $\lambda-$términos:

$$\frac{M = M'}{(\lambda x.M) = (\lambda x.M')} \quad \frac{M = M'}{(MN) = (M'N)} \quad \frac{M = M'}{(LM) = (LM')}$$

Definición 4.6 *La igualdad de $\lambda-$términos es la menor relación que satisface las seis propiedades anteriores.*

Una vez introducido el concepto de igualdad de términos estamos en condiciones de mostrar cómo colapsaría todo el $\lambda-$cálculo si se permitiera que la sustitución transformara variables libres en ligadas.

Ejemplo 4.7 *Probemos la siguiente falacia: para cualesquiera términos M y N, $M = N$.*
Sea $F \equiv \lambda xy.yx$. Entonces,

$$FMN \equiv ((\lambda x.(\lambda y.yx))M)N \rightarrow (\lambda y.yM)N \rightarrow NM.$$

En particular, $Fyx = xy$. Pero,

$$Fyx \equiv ((\lambda x.(\lambda y.yx))y)x \rightarrow (\lambda y.yy)x = xx.$$

Así pues, $xy = xx$. Si esa conclusión es cierta, la igualdad $\lambda xy.xy = \lambda xy.xx$ es también cierta. Por lo tanto, $MN = (\lambda xy.xy)MN = (\lambda xy.xx)MN =$

MM. Así pues, $MN = MM$ para cualquier par de términos M y N. En consecuencia, $N = (\lambda xy.y)MN = (\lambda xy.y)MM = M$. ■

El problema de la argumentación anterior viene de la $\beta-$conversión $(\lambda x.(\lambda y.yx))y \rightarrow (\lambda y.yy)$. Se tiene que y es una variable libre en el $\lambda-$término y, pero es ligada en la abstracción. Estamos violando la regla que hace que la $\beta-$conversión sea correcta.

Teorema de Church-Rosser

Hemos establecido anteriormente que si dos términos reducen a un tercero, han de ser iguales. El Teorema de Church-Rosser establece el recíproco.

Teorema 4.8 (Church-Rosser) *Si $M = N$, existe L tal que $M \twoheadrightarrow L$ y $N \twoheadrightarrow L$.*

Dos consecuencias de especial relevancia del Teorema son las siguientes:

- Si $M = N$ y N está en forma normal, entonces $M \twoheadrightarrow N$
- Si $M = N$ y ambos términos están en forma normal, entonces $M \equiv N$. De esto podemos deducir, por ejemplo, que $xy \neq xx$ o que $\lambda xy.x \neq \lambda xy.y$.

Una teoría de ecuaciones se dice inconsistente si cualquier ecuación se puede probar. El Teorema de Church-Rosser garantiza la consistencia del $\lambda-$cálculo.

Reducción en orden normal

Aunque diferentes sucesiones de reducciones no pueden llevar a formas normales distintas, pueden tener resultados completamente distintos: una puede terminar mientras que la otra no acaba nunca. En general, si M posee forma normal y admite una reducción infinta es porque contiene un subtérmino que no admite forma normal. Consideremos, por ejemplo $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$. La reducción

$$(\lambda y.a)\Omega \rightarrow a$$

llega a forma normal. Esto se corresponde a una evaluación por nombre. Es decir, no se reduce el argumento, sino que se sustituye tal cual en el cuerpo de la abstracción. Si en este caso intentamos normalizar el argumento, como éste es Ω, se genera una sucesión de reducciones infinita. Evaluar el

argumento antes de sustituir en el cuerpo de la abstracción se corresponde con la evaluación por valor.

La estrategia de reducción en *orden normal* consiste en, a cada paso, ejecutar la $\beta-$reducción más exterior y más a la izquierda. (Las eventuales $\eta-$reducciones se dejan para el final). La más a la izquierda significa, por ejemplo, reducir L antes de N en LN. La más exterior significa, por ejemplo, reducir $(\lambda x.M)N$ antes de reducir M o N. La reducción en orden normal se corresponde con la evaluación por nombre. Se puede probar que si existe forma normal y se aplica la estrategia de reducción en orden normal siempre se llega a la misma. Sin embargo, en la práctica computacional, es muy ineficiente. Supongamos que tenemos una codificación de los naturales y se define la función cuadrado $\mathbf{sqr} \equiv \lambda n.\mathbf{mult}\ nn$.[1] Se tiene:

$$\mathbf{sqr}(\mathbf{sqr}\ N) \to \mathbf{mult}(\mathbf{sqr}N)(\mathbf{sqr}\ N) \to \mathbf{mult}(\mathbf{mult}\ N\ N)(\mathbf{mult}\ N\ N)$$

y habríamos tenido que evaluar cuatro copias del término N. Si hubiésemos seguido la evaluación por valor, sólo lo tendríamos que haber hecho una sola vez pero, como hemos visto, esta estrategia puede dar lugar a que el proceso de reducción no termine.

La *evaluación perezosa* nunca evalúa un argumento más de una vez. Un argumento no se evalúa hasta que no se necesita para producir una respuesta infinita. Incluso en ese caso, puede evaluarse sólo en la medida de lo necesario. En consecuencia, admite listas infinitas.

4.3. Estructuras de datos en λ−cálculo

4.3.1. Booleanos

Cualquier codificación de los booleanos debe definir los términos **true**, **false** e **if** de forma que satisfagan (para cualesquiera M y N)

$$\begin{array}{rcl} \mathbf{if}\ \mathbf{true}\ MN & = & \mathrm{M} \\ \mathbf{if}\ \mathbf{false}\ MN & = & \mathrm{N}. \end{array}$$

La codificación usual es la siguiente:

[1] Suponemos definida la función multiplicación **mult**. Por otro lado, las palabras que sirven para indicar términos como **mult** se escriben en negrita para distinguirlas de las sucesiones de variables separadas como sqr.

$$\begin{array}{lcl} \textbf{true} & \equiv & \lambda xy.x \\ \textbf{false} & \equiv & \lambda xy.y \\ \textbf{if} & \equiv & \lambda pxy.pxy. \end{array}$$

El Teorema de Church-Rosser garantiza que **true** $\neq$ **false**. Es obvio que

$$\begin{array}{lcl} \textbf{true}\ MN & \equiv & (\lambda xy.x)MN \twoheadrightarrow M \\ \textbf{false}\ MN & \equiv & (\lambda xy.y)MN \twoheadrightarrow N \end{array}$$

Ambas reducciones funcionan para cualesquiera términos M y N posean, o no, forma normal. Al igual que las anteriores, las siguientes reducciones funcionan para cualesquiera términos M y N:

$$\begin{array}{lcl} \textbf{if true}\ MN & \twoheadrightarrow & M \\ \textbf{if false}\ MN & \twoheadrightarrow & N \end{array}$$

Una vez definidos **true**, **false** e **if**, podemos definir los conectivos usuales del Cálculo Proposicional.

$$\begin{array}{lcl} \textbf{and} & \equiv & \lambda pq.\ \textbf{if}\ p\ q\ \textbf{false} \\ \textbf{or} & \equiv & \lambda pq.\ \textbf{if}\ p\ \textbf{true}\ q \\ \textbf{not} & \equiv & \lambda p.\ \textbf{if}\ p\ \textbf{false}\ \textbf{true} \end{array}$$

Es fácil ver que los conectivos así definidos se comportan como es de esperar. Por otro lado, a partir de las codificaciones anteriores, como es bien sabido, se pueden codificar el resto de conectivos.

4.3.2. Pares ordenados

A partir de los $\lambda-$términos **true** y **false**, podemos codificar los pares ordenados. La función **pair** construye pares y **fst** y **snd** definen las proyecciones:

$$\begin{array}{lcl} \textbf{pair} & \equiv & \lambda xyf.fxy \\ \textbf{fst} & \equiv & \lambda p.p\ \textbf{true} \\ \textbf{snd} & \equiv & \lambda p.p\ \textbf{false} \end{array}$$

Es obvio que **pair** $MN \twoheadrightarrow \lambda f.fMN$, es decir, "empaqueta" M y N. De esta forma, un par así construido puede aplicarse a un término de la forma $\lambda xy.L$, produciendo $L[M/x][N/y]$. Así pues, cada par es su propio "desempaquetador". Las proyecciones funcionan de esta forma:

$$\begin{array}{lcl} \textbf{fst}\ (\textbf{pair}\ MN) & \twoheadrightarrow & \textbf{fst}\ (\lambda f.fMN) \\ & \to & (\lambda f.fMN)\ \textbf{true} \\ & \to & \textbf{true}\ MN \\ & \twoheadrightarrow & M \end{array}$$

Una reducción similar muestra que **snd** (**pair** MN) $=N$.

4.3.3. Números naturales

La codificación que daremos de los naturales es la original desarrollada por Church. Otras codificaciones son usuales hoy en día pero los números de Church contienen la estructura de control junto a la estructura de datos. Se define:

$$\begin{array}{lcl} \underline{0} & \equiv & \lambda fx.x \\ \underline{1} & \equiv & \lambda fx.fx \\ \underline{2} & \equiv & \lambda fx.f(fx) \\ \vdots & \vdots & \vdots \\ \underline{n} & \equiv & \lambda fx.\underbrace{f(\cdots(f}_{n \text{ veces}} x)\cdots) \end{array}$$

De esta forma, para todo $n \geq 0$, el número de Church $\underline{n}$ es la función que envía f a f^n. Cada número es un operador iterativo.

La codificación anterior permite definir de forma sencilla suma, producto y exponenciación:

$$\begin{array}{lcl} \textbf{add} & \equiv & \lambda mnfx.mf(nfx) \\ \textbf{mult} & \equiv & \lambda mnfx.m(nf)x \\ \textbf{expt} & \equiv & \lambda mn.nm \end{array}$$

Veamos que **add** y **mult** hacen lo esperado. Dejamos como ejercicio probar que **expt** $\underline{m}\ \underline{n} = \underline{m^n}$.

La adición es sencilla:

$$\begin{array}{lcl} \textbf{add}\ \underline{m}\ \underline{n} & \twoheadrightarrow & \lambda fx.\underline{m}\ f(\underline{n}\ fx) \\ & \twoheadrightarrow & \lambda fx.f^m(f^n)x \\ & \equiv & \lambda fx.f^{n+m}x \\ & \equiv & \underline{m+n} \end{array}$$

Algo más complicado es el producto:

$$\begin{array}{lcl} \textbf{mult}\ \underline{m}\ \underline{n} & \twoheadrightarrow & \lambda fx.\underline{m}\ (\underline{n}\ f)x \\ & \twoheadrightarrow & \lambda fx.(\underline{n}f)^m x \\ & \twoheadrightarrow & \lambda fx.(f^n)^m x \\ & \equiv & \lambda fx.f^{n\times m}x \\ & \equiv & \underline{m\times n} \end{array}$$

Como resulta fácil de ver, **add**, **mult** y **expt** sólo funcionan aplicadas a los numerales de Curch y no aplicadas a cualquier otro término.

4.3.4. Operaciones básicas en los numerales de Church

Las operaciones definidas en la Subsección anterior no son suficientes para trabajar con los números naturales. Por ejemplo, ¿qué ocurre con la substracción? Empecemos con dos funciones sencillas: la función sucesor y un test de nulidad.

$$\begin{array}{lcl} \textbf{suc} & \equiv & \lambda nfx.f(nfx) \\ \textbf{iszero} & \equiv & \lambda n.n(\lambda x.\ \textbf{false})\ \textbf{true} \end{array}$$

Es fácil ver que se verifican las siguientes reducciones para los numerales de Church $\underline{n}$:

$$\begin{array}{lcl} \textbf{suc}\ \underline{n} & \twoheadrightarrow & \underline{n+1} \\ \textbf{iszero}\ \underline{0} & \twoheadrightarrow & \textbf{true} \\ \textbf{iszero}\ \underline{n+1} & \twoheadrightarrow & \textbf{false} \end{array}$$

Más difícil es definir el predecesor ya que los numerales de Church están definidos iterativamente. Dados f y x, debemos encontrar g e y tales que $g^{n+1}y$ compute $f^n x$. Una g adecuada es $g(x,z) = (f(x), x)$. De este modo,

$$g^{n+1}(x,x) = (f^{n+1}(x), f^n(x)).$$

Ahora, podemos definir la función g como sigue:

$$\textbf{prefn} \equiv \lambda fp.\ \textbf{pair}\ (f\ (\textbf{fst}\ \ p))\ (\textbf{fst}\ p)$$

Una vez definida esta función, es fácil definir el predecesor y, en consecuencia, la substracción:

$$\begin{array}{lcl} \textbf{pre} & \equiv & \lambda nfx.\ \textbf{snd}\ (n\ (\textbf{prefn}\ f)(\textbf{pair}\ xx)) \\ \textbf{sub} & \equiv & \lambda mn.n\ \textbf{pre}\ m \end{array}$$

Se tiene $\textbf{pre}\ \underline{n+1} \twoheadrightarrow \underline{n}$ y $\textbf{pre}\ \underline{0} \twoheadrightarrow \underline{0}$

4.3.5. Listas

Representaremos las listas tal cual lo hace Lisp, es decir, a través de pares ordenados. La lista $[x_1, x_2, \ldots, x_n]$ se representará por $x_1 :: x_2 :: \ldots :: nil$. Para mantener las operaciones lo más simples posibles, emplearemos dos niveles. Cada construcción de la forma $x :: y$ será representada por (**false**,(x,y)). La primera coordenada servirá para distinguir la lista vacía del resto. En consecuencia, la lista vacía, **nil**, debería representarse por un par cuya primera componente fuera **true**. Sin embargo, una codificación más sencilla funciona. He aquí la codificación de las listas:

$$\begin{array}{lcl} \textbf{nil} & \equiv & \lambda z.z \\ \textbf{cons} & \equiv & \lambda xy.\ \textbf{pair false}(\ \textbf{pair}\ xy) \\ \textbf{null} & \equiv & \textbf{fst} \\ \textbf{hd} & \equiv & \lambda z.\ \textbf{fst}\ (\ \textbf{snd}\ z) \\ \textbf{tl} & \equiv & \lambda z.\ \textbf{snd}\ (\ \textbf{snd}\ z) \end{array}$$

Las funciones así definidas se comportan como deben:

$$\begin{array}{lcl} \textbf{null nil} & \twoheadrightarrow & \textbf{true} \\ \textbf{null}\ (\textbf{cons}\ MN) & \twoheadrightarrow & \textbf{false} \\ \textbf{hd}\ (\textbf{cons}\ MN) & \twoheadrightarrow & M \\ \textbf{tl}\ (\textbf{cons}\ MN) & \twoheadrightarrow & N \end{array}$$

La reducción **null nil**$\twoheadrightarrow$ **true** se produce por casualidad, mientras que el resto de leyes funcionan por la forma en la que hemos definido las operaciones sobre los pares ordenados. Por otro lado, conviene recdordar que las leyes referentes a **hd** y **tl** funcionan para cualesquiera términos, independientemente de que posean forma normal o no. Por lo tanto, **hd** y **tl** son funciones perezosas: no evalúan sus argumentos.

4.4. Recursividad en el λ−cálculo

4.4.1. Recursividad con puntos fijos

La recursividad puede derivarse en el λ−cálculo. Se puede, por lo tanto, modelizar las definiciones recursivas de funciones, incluso de aquellas que no terminan para algunos (o todos) sus argumentos. La codificación de la recursión es independiente de los detalles de la definición recursiva y de la representación de las estructuras de datos.

El secreto reside en el uso de un *combinador de punto fijo*, es decir, un término **Y** tal que $\mathbf{Y}\ F = F(\mathbf{Y}\ F)$ para todo término F. Esta terminología proviene del concepto de punto fijo para funciones: X es punto fijo de F si $F(X) = X$ (en este caso, $X = \mathbf{Y}\ F$). Por otro lado, un combinador es un término sin variables libres (también llamado término cerrado). Para codificar la recursión, F representa el cuerpo de la definición recursiva. La ley $\mathbf{Y}\ F = F(\mathbf{Y}\ F)$ permite desplegar F tantas veces como sea necesario.

Consideremos como aplicación la codificación del factorial, la concatenación de listas y la lista infinita $[0, 0, \ldots]$ en el λ−cálculo. Se deben verificar las ecuaciones:

$$\begin{array}{lcl} \textbf{fact}\ N & = & \textbf{if}\ (\textbf{iszero}\ N)\ \underline{1}\ (\textbf{mult}\ N\ (\textbf{fact}(\textbf{pre}\ N))) \\ \textbf{append}\ ZW & = & \textbf{if}\ (\textbf{null}\ Z)\ W\ (\textbf{cons}\ (\textbf{hd}\ Z)\ (\textbf{append}(\textbf{tl}\ Z)\ W)) \\ \textbf{zeroes} & = & \textbf{cons}\ \underline{0}\ \textbf{zeroes} \end{array}$$

Para conseguir que se verifiquen las ecuaciones anteriores basta con poner

$$\begin{array}{lcl} \textbf{fact} & \equiv & \mathbf{Y}\ (\lambda gn.\ \textbf{if}\ (\textbf{iszero}\ n)\ \underline{1}\ (\textbf{mult}\ n\ (g(\textbf{pre}\ N)))) \\ \textbf{append} & \equiv & \mathbf{Y}(\lambda gzw.\textbf{if}\ (\textbf{null}\ z)\ w\ (\textbf{cons}\ (\textbf{hd}\ z)\ (g(\textbf{tl}\ Z)\ W))) \\ \textbf{zeroes} & \equiv & \mathbf{Y}\ (\lambda g.\textbf{cons}\ \underline{0}\ g) \end{array}$$

En cada definición, la llamada recursiva se reemplaza por la variable g en $\mathbf{Y}(\lambda g.\ldots)$. Veamos como funciona en el caso de **zeroes**; el resto de los casos es similar:

$$\begin{array}{rcl} \textbf{zeroes} & \equiv & \mathbf{Y}\ (\lambda g.\textbf{cons}\ \underline{0}\ g) \\ & = & (\lambda g.\textbf{cons}\ \underline{0}\ g)\ \mathbf{Y}\ (\lambda g.\textbf{cons}\ \underline{0}\ g)) \\ & = & (\lambda g.\textbf{cons}\ \underline{0}\ g)\ \textbf{zeroes} \\ & \to & \textbf{cons}\ \underline{0}\ \textbf{zeroes} \end{array}$$

4.4.2. Combinadores de punto fijo

El combinador **Y** fue descubierto por Haskell B. Curry. Se define como

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Es obvio que es un combinador. Veamos que posee la propiedad de punto fijo:

$$\begin{array}{rcl} \mathbf{Y}\ F & \to & (\lambda x.F(xx))(\lambda x.F(xx)) \\ & \to & F((\lambda x.F(xx))(\lambda x.F(xx))) \\ & = & F(\mathbf{Y}\ F) \end{array}$$

Hemos aplicado dos $\beta-$reducciones seguidas de una $\beta-$expansión. No hay reducción posible $\mathbf{Y}\ F \twoheadrightarrow F(\mathbf{Y}\ F)$. Veremos posteriormente por qué este combinador se denomina *combinador paradójico.*

Este combinador no es el único conocido. Por ejemplo, el combinador Θ descubierto por Turing está dado por:

$$\begin{array}{rcl} A & \equiv & \lambda xy.y(xxy) \\ \Theta & \equiv & AA \end{array}$$

En este caso, sí que se tiene la reducción $\Theta\ F \twoheadrightarrow F(\Theta\ F)$:

$$\Theta\ F \equiv AAF \twoheadrightarrow F(AAF) \equiv F(\Theta\ F)$$

Una receta para construir combinadores de punto fijo es la siguiente. Pongamos

$$\Gamma = \gamma\gamma\cdots\gamma \ \ (n \text{ veces con } n \geq 2).$$

Sea ahora un alfabeto $\{a_1, \ldots, a_{n-1}\}$ con $n-1$ letras distintas y w una palabra cualquiera de longitud n en ese alfabeto. Es fácil ver que si tomamos $\gamma \equiv \lambda a_1 a_2 \ldots a_{n-1} f.f(wf)$, Γ es un combinador de punto fijo. De esta forma, Klop descubrió el combinador de punto fijo Y_{fpc} que declara que lo es[2]:.

$$L \equiv \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator)$$

$$Y_{fpc} \equiv LLLLLLLLLLLLLLLLLLLLLLLLLL$$

Finalizamos explicando por qué el combinador de punto fijo **Y** se llama combinador paradójico. Alonzo Church creó el λ−cálculo para formalizar una nueva teoría de conjuntos que, entre otras cosas, evitase la *paradoja de Russell*. Esta paradoja, tal cual la describió Russell en 1901 es la siguiente:

> *From now the assemblage of classes which are not members of themselves. This is a class: is it a member of itself or not? If it is, it is one of those classes that are members of themselves, i.e. it is not a member of itself. If it is not, it is not one of the classes that are not members of themselves, i.e it is a member of itself. Thus the two hypothesis- that it is, and that is not, a member of itself-each implies its contradictory. This is a contradiction.*

Utilizando notación matemática al uso, la paradoja se escribe como sigue:

Sea $R = \{X : X \ \text{ conjunto y } \ X \notin X\}$. Se tiene que si $R \in R$, entonces, $R \notin R$; y si $R \notin R$, entonces $R \in R$.

En su teoría, Church codificaba los conjuntos por sus funciones características (es decir, como predicados). La prueba de pertenencia $M \in N$ se codificaba por la aplicación $N(M)$, que podría ser verdadera o falsa. La abstracción $\{x|P\}$ se codificaba por $\lambda x.P$, donde P era algún λ−término que expresaba una propiedad de x. Desgraciadamente para Church, la paradoja de Russel era derivable en su teoría: el conjunto R descrito por Russell se codifica por $\lambda x.$ **not** (xx), lo que implica $RR =$ **not** (RR), lo que es contradictorio en términos lógicos. Curry descubrió esta contradicción. La ecuación de punto fijo para **Y** se obtiene de $RR =$ **not** (RR), si reemplazamos **not** por un término arbitrario F.

[2]Esta construcción puede verse en J.W. Klop: "New fixed point combinators from old". `http://www.cs.ru.nl/barendregt60/essays/klop/art16_klop.pdf`

Capítulo 5

Semántica y verificación de programas

Un programa no es muy diferente de una fórmula lógica. Es una secuencia de símbolos que se construye en base a reglas sintácticas formales y cuyo significado se comprende mediante una interpretación adecuada de los elementos del lenguaje. En programación, los símbolos se llaman expresiones o comandos y la correspondiente interpretación es una ejecución en una máquina, más que una evaluación de la tabla de verdad. Normalmente, la sintaxis de los lenguajes de programación se especifa mediante el uso de sistemas formales, pero la semántica se suele especificar de manera informal. Ahora bien. Si la semántica se define de manera informal, no hay manera de determinar la validez o corrección de un programa y nos vemos reducidos a testar y depurar, lo que es poco fiable.

En este Capítulo, se presenta un sistema formal para especificar la semántica de un lenguaje de programación imperativo y se define, de forma precisa, el concepto de programa correcto. Se presenta, asimismo, un sistema axiomático que puede usarse para probar que un programa es correcto con respecto a su especificación formal.

Hay una diferencia esencial entre los programas y el cálculo de predicados. El cálculo de predicados se centra en interpretaciones arbitrarias y, en consecuencia, en las fórmulas válidas. La mayoría de programas, sin embargo, tratan con dominios numéricos u otros dominios predefinidos (cadenas, listas, árboles,...). Para razonar sobre los programas, debemos tomar los teoremas en esos dominios como axiomas. Así, la completitud de sistema axiomático para la verificación de programs depende de la teoría del dominio.

5.1. Semántica de lenguajes de programación imperativos

Una *expresión* en un lenguaje de programación puede verse como una función que transfoma el estado de computación. Si las variables $(\mathtt{x}, \mathtt{y})$ tienen los valores $(8, 7)$ en un estado s, el resultado de ejecutar la expresión $\mathtt{x} := 2 * \mathtt{y} + 1$ es el estado s' en el que $(x, y) = (15, 7)$.

Definición 5.1 *Si un programa usa n variables* $(\mathtt{x_1}, \ldots, \mathtt{x_n})$*, un estado s consiste en una n−tupla de de valores* $(x_1, \ldots, x_n)$*, donde* x_i *es el valor de la variable* $\mathtt{x_i}$*.*

Como queremos razonar dentro del marco del cálculo de predicados, debemos sustituir los conjuntos de estados por predicados.

Definición 5.2 *Sea U el conjunto de todas las n−tuplas en un cierto dominio y sea* $U' \subseteq U$*. El* **predicado característico** *de* U'*, denotado por* $P_{U'}(x_1, \ldots, x_n))$*, se define mediante:*

$$U' = \{(x_1, \ldots, x_n) \in U : P_{U'}(x_1, \ldots, x_n\}.$$

Ejemplo 5.3 *Sea S la expresión* `x:= 2*y+1`*. Esta expresión transforma estados en* $\{(x, y) \mid true\}$ *en estados en* $\{(x, y) \mid x = 2y + 1\}$*. Supongamos que el conjunto de estados iniciales es* $\{(x, y) \mid y \leq 3\}$*. Tras ejecutar S, el estado final está en* $\{(x, y) \mid (x \leq 7) \wedge (y \leq 3)\}$*. En este sentido, se dice que S transforma* $y \leq 3$ *en* $(x \leq 7) \wedge (y \leq 3)$*.*

La semántica de un lenguaje de programación se da especificando cómo cada expresión en el lenguaje transforma un estado inicial en un estado final. El mismo concepto sirve para definir la semántica de un programa, que está compuesto de expresiones del lenguaje.

Definición 5.4 *Un* **aserto** *o* **terna de Hoare** *es una terna* $\{p\}\ S\ \{q\}$*, donde S es un programa, y p y q son fórmulas del cálculo de predicados llamadas* **precondición** *y* **postcondición***, respectivamente.*
Un aserto es verdadero, y se denota por $\vDash \{p\}\ S\ \{q\}$*, si, y sólo si, siempre que S empiece en un estado que satisfaga p y la computación termine, ésta termina en un estado que satisface q.*
Si $\vDash \{p\}\ S\ \{q\}$*, se dice que S es* **parcialmente correcto** *con respecto a p y q.*

Ejemplo 5.5 $\vDash \{y \leq 3\}$ `x:= 2*y+1` $\{(x \leq 7) \wedge (y \leq 3)\}$.

Ejemplo 5.6 $\vDash \{F\}\ S\ q$ *para cualquier S y q. De manera similar,* $\vDash p\ S\ \{T\}$ *para cualquier p y S.*

5.1.1. Precondiciones más débiles

La formalización de la semántica de un lenguaje en términos de predicados se da estableciendo, para cada expresión y postcondición, la mínima precondición que garantiza la verdad de la postcondición bajo la hipótesis de la finalización de la computación de la expresión. En este caso, mínima quiere decir que es aquella precondición que es verificada por mayor número de estados. Por ejemplo, $p \equiv (y \leq 3)$ no es, desde luego, la única precondición para la que $\vDash p$ `x:= 2*y+1` $\{(x \leq 7) \wedge (y \leq 3)\}$. Por ejemplo, podríamos tomar $p \equiv (y = 0)$. No es difícil ver que la precondición $(y \leq 3)$ es la que nos da el mayor número de estados iniciales posibles para los que, acabada la ejecución de `x:= 2*y+1` se obtiene un estado que satisface $\{(x \leq 7) \wedge (y \leq 3)\}$.

Definición 5.7 *Una fórmula A se dice más débil que B si $B \rightarrow A$. Dado un conjunto de fórmulas $\{A_1, A_2, \ldots\}$, A_i es la fórmula más débil del conjunto si $A_j \rightarrow A_i$ para todo j.*

Siempre se puede fortalecer una premisa y debilitar una consecuencia. Por ejemplo, si $p \rightarrow q$, entonces $(p \wedge r) \rightarrow q$ y $p \rightarrow (q \vee r)$.

Definición 5.8 *Para un programa S y una fórmula q se define la* **precondición más débil** *para S y q, y se denota por $wp(S, q)$, a la fórmula más débil p de entre las que verifican $\vDash \{p\}\ S\ \{q\}$.*

Consecuencia inmediata de la definición es el siguiente lema:

Lema 5.9 *Se tiene*

$$\vDash \{p\}\ S\ \{q\} \text{ si, y sólo, si } \vDash p \rightarrow wp(S, q)$$

La precondición más débil depende tanto del programa S como de la postcondición q. Así pues, wp transforma predicados ya que, dado un fragmento de un programa, define una transformación de la postcondición en un predicado (la precondición más débil). Por lo tanto, en vez de formalizar un programa como una transformación de estados, lo haremos como una transformación de predicados: del predicado postcondición al predicado precondición. Esta formulación permite comenzar con la especificación del resultado del programa completo y trabajar hacia atrás para diseñar o verificar un programa que consiga el resultado fijado.

5.1.2. Semántica de un fragmento de un lenguaje imperativo

La teoría que estamos desarrollando la inició Hoare para el Pascal[1]. Sin embargo, se puede extender a cualquier lenguaje imperativo. Nos centraremos, para facilitar la exposición, en fragmentos de programas que sean composiciones de asignaciones y expresiones `si` y `mientras`.

Definición 5.10 $wp(\texttt{x:=t}, p(x)) = p(x)\{x \leftarrow t\}$

A primera vista, la semántica de la asignación parece extraña, pero debe entenderse en términos de transformación de predicados. Si la sustitución t por x hace $p(x)$ verdadera ahora, sigue adelante y realiza la asignación, tras lo que $p(x)$ será verdadero y x tendrá valor t.

Ejemplo 5.11 $wp(\texttt{y:=y-1}, y \geq 0) = (y - 1 \geq 0) \equiv (y \geq 1)$

Definición 5.12 $wp(S_1; S_2, q) = wp(S_1, wp(S_2, q))$

La precondición $wp(S_2, q)$ caracteriza el conjunto de estados tales que una ejecución de S_2 lleva a un estado en el que q es verdadero. Si la ejecución de S_1 lleva a uno de esos estados, entonces $S_1; S_2$ lleva a un estado cuya postcondición es q. Veamos un par de ejemplos de aplicación de la definición. El primero es muy sencillo. El segundo tiene un poco más de enjundia.

Ejemplo 5.13

$$\begin{array}{rcl} wp(\texttt{x:=x*x; y:= y+1}, x < y) & = & wp(\texttt{x:=x*x}, wp(\texttt{y:= y+1}, x < y) \\ & = & wp(\texttt{x:=x*x}, x < y + 1) \\ & = & x^2 < y + 1 \end{array}$$

Ejemplo 5.14

$$\begin{array}{l} wp(\texttt{x:=x+a; y:= y-1}, x = (b - y) \cdot a) \\ = wp(\texttt{x:=x+a}, wp(\texttt{y:= y-1}, x = (b - y) \cdot a) \\ = wp(\texttt{x:=x+a}, x = (b - y + 1) \cdot a) \\ = x + a = (b - y + 1) \cdot a \\ \equiv x = (b - y) \cdot a \end{array}$$

[1]El artículo esencial es C.A.R. Hoare, N. Wirth: An Axiomatic Definition of the Programming Language PASCAL. *Acta Informatica* 2, 335-355 (1973). En `http://www.springerlink.com/content/lk30hp0718778688/`

Dada la precondición $x = (b-y)\cdot a$, la expresión del ejemplo, entendida como transformador de predicados, no hace nada. A este tipo de predicados se les llama invariantes. En cualquier caso, la expresión sí que cambia los valores de las variable `x` *e* `y`*.*

Definición 5.15 *Un predicado I se dice que es un* **invariante** *de S si y sólo si $wp(S, I) = I$.*

Seguimos definiendo la semántica con las expresiones `si`:

Definición 5.16 *Las dos definiciones de condición más débil para expresiones* `si` *siguientes son equivalentes:*

$$wp(\texttt{si B entonces S1 si no S2, q}) = \\ = (B \to wp(\texttt{S1}, q)) \land (\neg B \to wp(\texttt{S2}, q))$$

$$wp(\texttt{si B entonces S1 si no S2}, q) = \\ = (B \land wp(\texttt{S1}, q)) \lor (\neg B \land wp(\texttt{S2}, q))$$

La primera de las definiciones es fácil de entender ya que el predicado B crea una partición del conjunto de estados en dos conjuntos disjuntos y las precondiciones están determinadas por las acciones de cada `Si` en su subconjunto. La definición equivalente se sigue de la equivalencia lógica:

$$(p \to p) \land (\neg p \to r) \equiv (p \land q) \lor (\neg p \land r)$$

Ejemplo 5.17

$$\begin{aligned} & wp(\texttt{si y=0 entonces x:=0 si no x:= y+1}, x = y) \\ & = (y = 0 \to wp(\texttt{x:=0}, x = y)) \land (y \neq 0 \to wp(\texttt{x:= y+1}, x = y)) \\ & = (y = 0 \to (y = 0)) \land (y \neq 0 \to (y + 1 = y)) \\ & \equiv T \land ((y \neq 0) \to F) \\ & \equiv \neg(y \neq 0) \\ & \equiv y = 0 \end{aligned}$$

Falta, según lo anunciado, establecer la semántica de los bucles `mientras`:

Definición 5.18 *Las dos definiciones de condición más débil para expresiones* `mientras` *siguientes son equivalentes:*

$$wp(\texttt{mientras B hacer S, q}) = \\ = (\neg B \to q) \land (B \to wp(\texttt{S; mientras B hacer S}, q))$$

$$wp(\texttt{mientras B hacer S, q}) = \\ = (\neg B \land q) \lor (B \land wp(\texttt{S; mientras B hacer S}, q))$$

La segunda definición se sigue de la primera aplicando la misma equivalencia que en la definición de condición más débil para las expresiones `if`. En cuanto a la primera se entiende a la luz de la ejecución de una expresión `mientras`:

- La expresión termina inmediatamente si la expresión booleana se evalúa falsa, en cuyo caso el estado no cambia por lo que precondición y postcondición son la misma.
- La expresión booleana se evalúa como cierta y causa la ejecución del cuerpo del bucle. Hasta el final de la ejecución la expresión `mientras` trata de establecer la postcondición.

Ejemplo 5.19 *Dada la definición recursiva de la condición más débil para una expresión* `mientras`*, no se puede establecer de manera constructiva. Sin embargo, veremos un ejemplo que ayuda a entender la situación. Llamemos* W *a*

```
mientras x>0 hacer x:=x-1.
```

En ese caso,

$$\begin{array}{l} wp(W, x = 0) \\ = (\neg(x > 0) \wedge (x = 0)) \vee ((x > 0) \wedge wp(\texttt{x:= x-1;}W, x = 0)) \\ \equiv (x = 0) \vee ((x > 0) \wedge wp(\texttt{x:= x-1;}W, x = 0)) \\ \equiv (x = 0) \vee ((x > 0) \wedge wp(W, x = 0)\{x \leftarrow x - 1\}). \end{array}$$

Ahora debemos realizar la sustitución $\{x \leftarrow x-1\}$ *en* $wp(W, x = 0)$*. Haciendo la sustitución y simplificando, se tiene:*

$$\begin{array}{l} wp(W, x = 0) \\ = (x = 0) \vee (x = 1) \vee ((x > 0) \wedge wp(W, x = 0)\{x \leftarrow x - 1\}\{x \leftarrow x - 1\}). \end{array}$$

Continuando con la computación, se llega a:

$$\begin{array}{l} wp(W, x = 0) \\ = (x = 0) \vee (x = 1) \vee (x = 2) \vee \ldots \\ \equiv x \geq 0 \end{array}$$

Finalizamos la introducción a la semántica con algunos resultados sobre wp recogidos en el siguiente resultado:

Teorema 5.20 *Sea* S *un programa y* p *y* q *postcondiciones cualesquiera. Entonces,*

- $\models wp(S,p) \wedge wp(S,q) \leftrightarrow wp(S, p \wedge q)$.
- $\models wp(S,q) \wedge wp(S,\neg q) \leftrightarrow wp(S, false)$.
- $\models \neg wp(S,\neg q) \rightarrow wp(S,q)$.
- *Si* $p \rightarrow q$, *tenemos* $\models wp(S,p) \rightarrow wp(S,q)$.

5.2. El sistema deductivo $\mathcal{HL}$

Un sistema deductivo cuyas fórmulas sean asertos puede usarse en la prueba de propiedades de programas. El sistema deductivo $\mathcal{HL}$ (Hoare Logic) es robusto y relativamente completo para probar la corrección parcial. Si un programa es parcialmente correcto, el aserto que codifica la corrección del programa puede probarse a partir de los axiomas y de las reglas de inferencia. Sin embargo, la completitud es relativa a la completitud de la completitud de la teoría del correspondiente dominio, dado que debemos asumir que *todas* las fórmulas verdaderas en el dominio son axiomas. Por ejemplo, si un programa opera en el dominio de los enteros, entonces el sistema $\mathcal{HL}$ no puede ser absolutamente completo porque la teoría de los enteros no es completa.

Definición 5.21 (Sistema deductivo $\mathcal{HL}$) *Se toman como axiomas todas las fórmulas verdaderas sobre los dominios a los que pertenecen las variables del programa junto a:*

- *Axioma de asignación*

$$\vdash \{p(x)\{x \leftarrow t\}\}\texttt{x:=t}, \{p(x)\}$$

- *Regla de composición*

$$\frac{\vdash \{p\}\ S_1\ \{q\} \qquad \vdash \{q\}\ S_2\ \{r\}}{\vdash \{p\}\ S_1; S_2\ \{r\}}$$

- *Regla de la alternativa*

$$\frac{\vdash \{p \wedge B\}\ S_1\ \{q\} \qquad \vdash \{p \wedge \neg B\} S_2 \{q\}}{\vdash \{p\}\ \texttt{si}\ \ B\ \texttt{entonces}\ \ S_1\ \texttt{si no}\ \ S_2\ \{q\}}$$

- *Regla del bucle*

$$\frac{\vdash \{p \wedge B\} S \{p\}}{\vdash \{p\}\ \texttt{mientras}\ \ B\ \texttt{hacer}\ \ \ S\{p \wedge \neg B\}}$$

- *Regla de la consecuencia*

$$\frac{\vdash p_1 \to p \quad \vdash \{p\}\ S\ \{q\} \quad \vdash q \to q_1}{\vdash \{p_1\}\ S\ \{q_1\}}$$

Algunas observaciones relevantes sobre la definición son las siguientes:

- Como $\vdash A \to A$ es un axioma para cualquier A del dominio, la regla de la consecuencia puede usarse para fortalecer una precondición o debilitar una postcondición.
- Si se quiere probar el aserto

$$\vdash \{p_0\}\ \texttt{x:= t}\ \{p(x)\},$$

se asume $\vdash \{p(x)\{x \leftarrow t\}\}\texttt{x:=t}, \{p(x)\}$.

Si ahora $p_0 \to p(x)\{x \leftarrow t\}$ es cierta, el uso de la regla de la consecuencia prueba el aserto.

- La fórmula p en la regla del bucle se llama invariante y describe el comportamiento de una sola ejecución de S en el cuerpo de la expresión `mientras`. Para probar

$$\vdash \{p_0\}\ \texttt{mientras}\ B\ \texttt{hacer}\ S\ \{q\},$$

basta encontrar un invariante p. Si $p_0 \to p$ la precondición de la regla del bucle se verifica. Si además, se puede demostrar que $(p \wedge \neg B) \to q$ es verdadera, no importa cuantas veces se ejecute el bucle, la postcondición de la regla implica la postcondición q. La dificultad en la prueba de la corrección parcial de programas radica en la elección de un invariante adecuado.

5.3. Verificación de programas

Usemos $\mathcal{HL}$ para probar la corrección parcial del siguiente programa P dado por:

$\{T\}$
`x:=0;`
$\{x = 0\}$
`y:=b`

$\{x = 0 \wedge y = b\}$
`mientras y` $\neq 0$ `hacer`
  $\{x = (b - y) \cdot a\}$
  `x:= x+a; y:= y-1;`
`fmientras`
$\{x = a \cdot b\}$

Hemos anotado P con fórmulas entre expresiones. Dado

$$\{p_1\}\ S_1\ \{p_2\}\ S_2\ \ldots \{p_n\}\ S_n\ \{p_{n+1}\}$$

probaremos $\{p_i\}\ S_i\ \{p_{i+1}\}$ para cada i y se concluye

$$\{p_1\}\ S_1\ ; \cdots\ ; S_n\ \{p_{n+1}\}$$

por la aplicación reiterada de la regla de composición. Aunque no lo haremos, se puede probar que el sistema $\mathcal{HL}$ con anotaciones es equivalente a $\mathcal{HL}$ sin ellas.

Veamos, pues, que $\vdash \{T\}\ P\ \{x = a \cdot b\}$. Por el axioma de asignación se tiene $\{0 = 0\}$ `x:=0` $\{x = 0\}$ y dado que $T \rightarrow (0 = 0)$, se tiene $\{T\}$ `x:=0` $\{x = 0\}$. La prueba de que $\{x = 0\}$ `y:=b` $\{(x = 0) \wedge (y = b)\}$ es similar.

Sea ahora p la fórmula $x = (b - y) \cdot a$. En el ejemplo 5.13 mostramos que $\{p\}$ `x:= x+a; y:= y-1` $\{p\}$, es decir, que p es un invariante para el cuerpo del bucle. Por la regla de la consecuencia, podemos fortalecer la precondición y

$$\{p \wedge y \neq 0\}\ \texttt{x:= x+a; y:= y-1}\ \{p\}.$$

Esta es la premisa necesaria para aplicar la regla del bucle, por lo que:

$\{p\}$
`mientras y` $\neq 0$ `hacer`
  `x:= x+a; y:= y-1;`
`fmientras`
$\{p \wedge \neg(y \neq 0)\}$

Finalmente, usando los axiomas del dominio y la regla de la consecuencia se deduce $x = a \cdot b$.

Ejemplo 5.22 *Probar la corrección parcial de cada uno de los siguientes programas:*

1. Cálculo de potencias:

   $\{a \geq 0 \ \wedge \ b \geq 0\}$

   ```
   x:=a; y:= b; z:=1;
     mientras y ≠ 0 hacer
       si impar(y) entonces y:=y-1; z:= x*z
                    si no, x:=x*x; y=coc(y,2)
       fsi
     fmientras
   ```

   $\{z = a^b\}$

2. Cálculo de máximo común divisor:

   $\{a > 0 \ \wedge \ b > 0\}$

   ```
   x:=a; y:= b
     mientras x≠y hacer
       si x >y) entonces x:=x-y
                si no, y:=y-x
       fsi
     fmientras
   ```

   $\{x = mcd(a, b)\}$

3. Cálculo de raíces cuadradas enteras:

   $\{a \geq 0\}$

   ```
   x:=0; y:= 1
     mientras y≤a hacer
       x:=x+1; y:= y+2*x+1
     fmientras
   ```

   $\{0 \leq x^2 \leq a < (x+1)^2\}$

4. Cálculo de la inversa de una lista. (**NOTA** `hd` significa la cabeza de una lista, `tl` su resto y `cons` pone un elemento a la cabeza de una lista):

$\{true\}$

```
L:=a; L1:= nil
  mientras L≠nil hacer
    L1:= cons(hd(L),L1); L:=tl(L)
  fmientras
```

$\{L1 = reverse(a)\}$

5. Algoritmo de Euclides extendido:

$\{a > 0;\ b \geq 0\}$

$$\texttt{S:=} \begin{pmatrix} a & b \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

mientras $\texttt{s}_{12} \neq 0$ hacer

$$\texttt{q:=coc(s}_{11}\texttt{,s}_{12}\texttt{); Q:=} \begin{pmatrix} 0 & 1 \\ 1 & -\texttt{q} \end{pmatrix}\texttt{; S:=S*Q}$$

fmientras

$\{s_{21}a + s_{31}b = s_{11} \ \wedge \ s_{11} = mcd(a, b)\}$

5.4. Derivación de programas

Los asertos también pueden utilizarse en la síntesis (o derivación) de programas, es decir, la construcción de programas directamente a partir de una especificación formal. La clave reside en encontrar invariantes para los bucles, ya que el resto de aspectos de la prueba de corrección parcial son puramente mecánicos.
Ahora nos concentramos en dos ejemplos concretos.

5.4.1. Raíz cuadrada entera

Desarrollaremos dos métodos para encontrar la raíz cuadrada entera de un entero no negativo, es decir, partimos del aserto:

$$\{0 \leq a\}\ S\ \{0 \leq x^2 \leq a < (x+1)^2\}.$$

Solución número 1

Usaremos un bucle para calcular valores de `x` hasta que se verifique la postcondición. Supongamos que tomamos la primera parte de la postcondición como invariante y no tratamos de verificar la segunda parte hasta que acabe el bucle. Esta idea da el siguiente bosquejo del programa

$\{0 \leq a\}$
`x:=?;`
`mientras B(x,a) hacer`
  $\{0 \leq x^2 \leq a^2\}$
  `x:= ?;`
`fmientras`
$\{0 \leq x^2 \leq a < (x+1)^2\}$.

Sea p el invariante del bucle $\{0 \leq x^2 \leq a\}$. Dada la precondición $0 \leq a$, necesariamente la primera expresión ha de ser `x:=0`. La postcondición del bucle es $p \wedge \neg$`B(x,a)`, por lo que `B(x,a)` debe escogerse para que se verifique la postcondición. En este caso, es fácil: `B(x,a)` ha de ser la negación de $a < (x+1)^2$, es decir, `B(x,a)` es `(x+1)`$^2 \leq$ `a`. Finalmente, como el bucle termina cuando `x` es suficientemente grande, en el cuerpo del bucle podemos incrementar el valor de `x` en una unidad. Se tiene, pues el programa (anotado):

$\{0 \leq a\}$
`x:=0;`
`mientras (x+1)`$^2 \leq$ `a hacer`
  $\{0 \leq x^2 \leq a\}$
  `x:= x+1;`
`fmientras`
$\{0 \leq x^2 \leq a < (x+1)^2\}$.

Para ver la corrección parcial del programa se debe probar que $\{p \wedge B\}\ S\ \{p\}$, es decir,

$$\{0 \leq x^2 \leq a \wedge (x+1)^2 \leq a\}\ \texttt{x:=x+1}\{0 \leq x^2 \leq a\}.$$

Ahora, tenemos

$$\{0 \leq (x+1)^2 \leq a\}\ \texttt{x:=x+1}\{0 \leq x^2 \leq a\},$$

pero

$$(0 \leq x^2 \leq a \wedge (x+1)^2 \leq a) \rightarrow (0 \leq (x+1)^2 \leq a),$$

así que el hecho de que p es invariante se sigue de la regla de la consecuencia.

Solución número 2

La derivación de un programa está determinada, normalmente, por la forma en la que se escribe la postcondición. Escribamos la postcondición introduciendo una nueva variable. Obtenemos

$$(0 \leq x^2 \leq a < y^2) \wedge (y = x + 1).$$

La introducción de la nueva variable hace que el bucle deba afectar a ambas variables. Como en la postcondición se tiene $y = x + 1$, esta es la condición que finalizará el bucle. Para que el invariante se verifique, el valor de `y` debe ser siempre mayor que el de `x`. Además, no tiene sentido que el valor de `y` sea mayor que `a +1` , luego inicializamos `y:= a+1` y añadimos la condición $x < y \leq a + 1$ al invariante y se obtiene

$$p = (0 \leq x^2 \leq a < y^2) \wedge (x < y \leq a + 1)$$

como candidato a invariante. El bosquejo del programa (anotado) queda así:

$\{0 \leq a\}$
`x:=0; y:=a+1;`
`mientras y`$\neq$`x+1 hacer`
  $\{(0 \leq x^2 \leq a < y^2) \wedge (x < y \leq a + 1)\}$
  `?;`
`fmientras`
$\{0 \leq x^2 \leq a < (x + 1)^2\}$.

Ahora, en el bucle podemos incrementar el valor de `x` o reducir el valor de `y` siempre que se mantenga el invariante. Una forma de hacerlo es tomar el punto medio entre los valores de `x` e `y`, es decir, el cociente de la división de $x + y$ entre dos. Llamemos `z:= (x+y) div 2` a este cociente. El valor de `z` se lo debemos asignar a `y` o a `x` pero manteniendo el invariante. Con estas consideraciones, el bucle debería verificar $\{p \wedge B\}\ S1\{p\}$. Por lo tanto, el bucle (anotado) debería ser de la forma

$\{p \wedge (y \neq x + 1)\}$
`z:=(x+y) div 2;`
$\{p \wedge (y \neq x + 1) \wedge (z = \lfloor (x + y)/2 \rfloor)\}$
`si Cond(x,y,z) entonces`
  $\{p\{x \leftarrow z\}\}$
  `x:=z;`
  `si no`
    $\{p\{y \leftarrow z\}\}$

```
        y:=z;
fsi
```
$\{p\}$

donde `Cond(x,y,z)` es una condición que, dada la regla de la alternativa debe escogerse tal que

$$\{p \wedge (y \neq x+1) \wedge (z = \lfloor (x+y)/2 \rfloor) \wedge Cond(x,y,z)\} \rightarrow$$

$$((0 \leq z^2 \leq a < y^2) \wedge (z < y \leq a+1))$$

y

$$\{p \wedge (y \neq x+1) \wedge (z = \lfloor (x+y)/2 \rfloor) \wedge \neg Cond(x,y,z)\} \rightarrow$$

$$((0 \leq x^2 \leq a < z^2) \wedge (x < z \leq a+1)).$$

Las primeras condiciones de las consecuencias en las implicaciones anteriores, se verifican escogiendo `Cond(x,y,z)= z`$^2 \leq$`a`. ¿Qué ocurre con las segundas premisas? Si $x < y \leq a+1$ y $z = \lfloor (x+y)/2 \rfloor$, se deduce $z < y \leq a+1$. De manera similar, si $x < y \leq a+1$ y $z = \lfloor (x+y)/2 \rfloor$, se deduce $x < z \leq a+1$ siempre y cuando $y \neq x+1$.

El programa final, cuya corrección parcial está probada, es:

$\{0 \leq a\}$
```
x:=0; y:= a+1;
mientras y≠x+1 hacer
```
$\{(0 \leq x^2 \leq a < y^2) \wedge x < y \leq a+1\}$
```
    z:=(x+y) div 2;
    si z² ≤a entonces x:=z si no y:=z fsi
```
$\{0 \leq x^2 \leq a < (x+1)^2\}$

Puede parecer extraño derivar el programa al mismo tiempo que se prueba la corrección del mismo. Sin embargo, este método es recomendabler porque asegura que no hay errores en el programa.

5.4.2. Selección por eliminación

Nota: Este ejemplo de derivación de un programa está tomado de J.L. Balcázar, "Programación metódica", McGraw-Hill, 1993 (páginas 317 y ss).

Sean dados dos números naturales m y n, con $m \leq n$ y sea P una propiedad cualquiera que pueden cumplir, o no, los naturales entre m y n. Supongamos que queremos saber si dados naturales r y s entre m y n, hay algún otro natural α, $r \leq \alpha \leq s$ que verifique la propiedad P. Partimos, pues, del aserto

$$m \leq r \leq s \leq n \; S \; b = \exists \alpha r \leq \alpha \leq s \wedge P(\alpha).$$

Desarrollaremos un programa recursivo. Teniendo en cuenta que la precondición nos asegura que el dominio del cuantificador existencial es no vacío, el caso más simple es aquel en que este tiene un único elemento. En ese caso, si $r = s$, $b := P(r)$. Para el caso $r \neq s$, empezamos efectuando la siguiente derivación, en la que utilizaremos la equivalencia lógica $P \longrightarrow Q \equiv ((P \vee Q) = Q)$ [2]:

$$\exists \alpha(r \leq \alpha \leq s \wedge P(\alpha)) \equiv P(r) \vee \exists \alpha(r+1 \leq \alpha \leq s \wedge P(\alpha))$$

$$\equiv (r \neq s) P(r) \vee P(s) \vee \exists \alpha(r+1 \leq \alpha \leq s-1 \wedge P(\alpha)) \equiv$$

$$\equiv (\text{si } P(r) \longrightarrow P(s)) P(s) \vee \exists \alpha(r+1 \leq \alpha \leq s-1 \wedge P(\alpha)) \equiv$$

$$\equiv \exists \alpha(r+1 \leq \alpha \leq s \wedge P(\alpha))$$

Análogamente, se tiene

$$\exists \alpha(r \leq \alpha \leq s \wedge P(\alpha)) \equiv (\text{si } P(s) \longrightarrow P(r)) \; \exists \alpha(r \leq \alpha \leq s-1 \wedge P(\alpha))$$

Por otro lado,

$$(P(r) \longrightarrow P(s)) \vee (P(s) \longrightarrow P(r)) \equiv \neg P(r) \vee P(s) \vee \neg P(s) \vee P(r) \equiv T,$$

luego tenemos todos los casos cubiertos. Si ahora hacemos que los valores de m y n pasen a ser parámetros del algoritmo recursivo, hemos obtenido el siguiente programa, cuya corrección parcial está probada en virtud de los argumentos anteriores:

$\{m \leq n\}$

```
r:=m; s:= n;
```

[2] En lo que sigue haremos uso de la siguiente notación $A \equiv (C)B$ para indicar que bajo la condición de que C sea cierta A y B son equivalentes.

```
        mientras r≠s hacer
Inv: {m ≤ r ≤ s ≤ n∧∃α(r ≤ α ≤ s∧P(α)) = ∃α(m ≤ α ≤ n∧P(α))}
              si P(r) ⟶ P(s) entonces r:=r+1
                si no s:=s-1
              fsi
        fmientras
        b:=P(r).
    {b := ∃α(m ≤ α ≤ n ∧ P(α))}
```

La última asignación garantiza que si b se devuelve como valor cierto, r es un elemento que cumple P y puede devolverse si es oportuno.

Veamos ahora qué ocurre cuando tenemos garantía de éxito, es decir, cuando tenemos la seguridad de que existen elementos que cumplen la propiedad. En este caso, como precondición se tendrá $\{\exists\alpha(m \leq \alpha \leq n \wedge P(\alpha))\}$. Hemos dicho ya que si el valor booleano es T, podemos afirmar que el elemento encontrado cumple la condición P. En efecto, a la salida del bucle,

$$Inv \wedge (r = s) \equiv$$

$$r = s \wedge m \leq r \leq s \leq n \wedge \exists\alpha(r \leq \alpha \leq s \wedge P(\alpha)) = \exists\alpha(m \leq \alpha \leq n \wedge P(\alpha)) \equiv$$

$$\equiv m \leq r = s \leq n \wedge \exists\alpha(m \leq \alpha \leq n \wedge P(\alpha) = P(r)),$$

por lo que la asignación $b := P(r)$ da a b el valor requerido por la postcondición. Devolviendo el propio valor de r y suponiendo la precondición que garantiza el éxito, se tiene:

```
    {∃α (m ≤ α ≤ n ∧ P(α))}
        r:=m; s:= n;
        mientras r≠s hacer
Inv: {m ≤ r ≤ s ≤ n ∧ ∃α(r ≤ α ≤ s ∧ P(α))}
              si P(r) ⟶ P(s) entonces r:=r+1
                si no s:=s-1
              fsi
        fmientras
    {m ≤ r ≤ n ∧ P(r)}
```

Es trivial ver que el nuevo invariante se deduce del anterior gracias a la nueva precondición:

$$m \leq r \leq s \leq n \wedge \exists\alpha(m \leq \alpha \leq n \wedge P(\alpha)) = \exists\alpha(r \leq \alpha \leq s \wedge P(\alpha)) \equiv$$

$$m \leq r \leq s \leq n \wedge T = \exists\alpha(r \leq \alpha \leq s \wedge P(\alpha)) \equiv$$

$$m \leq r \leq s \leq n \wedge \exists\alpha(r \leq \alpha \leq s \wedge P(\alpha)).$$

Ejemplo de aplicación: localización del máximo

Como aplicación de lo anterior, consideramos el problema de localizar el valor máximo de un vector no nulo de naturales de longitud N. Consideremos como especificación del programa a derivar

$$\{1 \leq N\}\ S\ \{1 \leq k \leq N \wedge \forall \beta 1 \leq \beta \leq N \longrightarrow a(\beta) \leq a(k)\},$$

donde $a(i)$ designa el natural que ocupa el lugar $i-$ésimo del vector a. Nótese que como precondición es necesario exigir $1 \leq N$, pues de lo contrario es imposible establecer $1 \leq k \leq N$. Por oro lado, para aligerar la notación en la discusión que sigue, definimos

$$P(a,i) \equiv \forall \beta 1 \leq \beta \leq N \longrightarrow a(\beta) \leq a(i)\}.$$

Se tiene:

a) • $a(i) \leq a(j) \longrightarrow (P(a,i) \longrightarrow P(a,j))$
   • $a(j) \leq a(i) \longrightarrow (P(a,j) \longrightarrow P(a,i))$

b) $T \equiv a(i) \leq a(j) \vee a(i) \leq a(j)$

Con esto en cuenta y lo hecho previamente, se deriva el siguiente programa:

$\{1 \leq N \wedge \exists\alpha\ (1 \leq \alpha \leq N \wedge P(a,\alpha))\}$
```
  i:=1; j:= N;
  mientras i≠j hacer
```
Inv: $1 \leq i \leq j \leq N \wedge \exists\alpha(i \leq \alpha \leq j \wedge P(a,\alpha))\}$
```
        si a(i)) ≤ a(j) entonces i:=i+1
          si no j:=j-1
        fsi
  fmientras
```
$\{1 \leq i \leq N \wedge P(a,i)\}$