Lógica en Ciencias de la Computación
Notas de Clase.

Carlos Olarte

Mayo 2013

Prefacio

Este documento presenta las notas de clase del curso de Lógica en Ciencias de la Computación de la Pontificia Universidad Javeriana Cali. El estudiante debe considerar este documento solamente como una guía para el desarrollo de la clase. Para lograr un mejor entendimiento de los temas aquí tratados, se recomienda consultar los textos guía del curso [Gal03] y [HR04] de donde proviene el material de estas notas. En algunos capítulos del curso también se sugiere consultar [BA01], [Abr10], [BC04], [NvP01] y [MN12]. Finalmente, material adicional del curso se puede encontrar en la página `http://cic.puj.edu.co/wiki/doku.php?id=materias:laogica-cs`.

Índice general

1. Preliminares **7**

2. Sintaxis de la Lógica Proposicional **9**
2.1. El lenguaje de la Lógica Proposicional 9
2.2. Pruebas Inductivas 10
2.3. Funciones definidas inductivamente 12
2.4. Ordenamiento de Parejas 14
2.5. Ejercicios 15

3. Semántica de la Lógica Proposicional **17**
3.1. Modelos y Valuaciones 17
3.2. Conectivos Funcionalmente Completos 20
3.2.1. Álgebras Boolenas 22
3.3. Ejercicios 23

4. Procedimientos de Decisión para Lógica Proposicional **27**
4.1. Sistemas à la Hilbert 27
4.2. Cálculo de Secuentes 27
4.3. Asistentes de Prueba 30
4.4. Lógica Clásica y Lógica Intuicionista 32
4.5. Resolución en Lógica Proposicional 36
4.5.1. Programación Lógica y Cláusulas de Horn 36
4.5.2. Cálculo de secuentes para Fórmulas en FNC 38
4.6. Propiedades de los Sistemas de Pruebas 39
4.6.1. Correctitud 39
4.6.2. Completitud 41
4.6.3. Consistencia 42
4.7. Ejercicios 43

5. Sintaxis y Semántica de la Lógica de Predicados **47**
5.1. La necesidad de variables y cuantificadores 47
5.2. Términos y Fórmulas 47
5.3. Semántica de la Lógica de Predicados 50
5.4. Ejercicios 53

6. Sistemas de Prueba para Lógica de Predicados **57**
6.1. Sistema Clásico 57
6.2. Sistema Intuicionista 58
6.3. FOL en Coq 60
6.4. Especifación y Verificación 61
6.5. Tipos Inductivos 73

6.5.1. Números Naturales . . . 74
6.5.2. Estructuras de Datos . . . 78
6.6. Ejercicios . . . 84
6.7. Propiedades de los Sistemas de Pruebas en FOL . . . 86
6.7.1. Correctitud . . . 87
6.7.2. Completitud . . . 88
6.7.3. Consistencia . . . 90
6.8. Indecidibilidad de FOL . . . 90

7. Resolución y Programación Lógica **93**
7.1. Formas Normales . . . 93
7.2. Modelos de Herbrand . . . 95
7.3. Resolución . . . 96
7.4. Programación Lógica . . . 100

Capítulo 1

Preliminares

A continuación se resumen algunas nociones fundamentales de conjuntos y relaciones que serán utilizadas en los siguientes capítulos del curso. El lector no familiarizado con estos conceptos debe referirse a [Gal11] o [Ros04].

Definición 1.1 (Relación de Equivalencia)**.** *La relación $R \subseteq A \times A$ es una relación de equivalencia si R es:*

1. *Reflexiva:*$(x, x) \in R$.
2. *Transitiva: si $(x, y) \in R$ y $(y, z) \in R$ entonces $(x, z) \in R$.*
3. *Simétrica: si $(x, y) \in R$ entonces $(y, x) \in R$*

Definición 1.2 (Ordenes Parciales)**.** *Una relación $R \subseteq A \times A$ es un orden parcial si R es:*

1. *Reflexiva:*$(x, x) \in R$.
2. *Transitiva: si $(x, y) \in R$ y $(y, z) \in R$ entonces $(x, z) \in R$.*
3. *Antisimétrica: si $(x, y) \in R$ y $(y, x) \in R$ entonces $x = y$*

La relación R es un **orden total** *si además de las anteriores condiciones se cumple que para todo $x, y \in A$, $(x, y) \in R$ o $(y, x) \in R$.*

Definición 1.3 (Conjuntos bien fundados [1])**.** *Dado un orden parcial $(A, \leq)$, decimos que A está bien fundado si no existen cadenas infinitas decrecientes de la forma*

$$.... \leq x_j \leq ...x_i \leq x_2 \leq x_1$$

Lema 1.1. *Dado el conjunto parcialmente ordenado $(A, \leq)$, A está bien fundado si y solamente si (sii) todo subconjunto no vacío de A tiene un elemento minimal.*

Para los conjuntos bien fundados podemos utilizar el principio de inducción que se define a continuación.

Teorema 1.1 (Principio de Inducción)**.** *Asuma que $(A, \leq)$ es un conjunto bien fundado. Para probar que cierta propiedad P es válida para todo elemento de A, es suficiente con probar lo siguiente:*

1. *Si x es minimal entonces pruebe que $P(x)$ es válido.*
2. *Si x no es minimal, asuma que $P(y)$ es cierto para todo $y < x$ y pruebe que $P(x)$ es válido.*

La anterior definición nos dice que para probar que cierta propiedad es válida en A, debemos primero probar que es válida para todo elemento minimal, es decir, para todo x tal que no existe y menor a x ($y < x$[2]). Esto representa el **caso base**. El **caso inductivo** se prueba asumiendo que la propiedad es válida para todo $y < x$. A partir de esta *hipótesis inductiva*, se debe mostrar que la propiedad es válida para x.

[1] Well-founded sets

[2] Si $y \leq x$ y $y \neq x$ escribimos $y < x$, es decir, y es estrictamente menor (con respecto al orden $\leq$) que x

Capítulo 2

Sintaxis de la Lógica Proposicional

En esta capítulo vamos a definir de manera formal el lenguaje de la lógica proposicional. Esto nos va a permitir construir *fórmulas* bien formadas. [1]. Además, veremos como podemos utilizar inducción para probar propiedades sobre el lenguaje de la lógica proposicional (LP).

2.1. El lenguaje de la Lógica Proposicional

Al lenguaje de la LP le vamos a dar dos usos en este curso:

1. Teoría de las pruebas (sintáctico-deductivo): Manipulación de las fórmulas de acuerdo a un conjunto de reglas. En este caso, no interesa el significado de las fórmulas sino lo que podemos probar de ellas.

2. Teoría de Modelos (semántico): nos interesa darle significado a las fórmulas. Dicho significado se construye a partir de un modelo.

Luego probaremos que los modelos y las pruebas coinciden, es decir, aquello que probamos es *verdad* (correctitud) y todo lo que es verdad lo podemos probar (completitud).

Para construir el lenguaje de LP, haremos uso de *variables proposicionales* que denotaremos con las letras $P, Q, R, S, ...$ Una variable proposicional denota una *aserción* atómica la cuál podremos evaluar como verdadera o falsa. Algunos ejemplos de tales *proposiciones atómicas*

1. El programa termina

2. La variable x en el programa siempre toma un valor positivo.

3. Está lloviendo.

A partir de las proposiciones atómicas, podemos construir proposiciones compuestas utilizando los siguientes símbolos.

Definición 2.1 (Alfabeto de LP)**.** *Las fórmulas en LP se pueden construir utilizando los siguientes símbolos:*

1. *Un conjunto (contable) de variables proposicionales:* $P_1, P_2, ...$

2. *La constante* **falso** *(*$\bot$*).*

3. *Los conectivos lógicos conjunción (*$\wedge$*), disyunción (*$\vee$*), implicación (*$\supset$*), negación (*$\neg$*) y equivalencia (*$\equiv$*).*

4. *Los símbolo auxiliares "(" y ")".*

[1] ¿Por qué se requiere otro lenguaje y no utilizamos el español ?

Definición 2.2 (Sintaxis de LP). *El conjunto* $\texttt{Prop}$ *de fórmulas proposicionales es el conjunto más pequeño de cadenas construidas a partir del alfabeto en la Definición 2.1 tal que:*

1. *Toda variable proposicional* $P_i \in \texttt{Prop}$.
2. $\bot \in \texttt{Prop}$.
3. *Si* $A \in \texttt{Prop}$ *entonces* $\neg A \in \texttt{Prop}$.
4. *Si* $A, B \in \texttt{Prop}$ *entonces* $(A \wedge B), (A \vee B), (A \supset B), (A \equiv B) \in \texttt{Prop}$.
5. *Una cadena pertenece a* $\texttt{Prop}$ *solamente si está formada mediante la aplicación de las anteriores reglas.*

Ejemplo 2.1 (Fórmulas bien formadas). *Las siguientes cadenas no pertenecen al conjunto* $\texttt{Prop}$*, es decir, no son fórmulas bien formadas:*

1. $()$
2. $A \wedge B$
3. $)A \wedge B($.

Las siguientes son fórmulas bien formadas, es decir, son elementos del conjunto $\texttt{Prop}$ [2]

1. $(A \wedge B)$
2. $\neg(A \wedge B)$
3. $(\neg A \supset (B \wedge C))$.

2.2. Pruebas Inductivas

Gracias a la definición inductiva de `Prop` (Definición 2.2), podemos utilizar el principio de inducción para probar propiedades de las fórmulas de la LP (ver Definición 1.1).

Teorema 2.1 (`Prop` es bien fundado). *Define el siguiente ordenamiento de fórmulas:*

1. $F \leq F$ *(reflexividad).*
2. $F < \neg F$
3. $F < (F \otimes G)$ *para* $\otimes \in \{\wedge, \vee, \supset, \equiv\}$
4. $G < (F \otimes G)$ *para* $\otimes \in \{\wedge, \vee, \supset, \equiv\}$

Entonces, $(\texttt{Prop}, \leq)$ *está bien fundado.*

Demostración. Asuma que F es una fórmula. Note que en toda cadena decreciente de la forma $F > F_1 > F_2 > ...$ el número de conectivos de F_{i+1} es estrictamente menor al número de conectivos de F_i. Además, no existe una fórmula G tal que $G < \bot$ o $G < P$ para una variable proposicional P. Entonces, la cadena anterior es finita y está limitada por el número de conectivos de F. □

A partir del teorema anterior, y por el principio de inducción, es claro que para probar que cierta propiedad se cumple para cualquier fórmula debemos probar que:

1. La propiedad se cumple para $\bot$ y P (una variable proposicional). Esto representa el caso base.
2. Asumiendo que la propiedad se cumple para F, se debe probar que también se cumple para $\neg F$.

[2] ¿Por qué?

3. Asumiendo que la propiedad se cumple para F y G se debe probar que la propiedad se cumple para $(F \wedge G)$, $(F \vee G)$, $(F \supset G)$ y $(F \equiv G)$.

A continuación presentamos algunos ejemplos.

Teorema 2.2 (Paréntesis)**.** *Toda fórmula en* `Prop` *tiene igual número de paréntesis abiertos y cerrados.*

Demostración. Primero definimos las funciones NPA : `Prop` $\rightarrow N$ y NPC : `Prop` $\rightarrow N$ que dada una fórmula retornan, respectivamente, el número de paréntesis abiertos y cerrados. Asuma que $F \in$ `Prop`. Procedemos por inducción en la estructura de F.

- **Caso** P_i. Si F es una variable proposicional, trivialmente $NPA(F) = NPC(F) = 0$.
- **Caso** $\bot$. Trivialmente $NPA(\bot) = NPC(\bot) = 0$.
- **Caso** $\neg G$. Asuma que F es de la forma $\neg G$. Por inducción, podemos asumir que $NPA(G) = NPC(G)$[3]. Como $NPA(F) = NPA(G)$ y $NPC(F) = NPC(G)$ concluimos que $NPA(\neg G) = NPC(\neg G)$.
- **Caso** $(G_1 \otimes G_2)$. Asuma ahora que F es de la forma $(G_1 \otimes G_2)$ donde $\otimes \in \{\wedge, \vee, \supset, \equiv\}$. Por hipótesis inductiva sabemos que $NPA(G_1) = NPC(G_1)$ y que $NPA(G_2) = NPC(G_2)$. Es fácil ver que

$$
\begin{array}{rcl}
NPA(F) & = & NPA(G_1) + NPA(G_2) + 1 \\
NPC(F) & = & NPC(G_1) + NPC(G_2) + 1
\end{array}
$$

  Podemos concluir entonces que $NPA(F) = NPC(F)$.

□

Teorema 2.3. *Cualquier prefijo propio de una formulas es:*

1. *La cadena vacía, o*
2. *Una cadena no vacía de símbolos "¬", o*
3. *Una cadena con excesos de "("*

Demostración. Procedemos por inducción en la estructura de la fórmula.

- **Caso Base**. El único prefijo de P_i y de $\bot$ es la cadena vacía (condición (1)).
- **Caso** $\neg F$ Asume que todo prefijo de F cumple con alguna de las condiciones anteriores. Los posibles prefijos propios de $\neg F$ son:

  1. $\neg F'$ donde F' es un prefijo propio de F'. Si F' es la cadena vacía, entonces "¬" cumple con la condición (2). Si F' cumple con (2), entonces $\neg F'$ también cumple con (2). Finalmente, si F' cumple con (3), $\neg F'$ tiene el mismo número de "(" y por tanto, $\neg F'$ cumple con (3).
  2. Considere "¬". Este caso es igual al primer caso anterior.
  3. Finalmente se considera la cadena vacía que trivialmente cumple con (1)

- **Caso** $(F \otimes G)$ donde $\otimes \in \{\wedge, \vee, \supset, \equiv\}$.Asuma que F' u G' son prefijos propios de F y G respectivamente. Por hipótesis inductiva, sabes que F' y G' cumplen con alguna de las 3 condiciones. Los posibles prefijos propios de $(F \otimes G)$ son:

  1. $(F \otimes G'$. No consideramos que G' sea la cadena vacía. Esto lo consideramos en caso 2. abajo. Si G' cumple con (2), entonces $(F \otimes G'$ tiene exceso de paréntesis abiertos. Para esto, note que F está bien formada. Si G' cumple con (3), "$(F \otimes G$" tiene un paréntesis abierto extra, cumpliendo así con (3).

[3]Recuerde que en el paso inductivo asume que la propiedad es cierta para los elementos más pequeños, en este caso, para las *subformulas* de la fórmula que está considerando.

2. $(F\otimes$. La secuencia tiene exceso de paréntesis abiertos. Para esto note que F está bien formada.
3. $(F$. Este caso es igual al anterior.
4. $(F'$. Asuma que F' no es vacía (el caso cuando es vacía se considera en el caso 5.). Si F' cumple con (2), entonces "$(F'$" cumple con (3). Si F' cumple con (3), entonces "$(F'$" cumple con (3).
5. (. Trivialmente cumple con (3).
6. ϵ. Trivialmente cumple con (1).

□

Corolario 2.1. *Ningún prefijo propio de una fórmula es una fórmula bien formada.*

Demostración. Asuma que F es una fórmula y F' un prefijo propio de F. Por el Teorema 2.3, F' es una cadena vacía, una secuencia no vacía de $\neg$ o una fórmula con exceso de paréntesis abiertos. Ninguno de los tres casos anteriores corresponde a una fórmula bien formada. □

2.3. Funciones definidas inductivamente

Definición 2.3 (Número de Conectivos)**.** *Definimos el número de conectivos de una fórmula de la siguiente manera:*

$$NC(\bot) = NC(P) = 0$$
$$NC(\neg F) = 1 + NC(F)$$
$$NC((F \otimes G)) = 1 + NC(F) + NC(G)$$

Ahora definamos las subfórmulas que resultan de una fórmula

Definición 2.4 (Subfórmulas)**.** *Definimos las subfórmulas de una fórmula como $SUB : \texttt{Prop} \to \mathcal{P}(\texttt{Prop})$ de la siguiente manera:*

$$SUB(\bot) = \{\bot\}$$
$$SUB(P) = \{P\}$$
$$SUB(\neg F) = SUB(F) \cup \{\neg F\}$$
$$SUB((F \otimes G)) = SUB(F) \cup SUB(G) \cup \{(F \otimes G)\}$$

Lema 2.1. *Asuma que F es una fórmula bien formada tal que $NC(F) = n$. Entonces, $SUB(F)$ tiene a lo más $2n+1$ fórmulas, es decir:*

$$|SUB(F)| \leq 2n + 1$$

Demostración. Procedemos por inducción en la estructura de la fórmula F.

- **Caso Base**. Si F es $\bot$ o P, el número de conectivos es 0 y $|SUB(F)| = 1$. Trivialmente, $1 \leq 2 \times 0 + 1$.
- **Caso** $\neg F$. Asumimos que $|SUB(F)| \leq 2n + 1$ donde $n = NC(F)$. Sabemos que $NC(\neg F) = n + 1$ y que $SUB(\neg F) = SUB(F) \cup \{\neg F\}$. Así que tenemos lo siguiente:

$$|SUB(F)| \leq 2n + 1 \quad \text{(hipótesis)}$$
$$|SUB(F)| + 1 \leq 2n + 1 + 1 \quad \text{(aritmética)}$$
$$|SUB(\neg F)| \leq 2n + 1 + 1 \quad \text{(Definición)}$$
$$|SUB(\neg F)| \leq 2(n + 1) \quad \text{(aritemética)}$$
$$|SUB(\neg F)| \leq 2(n + 1) + 1 \quad \text{(aritemética)}$$

- **Caso** $(F \otimes G)$ donde $\otimes \in \{\wedge, \vee, \supset, \equiv\}$. Asuma que $NC(F) = n$, $NC(G) = m$ y por hipótesis, $|SUB(F)| \leq 2n+1$ y $|SUB(G)| \leq 2m+1$. Sabemos que $SUB((F\otimes G)) = SUB(F) \cup SUB(G) \cup \{(F \otimes G)\}$. Además, $NC((F \otimes G)) = 1 + n + m$. Podemos entonces deducir lo siguiente:

$$|SUB((F \otimes G))| \leq |SUB(F)| + |SUB(G)| + 1 \quad \text{(teoría de conjuntos)}$$
$$|SUB((F \otimes G))| \leq 2n + 1 + 2m + 1 + 1 \quad \text{(hipótesis y aritmética)}$$
$$|SUB((F \otimes G))| \leq 2(n + m + 1) + 1 \quad \text{(aritmética)}$$

□

Definamos ahora otra medida para las fórmulas.

Definición 2.5 (Profundidad de una Fórmula). *Defina la profundidad de una fórmula $L(F)$ como:*

$$
\begin{aligned}
&L(\bot) = L(P) = 1 \\
&L(\neg F) = 1 + L(F) \\
&L((F \otimes G)) = 1 + L(F) + L(G)
\end{aligned}
$$

A partir de la anterior definición, podemos probar lo siguiente.

Lema 2.2. *Para todo fórmula F, $|SUB(F)| \leq L(F)$*

Demostración. Procedemos por inducción en la estructura de F.

- **Caso Base**. Si $F = P$ o $F = \bot$ entonces, trivialmente $1 \leq 1$.
- **Caso $\neg F$**, Asuma que $|SUB(F)| \leq L(F)$. Sabemos que $SUB(\neg F) = SUB(F) \cup \{\neg F\}$ y que $L(\neg F) = 1 + L(F)$. Además, $|SUB(\neg F)| = |SUB(F)| + 1$. Como $|SUB(F)| \leq L(F)$ (por hipótesis), trivialmente $|SUB(F)| + 1 \leq L(F) + 1$.
- **Caso** $(F \otimes G)$ donde $\otimes \in \{\wedge, \vee, \supset, \equiv\}$. Asuma que $|SUB(F)| \leq L(F)$ y $|SUB(G)| \leq L(G)$. Sabemos que $SUB((F \otimes G)) = SUB(F) \cup SUB(G) \cup \{(F \otimes G)\}$. Además, $L((F \otimes G)) = 1 + L(F) + L(G)$. Por lo tanto, tenemos que: [4]

  $$|SUB((F \otimes G))| \leq |SUB(F)| + |SUB(G)| + 1$$

  Utilizando las hipótesis (y artimética), concluimos que

  $$|SUB((F \otimes G))| \leq L(F)| + L(G) + 1$$

□

Definición 2.6 (Medida). *Dada una fórmula F, definimos la medida de F como:*

$$
\begin{aligned}
&M(\bot) = M(P) = 0 \\
&M(\neg F) = 1 + M(F) \\
&M((F \otimes G)) = 1 + max(M(F), M(G))
\end{aligned}
$$

Teorema 2.4 (Formación de Fórmulas). *Definimos $Prop^i$ inductivamente como sigue:*

$$
\begin{aligned}
&Prop^0 = PVars \cup \{\bot\} \\
&Prop^1 = Prop^0 \cup \{\neg F \mid F \in Prop^0\} \cup \{(F \otimes G) \mid F, G \in Prop^0\} \\
&\ldots \\
&Prop^n = Prop^{n-1} \cup \{\neg F \mid F \in Prop^{n-1}\} \cup \{(F \otimes G) \mid F, G \in Prop^{n-1}\}
\end{aligned}
$$

Si $F \in Prop^i$ entonces $M(F) \leq i$

Demostración. Procedemos por inducción en los números naturales.

- **Caso $i = 0$**. Trivialmente para todo $F \in Prop^0$, $M(F) = 0$.
- **Caso $i > 0$**. Asuma que para todo $m < i$, si $F \in Prop^m$ entonces $M(F) \leq m$. Asuma ahora que $G \in Prop^i$. Consideramos tres casos:
  - Si G pertenece a $Prop^{i-1}$, por hipótesis $M(G) \leq i - 1$ y por tanto, $M(G) \leq i$

[4] Note que $SUB(F) \cap SUB(G)$ no necesariamente es $\emptyset$.

- Si G es de la forma $\neg F$ donde $F \in Prop^{i-1}$, por hipótesis, $M(F) \leq i-1$. Sabemos que $M(G) = M(F)+1$ y trivialmente $M(F)+1 \leq i$.
- Finalmente, asuma que G es de la forma $(F' \otimes F'')$ para $F', F'' \in Prop^{i-1}$. Por hipótesis, $M(F') \leq i-1$ y $M(F'') \leq i-1$. Sabemos que $M(G) = 1+max(M(F'), M(F''))$. Por lo tanto, $M(G) \leq 1+i-1$ y concluimos $M(G) \leq i$.

□

2.4. Ordenamiento de Parejas

En el Teorema 2.5 vamos a probar una propiedad para el conjunto de parejas de las números naturales, es decir, para $N \times N$. Para esto, necesitamos primero definir un ordenamiento sobre dicho conjunto.

Definición 2.7 (Orden de $N \times N$)**.** *Dadas dos parejas (x, y) y (x', y') de números naturales, decimos que $(x, y) \leq (x', y')$ si:*

1. *$x = x'$ y $y = y'$, o*
2. *$x < x'$, o*
3. *$x = x'$ y $y < y'$*

Es importante notar que en la definición anterior, la primera componente de la pareja predomina en el ordenamiento. Esto se conoce como un orden lexicográfico.

Si queremos utilizar el principio de inducción sobre parejas de naturales, primero debemos asegurarnos que $((N \times N) \leq)$ está bien fundado.

Observación 2.1 (Well-foundness)**.** *$((N \times N) \leq)$ está totalmente ordenado y es un conjunto bien fundado.*

Demostración. Es fácil mostrar que $\leq$ es una relación reflexiva, transitiva y antisimétrica. Además, $\leq$ es un orden total: $(a, b) \leq (a', b')$ o $(a', b') \leq (a, b)$.

Para probar que el conjunto está bien fundado, asuma, a manera de contradicción, que $(N \times N, \leq)$ no está bien fundado. Entonces debe existir una cadena infinita decreciente de la forma:

$$\cdots \leq (x_2, y_2) \leq (x_1, y_1)$$

Consideramos dos casos:

1. Hay una cadena infinita decreciente $\cdots \leq x_2 \leq x_1$. Esto contradice que $(N, \leq)$ es bien fundado.
2. Asuma que solo hay cadenas finitas decrecientes de la forma $x_n \leq ... \leq x_2 \leq x_1$. Entonces, debe existir una cadena infinita de la forma $\cdots j_{k+1} \leq j_k \leq .... \leq j_1$, lo cuál, de nuevo, contradice que $(N, \leq)$ está bien fundado.

□

Con la ayuda de la anterior definición podemos probar lo siguiente:

Teorema 2.5. *La función de Ackermann se define de la siguiente manera:*

$$A(x, y) = \begin{cases} y+1 & Si\ x = 0 \\ A(x-1, 1) & Si\ x > 0\ y\ y = 0. \\ A(x-1, A(x, y-1)) & Si\ x > 0\ y\ y > 0. \end{cases}$$

La función $A(x, y)$ es total, es decir, está definida para cualquier pareja en $N \times N$.

Demostración. Procedemos por inducción en el ordenamiento $(N \times N, \leq)$ como en la Definición 2.7.

- **Caso Base: (0,0)**. Trivialmente la función está definida y se evalúa a 1.
- **Caso Inductivo: (m,n)**. Asuma que para cualquier pareja $(a,b) < (m,n)$ la función $A(a,b)$ está definida. De acuerdo con la definición de $A(x,y)$ debemos considerar tres casos:
  1. Considere que $m = 0$. Entonces $A(0,n) = n+1$.
  2. Considere que $m > 0$ y $n = 0$. Entonces $A(m,n) = A(m-1,1)$. Note que $(m-1,1) < (m,n)$, así que por hipótesis, $A(m-1,1)$ está definido y por tanto, $A(m,n)$ está definido.
  3. Asuma que $m,n > 0$. Entonces $A(m,n) = A(m-1, A(m,n-1))$. Debemos probar que ambos, $A(m,n-1)$ y $A(m-1,A(m,n-1))$ están definidos. En el primer caso, notamos que $(m,n-1) < (m,n)$ y por tanto, está definido. En el segundo caso, asuma que el valor de $A(m,n-1) = x$. Notamos que $(m-1,x) < (m,n)$ y por inducción, está definida.

□

2.5. Ejercicios

Ejercicio 2.1. *Asuma un conjunto infinito de constantes $a_1, a_2, ...$. A partir de dichas constantes se definen los* tokens *que un programa puede utilizar de la siguiente manera:*

$$\mathcal{C} ::= a_i \mid (\mathcal{C} \otimes \mathcal{C}) \mid \triangle\mathcal{C}$$

Expresiones en el lenguaje se construyen a partir de los tokens por medio de la siguiente sintaxis:

$$\mathcal{P} ::= \texttt{skip} \mid \bigstar(\mathcal{P}) \mid (\mathcal{P}\blacklozenge\mathcal{P}) \mid \bullet\mathcal{C}$$

1. *Escriba 5 tokens (expresiones tipo $\mathcal{C}$) bien formados.*
2. *Escriba 5 programas (expresiones tipo $\mathcal{P}$) bien formadas en este lenguaje.*
3. *Asuma que P es un programa bien formado. Demuestre que si hay una ocurrencia del operador "$\triangle$" en P, dicha ocurrencia está precedida de una ocurrencia del operador "$\bullet$"* [5]

Diga si las siguientes frases son ciertas o falsas. Si considera que es falsa, muestre un contraejemplo. Si es cierta, escriba la prueba respectiva.

1. *El número de ocurrencias de "$\triangle$" en un programa bien formado es par.*
2. *Asuma que el proceso P no contiene ocurrencias del operador "$\blacklozenge$". El número de ocurrencias de "$\bullet$" en P es 0 o 1.*
3. *En un programa bien formado "`skip`" puede aparece máximo 1 vez.*
4. *Todo prefijo propio de un token es (1) la secuencia vacía; o (2) una secuencia de uno o varios "$\triangle$"; o (3) una secuencia con excesos de "(".*

[5] Por ejemplo, en la secuencia $abcdef...$ decimos que c *precede* a e

Capítulo 3

Semántica de la Lógica Proposicional

En este capítulo vamos a darle *significado* (semántica) a las fórmulas del lenguaje de la lógica proposicional. Para ello, definiremos modelos en los cuáles se les asigna valores de verdad a las variables proposicionales de la fórmula. Finalmente, definiremos cuando una fórmula es *válida* (una *tautología*), *satisfacible* e *insatisfacible*.

3.1. Modelos y Valuaciones

Para darle significado a una fórmula tal como $(P \wedge (Q \supset R))$, debemos definir una valuación para las variables proposicionales (en este caso P, Q, R) y luego asignar un significado a los conectivos lógicos. Las fórmulas son entonces evaluadas a dos posibles valores: verdadero ($\mathtt{T}$) y falso ($\mathtt{F}$). Dichos valores conforman el conjunto $\mathtt{Bool} = \{\mathtt{T}, \mathtt{F}\}$ que está totalmente ordenado ($\mathtt{F} < \mathtt{T}$).

Definición 3.1 (Valuación)**.** *Asuma que $VarP$ es el conjunto de las variables proposicionales. Una valuación $v : VarP \to \mathtt{Bool}$ es una función que asigna un valor de verdad a cada variable proposicional.*

Por ejemplo, podemos considerar la valuación $v = \{P \mapsto \mathtt{T}, Q \mapsto \mathtt{F}, R \mapsto \mathtt{F}, \cdots\}$.

A partir de una valuación, podemos interpretar (dar significado) a una fórmula de manera inductiva.

Definición 3.2 (Semántica)**.** *Dada una valuación $v : VarP \to \mathtt{Bool}$, la semántica de una fórmula F se define como la función $[\![\cdot]\!] : (VarP \to \mathtt{Bool}) \to \mathtt{Prop} \to \mathtt{Bool}$*

$$
\begin{array}{lcl}
[\![\bot]\!]_v & = & \mathtt{F} \\
[\![P]\!]_v & = & v(P), \ \textit{si } P \textit{ es una variable proposicional} \\
[\![\neg F]\!]_v & = & \mathtt{neg}([\![F]\!]_v) \\
[\![(F_1 \wedge F_2)]\!]_v & = & \mathtt{con}([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \vee F_2)]\!]_v & = & \mathtt{disj}([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \supset F_2)]\!]_v & = & \mathtt{imp}([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \equiv F_2)]\!]_v & = & \mathtt{equiv}([\![F_1]\!]_v, [\![F_2]\!]_v)
\end{array}
$$

Donde

P	Q	$\mathtt{neg}(P)$	$\mathtt{con}(P,Q)$	$\mathtt{disj}(P,Q)$	$\mathtt{imp}(P,Q)$	$\mathtt{equiv}(P,Q)$
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Note que la valuación de una fórmula compuesta se deriva *composicionalmente* de la valuación de sus subfórmulas. Además, el resultado de la evaluación depende *únicamente* de la valuación v de las variables proposicionales. Dicho de otra forma, el significado de los conectivos lógicos no varía y la valuación final de la fórmula depende de v.

Ejemplo 3.1 (Interpretación de Fórmulas)**.** *Asuma la siguiente valuación:* $v = \{P \mapsto \mathtt{F}, Q \mapsto \mathtt{T}, R \mapsto \mathtt{T}\}$.

1. *Asuma* $F = (P \wedge Q)$*:*

$$\begin{aligned} [\![F]\!]_v &= \mathtt{con}([\![P]\!]_v, [\![Q]\!]_v) \\ &= \mathtt{con}(v(P), v(Q)) \\ &= \mathtt{con}(\mathtt{F}, \mathtt{T}) = \mathtt{F}. \end{aligned}$$

2. *Asuma* $F = (Q \supset (Q \wedge (P \vee R)))$

$$\begin{aligned} [\![F]\!]_v &= \mathtt{imp}(v(Q), [\![(Q \wedge (P \vee R)]\!]_v) \\ &= \mathtt{imp}(\mathtt{F}, \mathtt{con}(v(Q), [\![(P \vee R)_v]\!])) \\ &= \mathtt{imp}(\mathtt{F}, \mathtt{con}(\mathtt{T}, \mathtt{disj}(\mathtt{F}, \mathtt{T}))) \\ &= \mathtt{imp}(\mathtt{F}, \mathtt{con}(\mathtt{T}, \mathtt{T})) \\ &= \mathtt{imp}(\mathtt{F}, \mathtt{T}) = \mathtt{T} \end{aligned}$$

Definición 3.3 (Modelo de una Fórmula)**.** *Dada una fórmula* F *y una valuación* v*, si* $[\![F]\!]_v = \mathtt{T}$ *decimos que* v *es un modelo para* F *o que* v satisface *a* F *y escribimos* $v \models F$*. Si* v *no satisface* F*, es decir,* v *hace falsa la fórmula (*$[\![F]\!]_v = \mathtt{F}$*) escribimos* $v \not\models F$*. Dado un conjunto de fórmulas* $\Gamma = \{F_1, \cdots, F_n\}$*, si* $v \models F_i$ *para toda* $F_i \in \Gamma$*, decimos que* v *es un modelo de* Γ *y escribimos* $v \models \Gamma$*. Si* v *no es modelo para alguna fórmula* $F_i \in \Gamma$*, escribimos* $v \not\models \Gamma$*.*

Dependiendo de la existencia o no de una valuación que haga verdadera una formula, tendremos fórmulas que son satisfacibles e insatisfacibles.

Definición 3.4 (Satisfacibilidad)**.** *Dada una fórmula* F*, si existe una valuación* v *tal que* $[\![F]\!]_v = \mathtt{T}$*, decimos que* F *es satisfacible. De lo contrario, decimos que* F *es insatisfacible.*

Las fórmulas que independientemente de la valuación v son ciertas, las llamaremos *tautologías*.

Definición 3.5 (Tautologías y validez)**.** *Una fórmula* F *se dice que es* válida *si para cualquier valuación* v*,* $[\![F]\!]_v = \mathtt{T}$ *y escribimos* $\models F$*.* F *también se conoce como una* tautología*.*

Ejemplo 3.2. *A continuación presentamos algunos ejemplos de fórmulas satisfacibles, insatisfacibles y válidas.*

- *La fórmula* $(P \wedge Q)$ *es satisfacible, puesto que la valuación* $v = \{P \mapsto \mathtt{T}, Q \mapsto \mathtt{T}\}$ *la hace verdadera, es decir,* $v \models (P \wedge Q)$*. Sin embargo,* $(P \wedge Q)$ *no es válida puesto que, por ejemplo,* $\{P \mapsto \mathtt{T}, Q \mapsto \mathtt{F}\} \not\models (P \wedge Q)$

- *La fórmula* $(P \supset P)$ *es válida. Para esto, note que cualquier valuación para* P *hace cierta la implicación:*

P	$(P \supset P)$
$\mathtt{T}$	$\mathtt{imp}(\mathtt{T}, \mathtt{T}) = \mathtt{T}$
$\mathtt{F}$	$\mathtt{imp}(\mathtt{F}, \mathtt{F}) = \mathtt{T}$

- *La fórmula* $(\neg P \wedge P)$ *is insatisfacible puesto que ninguna valuación para* P *la hace verdadera:*

P	$(\neg P \wedge P)$
$\mathtt{T}$	$\mathtt{con}(\mathtt{neg}(\mathtt{T}), \mathtt{T}) = \mathtt{F}$
$\mathtt{F}$	$\mathtt{con}(\mathtt{neg}(\mathtt{F}), \mathtt{F}) = \mathtt{F}$

La anterior noción de satisfabilidad se puede extender a conjuntos de fórmulas de la siguiente manera.

Definición 3.6 (Satisfabilidad de Conjuntos de Fórmulas)**.** *Dado un conjunto de fórmulas* $\Gamma = \{F_1, F_2, ..., F_n\}$*, decimos que* Γ *es satisfacible si existe una valuación* v *tal que* $[\![F_i]\!]_v = \mathtt{T}$ *para cada* $F_i \in \Gamma$*, es decir,* $v \models \Gamma$*. Decimos que* Γ *es insatisfacible si no es satisfacible.*

Lema 3.1 (Satisfabilidad de Conjuntos de Fórmulas)**.** *Sea* $\Gamma = \{F_1, F_2, ..., F_n\}$ *un conjunto de fórmulas.*

$$\Gamma \textit{ es satisfacible sii } \bigwedge_{i \in 1..n} F_i \textit{ es satisfacible.}$$

Demostración. Inmediato a partir de la definición. □

Definición 3.7 (Consecuencia Lógica)**.** *Asuma un conjunto de fórmulas* $\Gamma = \{F_1, F_2, ..., F_n\}$*. Decimos que* A *es una consecuencia lógica de* Γ *si para toda valuación* v *tal que* $[\![F_i]\!]_v = \mathtt{T}$ *para* $i \in 1..n$ *implica que* $[\![A]\!]_v = \mathtt{T}$*. Si* A *es consecuencia lógica de* Γ *escribimos* $\Gamma \models A$*.*

Ejemplo 3.3 (Consecuencia Lógica)**.** *Observe que* $P \models P$ *puesto que todo modelo de* P *es trivialmente un modelo de* P*. Por otro lado,* $P, (P \supset Q) \models Q$*. Para ver esto, asuma que* v *es un modelo de* P*. Por lo tanto* $v(P) = \mathtt{T}$*. Si* v *es un modelo de* $(P \supset Q)$*, dado que* $v(P) = \mathtt{T}$*, entonces, debe ser cierto que* $v(Q) = \mathtt{T}$*. Así que,* $v \models Q$*.*

Teorema 3.1. *La fórmula* F *es una tautología sii* $\neg F$ *es insatisfacible.*

Demostración. Tenemos que probar dos cosas:

1. ($\Rightarrow$) Asuma que F es una tautología. Por lo tanto, para toda v tenemos $[\![F]\!]_v = \mathtt{T}$. Como $[\![\neg F]\!]_v = \mathtt{neg}([\![F]\!]_v) = \mathtt{neg}(\mathtt{T}) = \mathtt{F}$ concluimos que F es insatisfacible.

2. ($\Leftarrow$) Ahora asuma que $\neg F$ es insatisfacible. Entonces, para toda v, $[\![\neg F]\!]_v = \mathtt{F}$. Por lo tanto, si $\mathtt{neg}([\![F]\!]_v) = \mathtt{F}$, por la definición de $\mathtt{neg}(\cdot)$, tenemos que $[\![F]\!]_v = \mathtt{T}$ y concluimos que para toda v $[\![F]\!]_v = \mathtt{T}$, es decir, F es una tautología.

□

Teorema 3.2. *Dado un conjunto de fórmulas* $\Gamma = \{G_1, \cdots, G_n\}$*y una fórmula* F*,* $\Gamma \models F$ *iff* $\models (\bigwedge_i G_i) \supset F$

Demostración. Vamos a denotar con H la conjunción $\bigwedge_i G_i$ donde $G_i \in \Gamma$. Debemos probar dos cosas:

1. ($\Rightarrow$) Asuma que $\Gamma \models F$. Dada una valuación v consideramos dos casos:

   a) Si $v \not\models G_j$ para algún $G_j \in \Gamma$, entonces por la definición de $\mathtt{con}(\cdot)$ sabemos que $v \not\models H$ y por la definición de $\mathtt{imp}(\cdot)$, trivialmente $v \models H \supset F$.

   b) Asuma ahora que $v \models G_j$ para todo $G_j \in \Gamma$. Como $\Gamma \models F$, por definición, sabemos que $v \models F$ y por tanto, $v \models H \supset F$.

   Puesto que para cualquier v se cumple que $v \models H \supset G$, concluimos que $\models H \supset F$.

2. ($\Leftarrow$) Ahora asumimos que $\models H \supset F$. Por definición, para toda v se cumple que $v \models H \supset F$. Por la definición de $\mathtt{imp}(\cdot)$, si $v \models H$ entonces $v \models F$. Además, si $v \models H$, entonces $v \models G_j$ para todo $G_j \in \Gamma$. Concluimos entonces que $\Gamma \models F$.

□

Teorema 3.3. *Sea* $\Gamma = \{G_1, \cdots, G_n\}$ *un conjunto de fórmulas y* F *una fórmula.*

$$\Gamma \models F \textit{ sii } \Gamma \cup \{\neg F\} \textit{ es insatisfacible}$$

Demostración. Tenemos que probar dos cosas:

1. ($\Rightarrow$) Asuma que $\Gamma \models F$. Por definición, para toda valuación v, si $v \models \Gamma$ entonces $v \models F$. Si $v \models F$ sabemos que $v \not\models \neg F$. Así que para toda valuación v, si $v \models \Gamma$ entonces $v \not\models \neg F$ y por tanto, $v \not\models \Gamma \cup \{\neg F\}$, es decir, $\Gamma \cup \{\neg F\}$ es insatisfacible.

2. ($\Leftarrow$) Asuma que $\Gamma \cup \{\neg F\}$ es insatisfacible. Es decir, no existe una valuación que satisfaga a ambos, Γ y a $\neg F$. Sea v una valuación tal que $v \models \Gamma$. Sabemos que $v \not\models \neg F$ y por tanto, $v \models F$. Así que concluimos que $\Gamma \models F$.

□

3.2. Conectivos Funcionalmente Completos

A partir de equivalencias lógicas, vamos a probar que podemos reducir el conjunto de conectivos lógicos en el lenguaje de la lógica proposicional y no perdemos poder de expresividad. Primero, debemos notar que el conectivo $\equiv$ induce una relación de equivalencia entre fórmulas.

Definición 3.8 (Equivalencia)**.** *Decimos que F y G son lógicamente equivalentes sii $(F \equiv G)$ es una tautología.*

Lema 3.2 (Equivalencias)**.** *Las siguientes propiedades son ciertas:*

1. $\models (P \equiv Q) \equiv ((P \supset Q) \wedge (Q \supset P))$
2. $\models (P \supset Q) \equiv (\neg P \vee Q)$
3. $\models (P \vee Q) \equiv (\neg P \supset Q)$
4. $\models (P \wedge Q) \equiv \neg(\neg P \vee \neg Q)$
5. $\models (P \vee Q) \equiv \neg(\neg P \wedge \neg Q)$
6. $\models \neg P \equiv (P \supset \bot)$
7. $\bot \equiv (P \wedge \neg P)$

Demostración. Por el momento no tenemos más remedio que construir las tablas de verdad y verificar que para cada propiedad de la forma $\models F \equiv G$, se cumple que $[\![F]\!]_v = [\![G]\!]_v$ para cualquier v.

P	Q	$(P \equiv Q)$	$F_1 : (P \supset Q)$	$F_2 : (Q \supset P)$	$(F_1 \wedge F_2)$	$(\neg P \vee Q)$	$(P \vee Q)$	$(\neg P \supset Q)$
T	T	T	T	T	T	T	T	T
T	F	F	F	T	F	F	T	T
F	T	F	T	F	F	T	T	T
F	F	T	T	T	T	T	F	F

P	Q	$(P \wedge Q)$	$\neg(\neg P \vee \neg Q)$	$\neg(\neg P \wedge \neg Q)$	$\neg P$	$(P \supset \bot)$	$(P \wedge \neg P)$
T	T	T	T	T	F	F	F
T	F	F	F	T	F	F	F
F	T	F	F	T	T	T	F
F	F	F	F	F	T	T	F

□

Existen otros conectores lógicos tales como XOR (o-exclusivo) cuya tabla de verdad es:

P	Q	$(P \oplus Q)$
T	T	F
T	F	T
F	T	T
F	F	F

Como se puede notar, la columna $(P \oplus Q)$ corresponde a la valuación de $\neg(P \equiv Q)$. El siguiente teorema muestra que cualquier función booleana puede ser expresada con el conjunto de conectivos $\{\wedge, \vee \neg, \supset, \equiv\}$.

Teorema 3.4 (Expresividad del Conjunto de Conectivos)**.** *Asuma una función f : `Bool`$^n \rightarrow$ `Bool`. Entonces, existe una función g que resulta de la composición de las funciones **neg**$(\cdot)$, **con**$(\cdot)$, **disj**$(\cdot)$, **imp**$(\cdot)$, **equiv**$(\cdot)$ tal que $f(\vec{t}) = g(\vec{t})$ para cualquier vector $\vec{t}$ de n elementos de `Bool`.*

Demostración. Procedemos por inducción en el número de argumentos de f.

x_{n+1}	x_1	x_2	$\cdots$	$\cdots$	x_n	f_1	f_2	$\cdots$	$\cdots$	f_k	f
T	-	-	-	-	-	-	-	-	-	-	-
T	-	-	-	-	-	-	-	-	-	-	-
					$\cdots$						
T	-	-	-	-	-	-	-	-	-	-	-
T	-	-	-	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-	-	-	-
					$\cdots$						
F	-	-	-	-	-	-	-	-	-	-	-
F	-	-	-	-	-	-	-	-	-	-	-

f_1

f_2

Figura 3.1: Representación de una función booleana f de $n+1$ argumentos a partir de funciones de n argumentos. Para n variables proposicionales, tenemos 2^{2^n} diferentes funciones, es decir, $k = 2^{2^n}$. Por otro lado, el número de filas en la tabla con $x_{n+1} = \mathtt{T}$ es igual a 2^n al igual que el número de filas donde $x_{n+1} = \mathtt{F}$.

- **Caso $i = 1$**. Podemos entonces construir 4 diferentes funciones:

P	f_1	f_2	f_3	f_4
T	T	T	F	F
F	T	F	T	F

  La función $f_1(P)$ la podemos reemplazar por $g(P) = \mathtt{disj}(P, \mathtt{neg}(P))$ (que corresponde a la fórmula $(P \vee \neg P)$. $f_2(P)$ se puede reemplazar por la identidad $g(P) = P$. f_3 corresponde a la negación de P, es decir, $g(P) = \mathtt{neg}(P)$. Finalmente, f_4 se puede reemplazar por $g(P) = \bot$.

- **Caso $i = n$**. Asumimos, por inducción, que para toda función $f' : \mathtt{Bool}^m \to \mathtt{Bool}$ donde $m < n$ podemos encontrar g' tal que $f' = g'$. La función f de aridad n debe tener en cuenta dos casos:

$$f(x_1, ..., x_{n-1}, \mathtt{T}) \quad \text{y} \quad f(x_1, ..., x_{n-1}, \mathtt{F})$$

  Es decir, cuando el último argumento es `T` y cuando es `F`. Entonces, los resultados cuando $x_n = \mathtt{T}$ se deben poder expresar como una función $f_1(x_1, ..., x_{n-1})$ y los de $x_n = \mathtt{F}$ como una función $f_1(x_1, ..., x_{n-1})$ (ver Figura 3.1). Es decir :

$$f(x_1, ..., x_{n-1}, \mathtt{T}) = f_1(x_1, ..., x_{n-1})$$
$$f(x_1, ..., x_{n-1}, \mathtt{F}) = f_2(x_1, ..., x_{n-1})$$

  Por inducción sabemos que f_1 y f_2 se pueden expresar como las funciones g_1 y g_2 en términos de $\mathtt{neg}(\cdot)$, $\mathtt{con}(\cdot)$, $\mathtt{disj}(\cdot)$, $\mathtt{imp}(\cdot)$ y $\mathtt{equiv}(\cdot)$. La función f puede ser entonces interpretada como

$$g(x_1, ..., x_{n-1}, x_n) = \mathtt{disj}(\mathtt{con}(x_n, g_1(x_1, ..., x_{n-1})), \mathtt{con}(\mathtt{neg}(x_n), g_2(x_1, ..., x_{n-1})))$$

□

De acuerdo con la prueba del anterior teorema, parece que no necesitamos todos los conectivos para expresar cualquier función booleana. En particular, en la prueba solo tuvimos en cuenta $\mathtt{con}(\cdot), \mathtt{disj}(\cdot)$ y $\mathtt{neg}(\cdot)$.

Corolario 3.1. *El conjunto de conectivos* $\{\wedge, \vee, \neg\}$ *es funcionalmente completo.*

Por otro lado, las equivalencias en el Lema 3.2 nos permiten encontrar otros conjuntos de conectivos funcionalmente completos.

Corolario 3.2. *Los siguientes conjuntos de conectivos son funcionalmente completos.*

1. $\{\vee, \neg\}$
2. $\{\wedge, \neg\}$
3. $\{\supset, \neg\}$
4. $\{\supset, \bot\}$

Demostración. Directamente de las equivalencias en el Lema 3.2 y el Teorema 3.4. □

Note que la conjunción por si sola no puede expresar la negación:

Observación 3.1. *Asuma que tiene las constantes $\bot$ y $\top$. Esta última se evalúa siempre a* **T***. Si queremos expresar la función* **neg**(P) *a partir de dichas constantes y la conjunción solamente tenemos dos posibilidades:*

1. **neg**$(P) = (\bot \wedge P)$ *o*
2. **neg**$(P) = (\top \wedge P)$ *o*

Es fácil ver que en el primer caso siempre obtenemos **F** *y el segundo caso corresponde a la función identidad.*

De la misma manera se puede probar que $\{\vee\}$ no es funcionalmente completo.

3.2.1. Álgebras Boolenas

En esta sección probaremos algunas otras equivalencias en lógica proposicional. En particular, probaremos que sus conectivos se comportan como un álgebra booleana.

Lema 3.3 (Propiedades de LP)**.** *Para toda fórmula F, G, H se cumple lo siguiente:*

- *Asociatividad:*
  - $\models ((F \vee G) \vee H) \equiv (F \vee (G \vee H))$ [1].
  - $\models ((F \wedge G) \wedge H) \equiv (F \wedge (G \wedge H))$
- *Conmutatividad:*
  - $\models (F \vee G) \equiv (G \vee F)$
  - $\models (F \wedge G) \equiv (G \wedge F)$
- *Distributividad:*
  - $\models (F \wedge (G \vee H)) \equiv ((F \wedge G) \vee (F \wedge H))$
  - $\models (F \vee (G \wedge H)) \equiv ((F \vee G) \wedge (F \vee H))$
- *Reglas de De Morgan*
  - $\models \neg(F \vee G) \equiv (\neg F \wedge \neg G)$
  - $\models \neg(F \wedge G) \equiv (\neg F \vee \neg G)$
- *Idempotencia*
  - $\models (F \vee F) \equiv F$
  - $\models (F \wedge F) \equiv F$

[1] Recuerde que $\equiv$ induce una relación de equivalencia entre fórmulas (ver Definición 3.8).

- *Doble negación*
  - $\models \neg\neg F \equiv F$
- *Absorción*
  - $\models (F \vee (F \wedge G)) \equiv F$
  - $\models (F \wedge (F \vee G)) \equiv F$
- *Elementos neutro y complementos* [2]
  - $\models (F \vee \bot) \equiv F$ *y* $\models (F \wedge \bot) \equiv \bot$
  - $\models (F \vee \top) \equiv \top$ *y* $\models (F \wedge \top) \equiv F$
  - $\models (F \vee \neg F) \equiv \top$ *y* $\models (F \wedge \neg F) \equiv \bot$

Definición 3.9 (Álgebra Booleana)**.** *Un álgebra booleana es una estructura* $\langle A, \otimes, \oplus, \sim, 0, 1\rangle$ *donde* $\otimes$ *y* $\oplus$ *son operaciones binarias,* $\sim$ *es una operación unaria y* $0, 1 \in A$*;* $\otimes$ *y* $\oplus$ *son asociativos y conmutativos;* $\otimes$ *distribuye sobre* $\oplus$ *y viceversa; para todo* $a \in A$*,* $a \otimes \sim a = 0$*,* $a \oplus \sim a = 1$ *;* $a \oplus 0 = a$ *y* $a \otimes 1 = a$*.*

Teorema 3.5. *La estructura* $\langle\{\mathtt{F}, \mathtt{T}\}, \wedge, \vee, \neg, \mathtt{F}, \mathtt{T}\rangle$ *es un álgebra booleana.*

Demostración. Inmediato de las propiedades en Lema 3.3. □

3.3. Ejercicios

Ejercicio 3.1. *Para cada una de las siguientes fórmulas diga si es una tautología, si es satisfacible o si es insatisfacible.*

1. $((P \wedge Q) \supset P)$
2. $((P \supset (P \supset Q)) \supset P)$.
3. $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$
4. $((P \supset Q) \supset (Q \supset P))$

Dado Γ *y* F *a continuación, muestre si* $\Gamma \models F$ *o* $\Gamma \not\models F$*:*

1. $\Gamma = \{\neg Q, (P \supset Q)\}$, $F = \neg P$.
2. $\Gamma = \{Q, (P \supset Q)\}$, $F = P$.
3. $\Gamma = \{P \supset Q\}$, $F = (\neg Q \supset \neg P)$.

Ejercicio 3.2 (Función NAND)**.** *Defina la función* $\mathtt{nand} : (\mathtt{Bool} \times \mathtt{Bool}) \to \mathtt{Bool}$ *como:*

P	Q	$\mathtt{nand}(P, Q)$
$\mathtt{T}$	$\mathtt{T}$	$\mathtt{F}$
$\mathtt{T}$	$\mathtt{F}$	$\mathtt{T}$
$\mathtt{F}$	$\mathtt{T}$	$\mathtt{T}$
$\mathtt{F}$	$\mathtt{F}$	$\mathtt{T}$

Dicha función corresponde al conectivo $\overline{\wedge}$ *(conjunción negada). Demuestre que* $\{\overline{\wedge}\}$ *es funcionalmente completo.*

[2] $\top$ es una constante que siempre se evalúa a $\mathtt{T}$.

Ejercicio 3.3. *Utilice las equivalencias lógicas vistas en este capítulo para encontrar una fórmula equivalente que solo contenga los conectivos que se especifican en cada caso.*

1. *Utilizando solamente* $\{\wedge,\ neg\}$
   - $((P \supset (Q \vee \neg(R \wedge \neg T))) \supset T)$
   - $(P \vee \neg(Q \vee (T \supset \neg R)))$
2. *Utilizando solamente* $\{\supset, \perp\}$
   - $(P \wedge (Q \vee (R \equiv T))$
   - $\neg(P \wedge \neg(R \vee \neg T))$
   - $(P \wedge Q) \equiv (Q \wedge P)$

Ejercicio 3.4. *Una fórmula se encuentra en Forma Normal Disyuntiva (FND) si es de la forma*

$$c_1 \vee c_2 \vee ... \vee c_n$$

donde cada c_i *es una conjunción de* literales [3]

1. *Demuestre que para toda fórmula* F *existe* F' *en FND tal que* $F \equiv F'$.
2. *Escriba un algoritmo que decida en tiempo lineal si una fórmula en FND es satisfacible.*

Ejercicio 3.5 (Semántica de lenguajes de programación)**.** *Vamos a construir programas en un lenguaje imperativo de programación muy simple. Definimos los programas de acuerdo a la siguiente sintaxis:*

$$\begin{array}{lcl} Z_{exp} & ::= & n \mid x \mid Z_{exp} + Z_{exp} \mid Z_{exp} \times Z_{exp} \mid Z_{exp} - Z_{exp} \mid Z_{exp} \div Z_{exp} \\ B_{exp} & ::= & \texttt{true} \mid \texttt{false} \mid Z_{exp} = Z_{exp} \mid Z_{exp} \leq Z_{exp} \mid \texttt{not}\ B_{exp} \mid B_{exp} \wedge B_{exp} \\ S_{exp} & ::= & x := Z_{exp} \mid \texttt{skip} \mid S_{exp}; S_{exp} \mid \texttt{if}\ B_{exp}\ \texttt{then}\ \{\ S_{exp}\ \} \end{array}$$

En las expresiones anteriores, n *puede ser cualquier número entero y* x *cualquier identificador de variable.* Z_{exp} *corresponde a expresiones aritméticas definidas sobre los enteros.* B_{exp} *corresponde a expresiones booleanas y* S_{exp} *a las sentencias que se pueden construir en el lenguaje. Vamos a utilizar* a, a_i *para denotar expresiones tipo* Z_{exp}, b, b_i *para expresiones tipo* B_{exp} *y* S, S_i *para expresiones tipo* S_{exp}.

Propiedades sintácticas

1. *Demuestre que toda expresión booleana está inmediatamente precedida de la palabra reservada* ***if***.
2. *Demuestre que en todo programa el número de "{" corresponde al número de "}". Además, el número de "{" es igual al número de "**if**" que aparecen en el programa.*

Propiedades semánticas *Vamos a definir una configuración de la siguiente manera:*

$$\langle S\ ;\ \{x_1 \mapsto v_1, \cdots, x_n \mapsto v_n\}\rangle$$

El primer componente representa una sentencia de un programa. El segundo componente representa el estado actual de las variables del programa (e.g., x_i *tiene el valor* v_i*). Vamos a utilizar* V *para denotar el segundo componente en una configuración.*

[3] Un literal es una variables proposicional P o su negación $\neg P$.

Para mostrar formalmente la ejecución de un programa, y razonar sobre su comportamiento, se definen transiciones entre configuraciones de la siguiente manera:

skip :	$\langle \texttt{skip} \; ; \; V\rangle \longrightarrow \langle \epsilon \; ; \; V\rangle$
asign :	$\langle x_i := a \; ; \; \{x_1 \mapsto v_1, \cdots, x_i \mapsto v_i, \cdots, x_n \mapsto v_n\}\rangle \longrightarrow \langle \epsilon \; ; \; \{x_1 \mapsto v_1, ...x_i \mapsto eval_n(V, a), ..., x_n \mapsto v_n\}\rangle$
comp1 :	*if* $\langle S_1 \; ; \; V\rangle \longrightarrow \langle S_1' \; ; \; V'\rangle$ *entonces* $\langle S_1; S_2 \; ; \; V\rangle \longrightarrow \langle S_1'; S_2 \; ; \; V'\rangle$
comp2 :	*if* $\langle S_1 \; ; \; V\rangle \longrightarrow \langle \epsilon \; ; \; V'\rangle$ *entonces* $\langle S_1; S_2 \; ; \; V\rangle \longrightarrow \langle S_2 \; ; \; V'\rangle$
if1 :	*if* $eval_b(V, b) = \texttt{T}$ *entonces* $\langle \texttt{if } b \texttt{ then } \{ \; S \; \} \; ; \; V\rangle \longrightarrow \langle S \; ; \; V\rangle$
if2 :	*if* $eval_b(V, b) = \texttt{F}$ *entonces* $\langle \texttt{if } b \texttt{ then } \{ \; S \; \} \; ; \; V\rangle \longrightarrow \langle \epsilon \; ; \; V\rangle$

En las anteriores expresiones, ϵ *denota que no hay ninguna sentencia por ejecutar. Una sentencia* ***skip*** *no modifica la asignación de las variables y reduce a* ϵ*.*

En una asignación $x := a$*, se cambia el valor de* x *por su nuevo valor* $eval_n(V, a)$*. Por ejemplo*

$$\langle x := 3 + 2 \; ; \; \{x \mapsto 1, y \mapsto 0\}\rangle \longrightarrow \langle \epsilon \; ; \; \{x \mapsto 5, y \mapsto 0\}\rangle$$

Para una composición de sentencias de la forma $S_1; S_2$ *tenemos dos casos: Si* S_1 *reduce a* S_1' *entonces* $S_1; S_2$ *reduce a* $S_1'; S_2$*. Por el contrario, si* S_1 *reduce a* ϵ*, entonces* $S_1; S_2$ *reduce a* S_2*. Note que en este último caso la valuación de las variables* V' *depende de la reducción de* S_1*. Por ejemplo:*

$$\langle x := 3 + 2; y := x \; ; \; \{x \mapsto 1, y \mapsto 0\}\rangle \longrightarrow \langle y := x \; ; \; \{x \mapsto 5, y \mapsto 0\}\rangle \longrightarrow \langle \epsilon \; ; \; \{x \mapsto 5, y \mapsto 5\}\rangle$$

Finalmente, si la expresión booleana b *se evalúa a* `T` *como por ejemplo en* $eval_b(\{x = 1, y = 3\}, x \leq y)$*, una sentencia de la forma* $\texttt{if } b \texttt{ then } \{ \; S \; \}$ *reduce a* S*. Si* $eval_b(V, b) = \texttt{F}$*, entonces la sentencia condicional reduce a* ϵ*.*

1. *Primero debemos definir de manera precisa las funciones de evaluación:*

   a) *Defina de manera inductiva la función* $eval_n : (Vars \to Z) \to Z_{exp} \to Z$*.*

   b) *Defina de manera inductiva la función* $eval_b : (Vars \to Z) \to B_{exp} \to \texttt{Bool}$*.*

2. *Asuma el siguiente programa :*

```
S =
  x:=5;
  if y <= x then { y:=3;x:=1 } ;
  x:=x+1;
  skip
```

   Muestre paso a paso como se evalúa el programa a partir de la configuración $\langle S \; ; \; \{x \mapsto 0, y \mapsto 0\}\rangle$ *hasta llegar a una configuración de la forma* $\langle \epsilon \; ; \; V\rangle$*.*

3. *Ahora vamos a probar una propiedad muy importante de este lenguaje conocida como* **determinismo***, es decir, cualquier evaluación de un programa conduce al mismo resultado* V*. Para ello procedemos de la siguiente forma:*

   a) *Pruebe que en cualquier transición de la forma* $\langle S \; ; \; V\rangle \longrightarrow \langle S' \; ; \; V'\rangle$*, si* $\langle S \; ; \; V\rangle \longrightarrow \langle S'' \; ; \; V''\rangle$ *entonces* $S' = S''$ *y* $V' = V''$*.*

   b) *Ahora asuma una transición de longitud* n *de la forma:*

$$\langle S_1 \; ; \; V_1\rangle \longrightarrow \langle S_2 \; ; \; V_2\rangle \longrightarrow ... \longrightarrow \langle \epsilon \; ; \; V_n\rangle$$

   donde $V_1 = \{x_1 \mapsto 0, x_2 \mapsto 0, ..., x_m \mapsto 0\}$*. Es decir, todas las variables inician en* 0 *y el programa termina con una valuación* V_n *después de* n *reducciones. Demuestre (con la ayuda del punto anterior) que si todas las variables se inician en* 0*, entonces el resultado final del programa* S_1 *siempre será* V_n*.*

4. Finalmente vamos a probar que todo programa en este lenguaje **termina**, *es decir, después de un número finito de reducciones se llega a una configuración $\langle \epsilon \ ; \ V \rangle$ para algún V. Para esto, procedemos de la siguiente manera:*

a) *Defina (de manera inductiva) una "medida" $M : S_{exp} \to N$ de la longitud de los programas.*

b) *Demuestre que si $\langle S \ ; \ V \rangle \longrightarrow \langle S' \ ; \ V' \rangle$ entonces $M(S') < M(S)$.*

c) *Explique por qué los dos argumentos anteriores son suficientes para mostrar que todo programa en este lenguaje termina.*

Capítulo 4

Procedimientos de Decisión para Lógica Proposicional

En este capítulo exploraremos algoritmos para decidir si una fórmula en lógica proposicional es válida o no. Primero exploraremos el Cálculo de Secuentes à la Gentzen y probaremos su correctitud y completitud. Luego, con ayuda de un asistente de pruebas (Coq) probaremos algunos resultados para lógica proposicional. También estudiaremos el procedimiento de resolución para LP y su importancia en computación. Al final discutiremos acerca de la importancia del problema de satisfacción de fórmulas proposicionales para la teoría de la complejidad.

4.1. Sistemas à la Hilbert

Hasta este punto hemos probado la validez (o no) de una fórmula en LP mediante el uso de tablas de verdad. Este método no es muy eficiente puesto que para n variables proposicionales, debemos calcular 2^n filas en la tabla de verdad.

Para establecer una prueba que cierta fórmula se deduce de un conjunto de hipótesis, debemos proveer un conjunto de axiomas (fórmulas que son universalmente válidas) y un conjunto de reglas de inferencia que nos permiten manipular los símbolos en la prueba. Por ejemplo, en el sistema de deducción de Hilbert para lógica proposicional, tenemos tres axiomas y una única regla de inferencia:

- **Axiomas**:

  A1: $(P \supset (Q \supset P))$.
  A2: $((P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R)))$.
  A3: $(\neg P \supset \neg Q) \supset ((\neg P \supset Q) \supset P)$.

- **Reglas de inferencia**

  MP: –Modus ponens–: De P y $(P \supset Q)$ se puede inferir B.

Las pruebas en este sistema tienden a ser muy largas y se debe ser muy creativo para instanciar los axiomas. En este curso utilizaremos los el cálculo de secuentes para realizar las pruebas.

4.2. Cálculo de Secuentes

El cálculo de secuentes es otro sistema deductivo en donde se tienen pocos axiomas pero se cuenta con reglas explícitas para manipular cada uno de los símbolos lógicos durante la prueba.

Suponga que se quiere probar la siguiente fórmula es una tautología:[1]

$$(P \wedge (P \supset Q)) \supset Q$$

De acuerdo con el Lema 3.1, si queremos probar que F es una tautología, una estrategia válida es preguntarnos si $\neg F$ es insatisfacible. Por ahora, ignorando el significado del símbolo "$\longrightarrow$", ubicamos la fórmula que queremos probar al lado derecho:

$$\longrightarrow (P \wedge (P \supset Q)) \supset Q$$

Lo que está al lado derecho de "$\longrightarrow$" es lo que queremos hacer falso y al lado izquierdo lo que queremos hacer verdadero. Así que, para que una implicación $F \supset G$ sea falsa, F debemos hacerla verdadera y G debemos hacerla falsa:

$$\dfrac{(P \wedge (P \supset Q) \longrightarrow Q}{\longrightarrow (P \wedge (P \supset Q)) \supset Q}$$

Ahora, para hacer verdadera la fórmula $P \wedge (P \supset Q)$, ambos, P y $(P \supset Q)$ deben ser verdaderos:

$$\dfrac{\dfrac{P, P \supset Q \longrightarrow Q}{(P \wedge (P \supset Q) \longrightarrow Q}}{\longrightarrow (P \wedge (P \supset Q)) \supset Q}$$

Para hacer verdadera a $P \supset Q$, debemos tomar dos caminos: hacer P falsa o Q verdadera:

$$\dfrac{\dfrac{\dfrac{P, \neg P \longrightarrow Q \qquad P, Q \longrightarrow Q}{P, P \supset Q \longrightarrow Q}}{(P \wedge (P \supset Q) \longrightarrow Q}}{\longrightarrow (P \wedge (P \supset Q)) \supset Q}$$

En la rama del lado izquierdo, note que es imposible hacer verdadero a P y a $\neg P$ (la fórmula $P \wedge \neg P$ es insatisfacible). Al lado derecho, si Q (del lado izquierdo) es verdadero, no es posible que Q (al lado derecho) sea falso. En estos casos decimos que las ramas del secuente están cerradas y hemos encontrado una prueba de la validez de la fórmula.

A continuación formalizamos los pasos que acabamos de hacer.

Definición 4.1 (Cálculo de Secuentes)**.** *A continuación utilizaremos* Δ, Γ *para denotar conjuntos de propo-*

[1]De ahora en adelante omitiremos los paréntesis que no sean necesarios.

siciones. Las reglas de inferencia del cálculo de secuentes son las siguientes [NvP01]: [2]

Axioma

$$\Gamma, F \longrightarrow \Delta, F$$

Reglas de Inferencia

$$\frac{\Gamma, F, G \longrightarrow \Delta}{\Gamma, F \wedge G \longrightarrow \Delta}\ \wedge_I \qquad \frac{\Gamma \longrightarrow \Delta, F \qquad \Gamma \longrightarrow \Delta, G}{\Gamma \longrightarrow \Delta, F \wedge G}\ \wedge_D$$

$$\frac{\Gamma, F \longrightarrow \Delta \qquad \Gamma, G \longrightarrow \Delta}{\Gamma, F \vee G \longrightarrow \Delta}\ \vee_I \qquad \frac{\Gamma \longrightarrow \Delta, F, G}{\Gamma \longrightarrow \Delta, F \vee G}\ \vee_D$$

$$\frac{\Gamma \longrightarrow \Delta, F \qquad \Gamma, G \longrightarrow \Delta}{\Gamma, F \supset G \longrightarrow \Delta}\ \supset_I \qquad \frac{\Gamma, F \longrightarrow \Delta, G}{\Gamma \longrightarrow \Delta, F \supset G}\ \supset_D$$

$$\frac{\Gamma \longrightarrow \Delta, F}{\Gamma, \neg F \longrightarrow \Delta}\ \neg_I \qquad \frac{\Gamma, F \longrightarrow \Delta}{\Gamma \longrightarrow \Delta, \neg F}\ \neg_D$$

$$\frac{}{\Gamma, \bot \longrightarrow \Delta}\ \bot_I$$

Veamos algunos ejemplos.

Ejemplo 4.1 (Secuentes)**.** *Primero probemos que* $(P \vee Q) \equiv (Q \vee P)$

$$\frac{\dfrac{\dfrac{\dfrac{P \longrightarrow Q, P \qquad Q \longrightarrow Q, P}{P \vee Q \longrightarrow Q, P}\ \vee_I}{P \vee Q \longrightarrow Q \vee P}\ \vee_D}{\longrightarrow (P \vee Q) \supset (Q \vee P)}\ \supset_D \qquad \dfrac{\dfrac{\dfrac{Q \longrightarrow P, Q \qquad P \longrightarrow P, Q}{Q \vee P \longrightarrow P, Q}\ \vee_I}{Q \vee P \longrightarrow P \vee Q}\ \vee_D}{\longrightarrow (Q \vee P) \supset (P \vee Q)}\ \supset_D}{\longrightarrow ((P \vee Q) \supset (Q \vee P)) \wedge ((Q \vee P) \supset (P \vee Q))}\ \wedge_D$$

Ahora probemos que $(P \supset Q) \longrightarrow (\neg Q \supset \neg P)$*:*

$$\frac{\dfrac{\dfrac{\dfrac{P \longrightarrow P, Q \qquad Q, P \longrightarrow Q}{P \supset Q, P \longrightarrow Q}\ \supset_I}{P \supset Q, \longrightarrow \neg P, Q}\ \neg_D}{P \supset Q, \neg Q \longrightarrow \neg P}\ \neg_I}{P \supset Q \longrightarrow \neg Q \supset \neg P}\ \supset_D$$

Finalmente, probamos que $P \wedge (P \supset Q) \wedge (Q \supset R) \longrightarrow P \wedge R$ [3]

$$\frac{\dfrac{P, Q \supset R \longrightarrow P, P \wedge R \qquad \dfrac{P, Q \longrightarrow Q, P \wedge R \qquad \dfrac{\dfrac{P, Q, R \longrightarrow P \qquad P, Q, R \longrightarrow R}{P, Q, R \longrightarrow P \wedge R}\ \wedge_D}{}}{P, Q, Q \supset R \longrightarrow P \wedge R}\ \supset_I}{P, (P \supset Q), (Q \supset R) \longrightarrow P \wedge R}\ \supset_I}{P \wedge (P \supset Q) \wedge (Q \supset R) \longrightarrow P \wedge R}\ \wedge_I$$

[2] La regla $\bot_I$ no es realmente necesaria puesto que $\bot \equiv P \wedge \neg P$

[3] Vamos a utilizar doble barra cuando aplicamos varias reglas al tiempo.

Asuma que no es posible probar cierta fórmula F a partir de un conjunto de fórmulas Γ, es decir, $\Gamma \not\longrightarrow F$. El procedimiento que acabamos de utilizar en el ejemplo anterior también nos permite mostrar una posible valuación que hace verdaderas las fórmulas en Γ y que hace falsa a F. Esto sucede, en el caso de la lógica proposicional, cuando alguna de las hojas del árbol de la derivación contienen solamente variables proposicionales (no símbolos lógicos) y no se puede aplicar la regla **axioma**.

Ejemplo 4.2 (Refutación). *Tratemos de probar lo siguiente:*

$$\dfrac{\dfrac{?}{\longrightarrow P, P} \qquad \dfrac{?}{Q \longrightarrow P}}{P \supset Q \longrightarrow P} \supset_I$$

En las hojas del árbol no podemos aplicar ninguna de las reglas. Considere la valuación $v = \{P \mapsto \mathtt{F}, Q \mapsto \mathtt{T}\}$. *Escogemos esta valuación porque* P *aparece al lado derecho y* Q *al lado izquierdo. Note que* $[\![P \supset Q]\!]_v = \mathtt{T}$ *y* $[\![P]\!]_v = \mathtt{F}$. *Así que el secuente* $P \supset Q \longrightarrow P$ **NO** *es probable y encontramos una valuación que lo hace falso.*

4.3. Asistentes de Prueba

Las estrategias que hemos utilizado pueden ser automatizadas por un probador de teoremas. En este curso utilizaremos Coq `http://coq.inria.fr` para facilitar la prueba de nuestros teoremas. La documentación completa de Coq se puede encontrar en `http://coq.inria.fr/documentation`.

Ejemplo 4.3 (Declaraciones). *Para probar teoremas simples de la lógica proposicional, primero declaramos algunas variables de tipo* `Prop` *y luego escribimos el teorema que queremos probar.*

```
(* Primera prueba en Coq *)
Section Primer_Programa.
  Variables P Q : Prop.
  Theorem T1: P->P.
    intro H.
    assumption.
  Qed.
End Primer_Programa.
```

En la anterior prueba, se utilizaron dos tácticas:

- `intro`*: Funciona como la regla* $\supset R$
- `assumption`*: Funciona como la regla* **Axioma**.

Ejemplo 4.4 (Conectivos). *Ahora utilicemos otros conectivos:*

```
Section Conectivos.
 Variable P Q:Prop.
  Theorem Conmutatividad: (P/\Q) <-> (Q /\ P).
  split.
  intro H.
  destruct H as [H1 H2].
  split.
  assumption.
  assumption.
  (* Ahora la segunda implicacion *)
  intro H; destruct H as [H1 H2];split; assumption.
Qed.
 Theorem And_Or: (P/\Q) -> (P \/ Q).
```

```
  intro H.
  destruct H as [H1 H2].
  right;assumption.
Qed.
 Theorem weak_peirce : ((((P->Q)->P)->P)->Q)->Q.
  intro H; apply H.
  intro H1; apply H1.
  intro H2; apply H.
  intro H3;assumption.
Qed.
End Conectivos.
```

Las tácticas que fueron utilizadas en las anteriores pruebas se resumen en el Cuadro 4.1 (ver [Ber10] y [BC04] para más detalles).

Es posible también probar secuentes de la forma $\Gamma \longrightarrow F$. Para esto, se deben adicionar como hipótesis las fórmulas en Γ y luego establecer el teorema $\longrightarrow F$. Por ejemplo, la prueba de:

$$P \supset Q \longrightarrow (P \wedge (Q \supset R)) \supset R$$

se puede obtener de la siguiente manera:

```
(* Adicionando hipotesis *)
Section Add_Hipotesis.
 Variables P Q R :Prop.
 Hypothesis H1: P -> Q.
 Theorem Th1: (P /\ (Q->R)) -> R.
   intro H2.
   destruct H2 as [H3 H4].
```

Conectivo	**Tácticas**
$H : A \supset B \longrightarrow \Gamma$	apply H (B debe unificar con el goal Γ)
$\Gamma \longrightarrow A \supset B$	intro. | intro H. | intros. | intros H1 H2 ....
$H : A \wedge B \longrightarrow \Gamma$	destruct H as [H1 H2].
$\Gamma \longrightarrow A \wedge B$	split.
$H : A \vee B \longrightarrow \Gamma$	destruct H as [H1 — H2].
$\Gamma \longrightarrow A \vee B$	right. | left.
$\Gamma \longrightarrow \neg A$	intro. | intro H.
$H : \neg A \longrightarrow \Gamma$	unfold not in H.

Algunas tácticas de uso general:

Conectivo	**Tácticas**
assert A.	Introduce A como una hipótesis (la cual debe ser probada).
contradiction.	Si se tienen como hipótesis F y $\neg F$ termina la prueba.
absurd F.	Introduce el nuevo goal $F \wedge \neg F$.
exact (H1 H2).	Si $H1 : A \supset B$ y $H2 : A$ y el goal es B, entonces aplica $H1$ en $H2$ para producir una prueba de B.
assumption.	Si se tiene como hipótesis $H : F$ y el goal es F termina la prueba.
tac1 ; tac2	Aplica la táctica tac1 y la táctica tac2 a todos los subgoals que se deriven.

Cuadro 4.1: Tácticas de prueba en Coq

```
   apply H4;apply H1; assumption.
Qed.
End Add_Hipotesis.
```

Note que también hubiera podido probar $\longrightarrow (P \supset Q) \supset (P \wedge (Q \supset R)) \supset R$. Esto lo formalizamos a continuación.

Teorema 4.1. *Asuma un conjunto de fórmulas* $\Gamma = \{G_1, G_2, \cdots, G_n\}$. *El secuente* $\Gamma \longrightarrow F$ *es probable sii el secuente* $\longrightarrow \bigwedge_i G_i \supset F$ *es probable.*

Demostración. Debemos probar dos cosas:

1. $(\Rightarrow)$. Asuma que $\Gamma \longrightarrow F$ es probable. Por la regla $\supset_D$ y $\wedge_I$ tenemos la siguiente derivación:

$$\dfrac{\dfrac{\overset{\Xi}{G_1, \cdots, G_n \longrightarrow F}}{\bigwedge_i G_i \longrightarrow F}\wedge_I}{\longrightarrow \bigwedge_i G_i \supset F}\supset_D$$

   Por hipótesis $\Gamma \longrightarrow F$ es probable y por tanto, Ξ es una prueba.

2. Ahora asuma que $\longrightarrow \bigwedge_i G_i \supset F$ es probable. Como el conectivo principal es $\supset$ en dicho secuente, la única posible derivación debe ser:

$$\dfrac{\bigwedge_i G_i \longrightarrow F}{\longrightarrow \bigwedge_i G_i \supset F}\supset_D$$

   Aplicando $\wedge_I$ podemos concluir que $\Gamma \longrightarrow F$ [4].

□

4.4. Lógica Clásica y Lógica Intuicionista

Asuma que queremos probar que existen dos números irracionales a y b tal que a^b es un número racional. Podríamos proceder de la siguiente manera. Tome $a = b = \sqrt{2}$. Sabemos que $\sqrt{2}$ es un número irracional. Entonces tenemos dos posibilidades:

1. Si $\sqrt{2}^{\sqrt{2}}$ es un número racional entonces ya encontramos a y b irracionales tal que a^b es un racional.

2. Si $\sqrt{2}^{\sqrt{2}}$ no es un número irracional, entonces tome $a = \sqrt{2}^{\sqrt{2}}$ y $b = \sqrt{2}$. Entonces $a^b = \sqrt{2}^2 = 2$, un número racional y concluimos que $a = \sqrt{(2)}^{\sqrt{(2)}}$ y $b = \sqrt{2}$ son los número que estábamos buscando.

En la prueba anterior utilizamos el principio de " Principio del tercero excluido": Toda proposición es cierta o falsa así no tengamos una prueba de ello ($P \vee \neg P$).

La lógica intuicionista no asume tal principio y, en general, $P \vee \neg P$ no es probable. A continuación presentamos el cálculo de secuentes para lógica intuicionista.

[4] De hecho la prueba de esta parte requiere utilizar el principio de invertibilidad de la regla $\wedge_I$ [NvP01].

Definición 4.2 (Sistema para lógica intuicionista [NvP01]). *Asuma que* Γ *es un conjunto de fórmulas. Las siguientes son las reglas para el cálculo de lógica intuicionista.*

Axioma

$$\Gamma, F \Longrightarrow F$$

Reglas de Inferencia

$$\frac{\Gamma, G, G' \Longrightarrow F}{\Gamma, G \wedge G' \Longrightarrow F}\wedge_I \qquad \frac{\Gamma \Longrightarrow F \qquad \Gamma \Longrightarrow G}{\Gamma \Longrightarrow F \wedge G}\wedge_D$$

$$\frac{\Gamma, G \Longrightarrow F \qquad \Gamma, G' \Longrightarrow F}{\Gamma, G \vee G' \Longrightarrow F}\vee_I$$

$$\frac{\Gamma \Longrightarrow F}{\Gamma \Longrightarrow F \vee G}\vee_{D_D} \qquad \frac{\Gamma \Longrightarrow G}{\Gamma \Longrightarrow F \vee G}\vee_{D_D}$$

$$\frac{\Gamma, G \supset G' \Longrightarrow G \qquad \Gamma, G' \Longrightarrow F}{\Gamma, G \supset G' \Longrightarrow F}\supset_I \qquad \frac{\Gamma, F \Longrightarrow G}{\Gamma \Longrightarrow F \supset G}\supset_D$$

$$\frac{}{\Gamma, \bot \Longrightarrow \Delta}\bot_I$$

Note que en el sistema anterior, al lado derecho solo hay una fórmula y no un conjunto de fórmulas como en el sistema de la Definición 4.1. Por otro lado, la regla de la disyunción a la derecha debe escoger alguna de las fórmulas en la disyunción. Una vez se escoge una de las subfórmulas, la otra alternativa se descarta.

Ejemplo 4.5. *Utilizando el sistema de la Definición 4.1, podemos probar lo siguiente:*

$$\dfrac{\dfrac{P \longrightarrow P}{\longrightarrow P, \neg P}\neg_D}{\longrightarrow P \vee \neg P}\vee_D$$

Sin embargo, utilizando el sistema de la Definición 4.2 tenemos dos opciones: una utilizando $\vee_{D_D}$*:*

$$\dfrac{\begin{array}{c}\Xi\\ \Longrightarrow \neg P\end{array}}{\Longrightarrow P \vee \neg P}\vee_{D_D}$$

o utilizando $\vee_{D_I}$*:*

$$\dfrac{\begin{array}{c}\Xi'\\ \Longrightarrow P\end{array}}{\Longrightarrow P \vee \neg P}\vee_{D_D}$$

Como se puede apreciar, ni Ξ *ni* Ξ' *son pruebas y por tanto, no podemos probar* $P \vee \neg P$*.*

Asuma que la fórmula F contiene las variables proposicionales $P_1, \cdots, P_n$ y considere la siguiente fórmula:

$$F' = ((P_1 \vee \neg P_1) \wedge (P_2 \vee \neg P_2) \wedge \cdots \wedge (P_n \vee \neg P_n)) \supset F$$

Si F es probable clásicamente (con el sistema de la Definición 4.1), entonces F' es probable con el sistema de la definición 4.2 y F y F' son equivalentes clásicamente. [5]

La anterior construcción nos permite hacer pruebas con el sistema $\Longrightarrow$ adicionando como premisas las fórmulas $(P_i \vee \neg P_i)$.

[5] Note que para cualquier Γ y F, si el secuente $\Gamma \Longrightarrow F$ es probable, entonces $\Gamma \longrightarrow F$ es probable.

Ejemplo 4.6. *Asuma de nuevo la fórmula* $P \vee \neg P$*. Realizando la transformación antes descrita, observamos que el siguiente secuente es probable con la regla* **axioma***:*

$$(P \vee \neg P) \Longrightarrow (P \vee \neg P)$$

Ahora, probemos $\neg\neg P \longrightarrow P$*. Clásicamente, obtenemos la siguiente derivación:*

$$\dfrac{\dfrac{P \longrightarrow P}{\longrightarrow P, \neg P}\,\neg_D}{\neg\neg P \longrightarrow P}\,\neg_I$$

En lógica intuicionista no podemos probarlo. Sin embargo, podemos adicionar como premisa $P \vee \neg P$*:*

$$\dfrac{P, \neg\neg P \Longrightarrow P \qquad \dfrac{\dfrac{P \supset \bot \Longrightarrow P \supset \bot \qquad \dfrac{}{\bot \Longrightarrow P}\,\bot_I}{(P \supset \bot) \supset \bot, P \supset \bot \Longrightarrow P}\,\supset_I}{\neg\neg P, \neg P \Longrightarrow P}\,def}{P \vee \neg P, \neg\neg P \Longrightarrow P}\,\vee_I$$

Finalmente, considere la fórmula $P \supset (\neg Q \vee (P \wedge Q))$*. Clásicamente tenemos lo siguiente:*

$$\dfrac{\dfrac{\dfrac{\dfrac{P, Q \longrightarrow P \qquad P, Q \longrightarrow Q}{P, Q \longrightarrow P \wedge Q}\,\wedge_D}{P \longrightarrow \neg Q, P \wedge Q}\,\neg_I}{P \longrightarrow \neg Q \vee (P \wedge Q)}\,\vee_D}{\longrightarrow P \supset (\neg Q \vee (P \wedge Q))}\,\supset_D$$

Con el cálculo intuicionista, adicionamos las respectivas premisas para poder realizar la prueba [6]*:*

$$\dfrac{\dfrac{\dfrac{P, P \vee \neg P, \neg Q \Longrightarrow \neg Q}{P, P \vee \neg P, \neg Q \Longrightarrow \neg Q \vee (P \wedge Q)}\,\vee_{D_I} \qquad \dfrac{\dfrac{P, P \vee \neg P, Q \Longrightarrow P \qquad P, P \vee \neg P, Q \Longrightarrow Q}{P, P \vee \neg P, Q \Longrightarrow P \wedge Q}\,\wedge_D}{P, P \vee \neg P, Q \Longrightarrow \neg Q \vee (P \wedge Q)}\,\vee_{D_D}}{P, P \vee \neg P, Q \vee \neg Q \Longrightarrow \neg Q \vee (P \wedge Q)}\,\vee_I}{P \vee \neg P, Q \vee \neg Q \Longrightarrow P \supset (\neg Q \vee (P \wedge Q))}\,\supset_D$$

Observación 4.1. *A diferencia del sistema clásico, en lógica intuicionista importa el orden en el cual se aplican las reglas. En particular, la regla de disyunción a la derecha elimina una de las alternativas y por tanto, se debe postergar su uso en la derivación. Por ejemplo, asuma el siguiente secuente en lógica clásica:*

$$\dfrac{\dfrac{\dfrac{P \longrightarrow P, Q, R \qquad \overset{(*)}{P, Q \longrightarrow Q, R}}{P, P \supset Q \longrightarrow Q, R}\,\supset_I \qquad \dfrac{P \longrightarrow P, Q, R \qquad \overset{(**)}{P, R \longrightarrow Q, R}}{P, P \supset R \longrightarrow Q, R}\,\supset_I}{P, (P \supset Q) \vee (P \supset R) \longrightarrow Q, R}\,\vee_I}{P, (P \supset Q) \vee (P \supset R) \longrightarrow Q \vee R}\,\vee_D$$

Note que en la anterior derivación, la escogencia de probar Q *o* R *se mantuvo hasta el final de la prueba. Más específicamente, note que en la aplicación del axioma en la hoja* $(*)$ *se tiene una prueba de* Q *mientras que en la hoja* $(**)$ *se tiene la prueba de* R*.*

Ahora si intentamos hacer la misma prueba con el sistema intuicionista obtenemos lo siguiente:

$$\dfrac{\dfrac{\dfrac{P, P \supset Q \Longrightarrow P \qquad P, Q \Longrightarrow Q}{P, P \supset Q \Longrightarrow Q}\,\supset_I \qquad \dfrac{P, P \supset R \Longrightarrow P \qquad \overset{\texttt{fail!!}}{P, R \Longrightarrow Q}}{P, P \supset R \Longrightarrow Q}\,\supset_I}{P, (P \supset Q) \vee (P \supset R) \Longrightarrow Q}\,\vee_I}{P, (P \supset Q) \vee (P \supset R) \Longrightarrow Q \vee R}\,\vee_{D_D}$$

[6] De hecho, en este caso, con $Q \vee \neg Q$ es suficiente.

En el anterior secuente, la rama más a la izquierda no puede ser probada. Si en vez de utilizar $\vee_{D_D}$ al principio se utiliza $\vee_{D_I}$, la rama más a la derecha (probando R a partir de P y $P \supset Q$) fallaría.

Lo anterior indica que no podemos utilizar la regla de disyunción a la derecha y debemos escoger otro conectivo para iniciar la prueba, en este caso, la disyunción al lado izquierdo:

$$\dfrac{\dfrac{P \Longrightarrow P \qquad \dfrac{P, Q \Longrightarrow Q}{P, Q \Longrightarrow Q \vee R}\,\vee_{I_D}}{P, P \supset Q \Longrightarrow Q \vee R}\,\supset_I \qquad \dfrac{P, P \supset R \Longrightarrow P \qquad \dfrac{P, R \Longrightarrow R}{P, R \Longrightarrow Q \vee R}\,\vee_{I_I}}{P, P \supset R \Longrightarrow Q \vee R}\,\supset_I}{P, (P \supset Q) \vee (P \supset R) \Longrightarrow Q \vee R}\,\vee_I$$

En el secuente anterior, la escogencia de Q o R a la derecha se postergó hasta después de la escogencia de las implicaciones al lado izquierdo para poder terminar la prueba.

En Coq, la librería "Classical" puede ser utilizada para realizar pruebas que requieran el axioma $P \vee \neg P$. Veamos como se prueban los secuentes del ejercicio anterior.

Ejemplo 4.7 (Pruebas utilizando "Classical"). *En estos ejemplos, la táctica "elim (classic P)" introduce $P \vee \neg P$ como hipótesis en la prueba:*

```
Section Classical.
(* Adicionando la libreria *)
Require Import Classical.

Variable P Q : Prop.

Theorem DobleNegacion : ~~P -> P.
  elim (classic P).
  intros H1 H2; assumption.
  intros H1 H2.
  contradiction.
Qed.

Lemma EjClasic : P -> (~Q \/ (P /\ Q)).
  intro H.
  elim (classic Q).
  intro H1;right;split;assumption.
  intro H1;left;assumption.
Qed.
```

Una manera alternativa es adicionar como hipótesis en la prueba la fórmula $P \vee \neg P$ y probar dicha hipótesis utilizando el axioma "classic":

```
Theorem DobleNegacion' : ~~P -> P.
  assert (P \/ ~P).
  exact (classic P).
  intro H1.
  destruct H as [H2 | H3].
  assumption.
  unfold not in H3.
  unfold not in H1.
  assert False.
  exact (H1 H3).
  contradiction.
Qed.
```

4.5. Resolución en Lógica Proposicional

Resolución es una de las técnicas más populares para probar la validez de una fórmulas. A diferencia de los sistemas $\longrightarrow$ y $\Longrightarrow$, en resolución consideramos sistemas más simples y *orientados por objetivos (goals).*

4.5.1. Programación Lógica y Cláusulas de Horn

Podemos ver un *programa* en lógica proposicional como un conjunto de fórmulas que le dan *significado* a las variables proposicionales (y a los predicados en lógica de primer orden). De estos programas interesa saber si un *objetivo* se puede derivar, es decir, si el objetivo es una conclusión de las reglas del programa. Lo que vamos a hacer entonces es probar el secuente $\Gamma \longrightarrow G$ donde Γ representa el programa y G el objetivo.

Restringiendo el tipo de fórmulas que se pueden utilizar para especificar los programas y los objetivos, podemos utilizar técnicas más apropiadas (eficientes) para probar el secuente $\Gamma \longrightarrow G$.

Definición 4.3 (Programas y Objetivos)**.** *Los programas (D) y objetivos (G) se pueden construir por medio de la siguiente sintaxis:*

$$\begin{array}{lcl} G & ::= & P \mid G \wedge G \\ D & ::= & P \mid G \supset P \mid D \wedge D \end{array}$$

La sintaxis de la definición anterior da lugar a fórmulas que vamos a llamar cláusulas de Horn.

Definición 4.4 (Clásulas de Horn)**.** *Llamamos literal a una variable proposicional o su negación. Una cláusula C es una conjunción de literales de la forma:*

$$C = A_1 \vee A_2 \vee \cdots A_m$$

Una fórmula F está en forma normal conjuntiva (FNC) si es una conjunción de cláusulas:

$$F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$$

A partir de la sintaxis en la Definición 4.3, podemos utilizar el siguiente sistema de pruebas para probar secuentes de la forma $\Gamma \longrightarrow G$.

Definición 4.5 (Sistema $\longrightarrow_p$)**.** *Dada un conjunto de cláusulas Γ y un objetivo G, el secuente $\Gamma \longrightarrow G$ es probable si y solamente si el secuente $\Gamma \longrightarrow_p G$ es probable donde:*

Axioma

$$\frac{}{\Gamma, P \longrightarrow_g P}$$

Reglas de Inferencia

$$\frac{\overline{\Gamma \longrightarrow_p G_1} \quad \overline{\Gamma \longrightarrow_p G_2}}{\Gamma \longrightarrow_p G_1 \wedge G_2}\ \mathtt{split}$$

$$\frac{\overline{\Gamma, G \supset P \longrightarrow_p G}}{\Gamma, G \supset P \longrightarrow_p P}\ \mathtt{apply}$$

Note que en el sistema anterior no se considera la regla $\wedge_I$ puesto que Γ es un conjunto de cláusulas (representando una conjunción de cláusulas). Además, la regla `apply` selecciona la cláusula (regla) $G \supset P$ si el lado derecho de la implicación coincide con el objetivo que se está probando. Por lo tanto, el suencuente en el lado derecho de

$$\frac{\overline{\Gamma, G \supset P \longrightarrow_p G} \qquad \overline{\Gamma, P \longrightarrow_p P}}{\Gamma, G \supset P \longrightarrow_p P}\ \mathtt{apply}$$

es trivialmente probable por la regla axioma y se omite.

El anterior sistema corresponde al modelo de computación de Prolog. En éste, cuando se quiere probar un objetivo, se seleccionan una regla del programa y se adicionan al objetivo los antecedentes de la implicación. A esta técnica se le conoce como SLD resolution (Selective Linear Definite clause resolution). El programa termina con la respuesta `yes` si al final de la derivación puede aplicar la regla axioma que corresponde a encontrar un hecho (`fact`) especificado en el programa.

Notation 4.1 (Notación de Prolog)**.** *Las reglas del programa las escribiremos de la siguiente manera:*

```
p:- q1,q2,...qn
```

Esta regla corresponde a la fórmula (al lado izquierdo del secuente):

$$q_1 \wedge q_2 \wedge \cdots q_n \supset p \equiv (\neg q_1 \vee \cdots \vee \neg q_n \vee p)$$

Un hecho (`fact`), se escribe de la siguiente forma:

```
p.
```

y corresponde a la fórmula $\top \supset p$ al lado izquierdo del secuente. Finalmente, un objetivo (goal) se especifica de la siguiente manera:

```
? p1,p2,...pn
```

que corresponde a la fómrula

$$p_1 \wedge p_2 \wedge \cdots \wedge p_n$$

al lado derecho del secuente

Veamos un ejemplo simple en GNU-Prolog (http://www.gprolog.org).

Ejemplo 4.8 (Prolog)**.** *Primero escribimos un script con las cláusulas:*

```
%%%%%%%%%%%%%%%
% Ejemplo Prolog
% archivo: ejemplo1.pl
%%%%%%%%%%%%%%%

%Reglas
p:- q,r.
r:-q.
% Hechos
q.
```

Ahora cargamos el programa:

```
[ejemplo1].
?- p,q.
yes
```

Es decir, del programa se puede concluir $p \wedge q$.

En el algoritmo de resolución se debe escoger cuál de las reglas del programa aplicar para resolver el objetivo actual. Esto genera un escogencia no determinística. Una búsqueda de una prueba para cierto objetivo corresponde entonces en explorar todas las posibles opciones hasta encontrar una prueba (backchaining).

Ejemplo 4.9 (Prolog)**.** *Asuma que se tiene información de las rutas aéreas entre las ciudades de Colombia. Por ejemplo:*

```
% Rutas
cartagena:- medellin.     %R1
cartagena:- bogota.       %R2
bogota:- cali.            %R3
barranquilla:-  cali.     %R4.
```

Ahora asuma que está en Cali:

```
%Hechos
cali.
```

Podemos probar, por ejemplo, que se puede llegar a Cartagena: [7]

```
 ?- cartagena.
yes
```

Para esta derivación, una posibilidad, puede ser utilizar la cláusula medellin ⊃ cartagena. Sin emabrgo, esto no conduce a una prueba de cartagena:

$$\dfrac{\dfrac{\texttt{fail!}}{\Gamma \longrightarrow_p medellin}}{\Gamma \longrightarrow_p cartagena}\ \texttt{apply R1}$$

El algoritmo SLD intenta entonces con la regla R2:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \longrightarrow_p cali}\ axioma}{\Gamma \longrightarrow_p bogota}\ \texttt{apply R3}}{\Gamma \longrightarrow_p cartagena}\ \texttt{apply R2}$$

4.5.2. Cálculo de secuentes para Fórmulas en FNC

A continuación presentamos una representación alternativa para las fórmulas en forma normal conjuntiva (FNC).

Notation 4.2 (Notación de Cláuluas). *Asuma la siguiente fórmula en FNC:*

$$F = (P \vee \neg Q) \wedge (R \vee Q) \wedge (\neg R \vee P \vee Q) \wedge P$$

Si el literal A es de la forma $\neg P$ lo vamos a representar como $\overline{P}$. Además, las cláusulas las representamos como conjuntos de literales (unidos por disyunciones), y toda la fórmula como un conjunto de conjuntos de literales (unidos por conjunciones):

$$F = \{P, \overline{Q}\}, \{R, Q\}, \{\overline{R}, P, Q\}, P$$

En el último término escribimos P en vez de $\{P\}$.

A continuación definimos un sistema muy simple que solo hace uso de la regla *resolución*.

Definición 4.6 (Sistema GFNC). *Asuma que Γ es un conjunto (conjunción) de cláusulas. El sistema $\longrightarrow_g$ se define como:*

Axioma

$$\dfrac{}{\Gamma, P, \overline{P} \longrightarrow_g}$$

Regla de Inferencia

$$\dfrac{\overline{\Gamma_1, C_1, \cdots, C_m \longrightarrow_g} \quad \overline{\Gamma_2, A \longrightarrow_g}}{\Gamma, \{A_{11}, ..., A_{1k}, A\}, \cdots, \{A_{n1}, \cdots, A_{nl}, A\} \longrightarrow_g}$$

[7] En el siguiente capítulo podremos construir programas más interesantes con la ayuda de predicados y el algoritmo de unificación.

donde A y todo A_{ij} son literales; Γ_1 y Γ_2 son subconjuntos (posiblemente vacíos y no necesariamente disyuntos) del conjunto $\Gamma \cup \{\{A_{11}, ..., A_{1k}, A\}, \cdots, \{A_{n1}, \cdots, A_{nl}, A\}\}$; y $\{C_1, \cdots, C_m\}$ es un subconjunto no vacío de $\{\{A_{11}, ..., A_{1k}\}, \cdots, \{A_{n1}, \cdots, A_{nl}\}\}$.

Ejemplo 4.10. *Considere el programa del Ejemplo 4.9. Con el sistema $\longrightarrow_g$ obtenemos la siguiente derivación:*

$$\dfrac{\dfrac{\dfrac{}{bogota, barranquilla, \overline{bogota}, \overline{medellin} \longrightarrow_g} \qquad \dfrac{}{cartagena, \overline{cartagena} \longrightarrow_g}}{\overline{cartagena}, \{\overline{medellin}, cartagena\}, \{\overline{bogota}, cartagena\}, bogota, barranquilla \longrightarrow_g} \qquad \dfrac{}{cali, \overline{cali} \longrightarrow_g}}{\overline{cartagena}, \{\overline{medellin}, cartagena\}, \{\overline{bogota}, cartagena\}, \{\overline{cali}, bogota\}, \{\overline{cali}, barranquilla\}, cali \longrightarrow_g}$$

4.6. Propiedades de los Sistemas de Pruebas

Hasta ahora hemos asumido que los sistemas en las definiciones 4.1 y 4.2 *funcionan bien*, es decir, solo prueban *tautologías*. A esta propiedad se le conoce como **correctitud**. Además, debemos preguntarnos si toda *verdad* en la lógica proposicional puede ser probada por dichos sistemas. A esta propiedad se le conoce como **completitud**. A continuación probaremos dichas propiedades para el sistema de la Definición 4.1. Seguiremos el método de prueba utilizado en [NvP01].

4.6.1. Correctitud

Primero vamos a redefinir la función de valuación de la Definición 3.2 para facilitar las pruebas que vienen a continuación. Recuerde que el conjunto `Bool` está ordenado de tal manera que $\texttt{F} < \texttt{T}$ y por tanto, $max(\texttt{F}, \texttt{T}) = \texttt{T}$ y $min(\texttt{F}, \texttt{T}) = \texttt{F}$. Además, vamos a asignar 1 al valor de verdad `T` y 0 al valor de verdad `F`.

Definición 4.7 (Semántica de Fórmulas)**.** *Dada una valuación $v : VarP \to \{0, 1\}$, la semántica de una fórmula F se define como la función $[\![\cdot]\!] : (VarP \to \{0, 1\}) \to \textit{Prop} \to \{0, 1\}$*

$$\begin{array}{lcl}
[\![\bot]\!]_v & = & 0 \\
[\![P]\!]_v & = & v(P),\ \textit{si } P \textit{ es una variable proposicional} \\
[\![\neg F]\!]_v & = & 1 - [\![F]\!]_v \\
[\![(F_1 \wedge F_2)]\!]_v & = & min([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \vee F_2)]\!]_v & = & max([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \supset F_2)]\!]_v & = & max(1 - [\![F_1]\!]_v, [\![F_2]\!]_v)
\end{array}$$

Observación 4.2. *Note que $[\![F_1 \supset F_2]\!]_v = 1$ si y solo si $[\![F_1]\!]_v \leq [\![F_2]\!]$. Sea $\Gamma = \{F_1, ..., F_n\}$ un conjunto de fórmulas. Se define $\bigwedge \Gamma$ como la fórmula $F_1 \wedge \cdots \wedge F_n$. Análogamente, $\bigvee \Gamma$ se define como la fórmula $F_1 \vee \cdots \vee F_n$. Además, $\bigwedge(\Gamma, G)$ lo interpretamos como $\bigwedge(\Gamma \cup \{G\})$ y $\bigvee(\Gamma, G)$ como $\bigvee(\Gamma \cup \{G\})$. Si Γ es vacío, $\bigwedge \Gamma = \texttt{F}$ y $\bigvee \Gamma = \texttt{T}$.*

Definición 4.8 (Refutación de secuentes)**.** *Un secuente $\Gamma \longrightarrow \Delta$ es refutable si hay una valuación v tal que $[\![\bigwedge \Gamma]\!]_v > [\![\bigvee \Delta]\!]_v$. El secuente $\Gamma \longrightarrow \Delta$ es válido si no es refutable. En otras palabras, el secuente $\Gamma \longrightarrow \Delta$ es válido si para toda valuación v se cumple que $[\![\bigwedge \Gamma]\!]_v \leq [\![\bigvee \Delta]\!]_v$*

Teorema 4.2 (Correctitud)**.** *Si el secuente $\Gamma \longrightarrow \Delta$ es probable, entonces es válido.* [8]

Demostración. Asumimos que el secuente $\Gamma \longrightarrow \Delta$ es probable. Así que hay una derivación utilizando las reglas en la Definición 4.1 que termina con la regla **axioma** o $\bot_I$. La prueba procede por inducción en la altura de la derivación.

- **Caso Axioma**. Si la prueba termina con

$$\overline{\Gamma, G \longrightarrow \Delta, G}$$

  Note que no importa cual es el valor de $[\![G]\!]_v$, la desigualdad $[\![\bigwedge(\Gamma, G)]\!]_v \leq [\![\bigvee(\Delta, G)]\!]_v$ se cumple.

[8] Dicho de otra manera, todo *teorema* de $\longrightarrow$ es una *tautología*.

- **Caso $\bot_I$**. Asuma ahora que la prueba termina de la siguiente manera:

$$\frac{}{\Gamma, \bot \longrightarrow \Delta} \bot_I$$

Trivialmente $[\![\bigwedge(\Gamma, \bot)]\!]_v = 0$ y por tanto $[\![\bigwedge(\Gamma, \bot)]\!]_v \leq [\![\bigvee \Delta]\!]_v$.

- **Caso $\wedge_I$**. Si la prueba termina con la aplicación de $\wedge_I$:

$$\frac{\overset{\Xi}{\Gamma, G_1, G_2 \longrightarrow \Delta}}{\Gamma, G_1 \wedge G_2 \longrightarrow \Delta} \wedge_I$$

Por inducción sabemos que $[\![\bigwedge(\Gamma \cup \{G_1, G_2\})]\!]_v \leq [\![\bigvee \Delta]\!]_v$. Concluimos este caso observando que $[\![\bigwedge(\Gamma \cup \{G_1, G_2\})]\!]_v = [\![\bigwedge(\Gamma, G_1 \wedge G_2)]\!]_v$.

- **Caso $\vee_D$**. Si la prueba termina con

$$\frac{\overset{\Xi}{\Gamma \longrightarrow \Delta, G_1, G_2}}{\Gamma \longrightarrow \Delta, G_1 \vee G_2} \vee_D$$

Por inducción sabemos que $[\![\bigwedge \Gamma]\!]_v \leq [\![\bigvee(\Delta \cup \{G_1, G_2\})]\!]_v$. Concluimos este caso observando que $[\![\bigvee(\Delta \cup \{G_1, G_2\})]\!]_v = [\![\bigvee(\Delta, G_1 \vee G_2)]\!]_v$.

- **Caso $\wedge_D$**. Asuma ahora que la prueba termina con la aplicación de $\wedge_D$:

$$\frac{\overset{\Xi}{\Gamma \longrightarrow \Delta, G_1} \qquad \overset{\Xi'}{\Gamma \longrightarrow \Delta, G_2}}{\Gamma \longrightarrow \Delta, G_1 \wedge G_2} \wedge_D$$

Por inducción sabemos que $[\![\bigwedge \Gamma]\!]_v \leq [\![\bigvee(\Delta, G_1)]\!]_v$ y $[\![\bigwedge \Gamma]\!]_v \leq [\![\bigvee(\Delta, G_2)]\!]_v$. Note que

$$[\![\bigvee(\Delta, G_1 \wedge G_2)]\!]_v = min([\![\bigvee(\Delta, G_1)]\!]_v, [\![\bigvee(\Delta, G_2)]\!]_v)$$

y por tanto, $[\![\bigwedge \Gamma]\!]_v \leq [\![\bigvee(\Delta, G_1 \wedge G_2)]\!]_v$.

- **Caso $\vee_I$** Asuma que

$$\frac{\overset{\Xi}{\Gamma, G_1 \longrightarrow \Delta} \qquad \overset{\Xi'}{\Gamma, G_2 \longrightarrow \Delta}}{\Gamma, G_1 \vee G_2 \longrightarrow \Delta} \vee_I$$

Este caso se prueba como el anterior notando que por inducción $[\![\bigwedge(\Gamma, G_1)]\!]_v \leq [\![\bigvee \Delta]\!]_v$ y $[\![\bigwedge(\Gamma, G_2)]\!]_v \leq [\![\bigvee \Delta]\!]_v$. Además,

$$[\![\bigwedge(\Gamma, G_1 \vee G_2)]\!]_v = max([\![\bigwedge(\Delta, G_1)]\!]_v, [\![\bigwedge(\Delta, G_2)]\!]_v)$$

- **Caso $\supset_D$**. Si la prueba termina con

$$\frac{\overset{\Xi}{\Gamma, G_1 \longrightarrow \Delta, G_2}}{\Gamma \longrightarrow \Delta, G_1 \supset G_2} \supset_I$$

sabemos por inducción que $[\![\bigwedge(\Gamma, G_1)]\!]_v \leq [\![\bigvee(\Delta, G_2)]\!]_v$. Si $[\![\bigvee \Delta]\!]_v = 1$, trivialmente concluimos que $[\![\bigwedge(\Gamma)]\!]_v \leq [\![\bigvee(\Delta, G_1 \supset G_2)]\!]_v$. Si $[\![\bigvee \Delta]\!]_v = 0$, debe ser cierto que $[\![\bigwedge(\Gamma, G_1)]\!]_v \leq [\![G_2]\!]_v$ y por tanto, $min([\![\bigwedge \Gamma]\!]_v, [\![G_1]\!]_v) \leq [\![G_2]\!]_v$. Consideramos los siguientes casos:

  - Si $[\![G_1]\!]_v = 0$ entonces trivialmente $[\![\bigwedge(\Gamma)]\!]_v \leq max(1 - [\![G_1]\!]_v, [\![G_2]\!])$ y por tanto, concluimos que $[\![\bigwedge(\Gamma)]\!]_v \leq [\![G_1 \supset G_2]\!]_v$.

- Si $[\![G_1]\!]_v = 1$ consideramos de nuevo dos casos. Si $[\![\bigwedge \Gamma]\!]_v = 0$, trivialmente concluimos que $[\![\bigwedge(\Gamma)]\!]_v \leq max(1 - [\![G_1]\!]_v, [\![G_2]\!])$. De lo contrario, como sabemos que $min([\![\bigwedge \Gamma]\!], [\![G_1]\!]_v) \leq [\![G_2]\!]_v$, debe ser cierto que $[\![G_2]\!]_v$=1 y por tanto, $[\![\bigwedge(\Gamma)]\!]_v \leq max(1 - [\![G_1]\!]_v, [\![G_2]\!])$. Concluimos entonces que $[\![\bigwedge(\Gamma)]\!]_v \leq [\![G_1 \supset G_2]\!]_v$.
- **Caso $\supset_I$**. En este caso la prueba termina con la derivación

$$\frac{\overset{\Xi}{\Gamma \longrightarrow \Delta, G_1} \qquad \overset{\Xi'}{\Gamma, G_2 \longrightarrow \Delta}}{\Gamma, G_1 \supset G_2 \longrightarrow \Delta} \supset_I$$

  Por inducción sabemos que $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, [\![G_1]\!]_v)$ y que $min([\![\bigwedge \Gamma]\!]_v, [\![G_2]\!]_v) \leq [\![\bigvee \Delta]\!]_v$.Considere los siguientes casos.

  - Asuma que $[\![G_1]\!]_v = 1$. Por lo tanto,

$$\begin{aligned} &[\![\bigwedge(\Gamma, G_1 \supset G_2)]\!]_v \\ &\qquad = min([\![\bigwedge \Gamma]\!]_v, max(1 - [\![G_1]\!]_v, [\![G_2]\!]_v)) \\ &\qquad = min([\![\bigwedge \Gamma]\!]_v, max(0, [\![G_2]\!])) \\ &\qquad = min([\![\bigwedge \Gamma]\!]_v, [\![G_2]\!]_v) \end{aligned}$$

    Como sabemos que $min([\![\bigwedge \Gamma]\!]_v, [\![G_2]\!]_v) \leq [\![\bigvee \Delta]\!]_v$, concluimos $[\![\bigwedge(\Gamma, G_1 \supset G_2)]\!]_v \leq [\![\bigvee \Delta]\!]_v$.
  - Asuma ahora que $[\![G_1]\!]_v = 0$. Si $[\![\bigvee \Delta]\!]_v = 1$, trivialmente $[\![\bigwedge(\Gamma, G_1 \supset G_2)]\!]_v \leq [\![\bigvee \Delta]\!]_v$. Por otro lado, si $[\![\bigvee \Delta]\!]_v = 0$, dado que $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, [\![G_1]\!]_v)$, necesariamente $[\![\bigwedge \Gamma]\!]_v = 0$ y podemos concluir que $[\![\bigwedge(\Gamma, G_1 \supset G_2)]\!]_v \leq [\![\bigvee \Delta]\!]_v$.
- **Caso $\neg_D$** La derivación termina con

$$\frac{\overset{\Xi}{\Gamma, G \longrightarrow \Delta}}{\Gamma \longrightarrow \Delta, \neg G} \neg_D$$

  Por inducción sabemos que $min([\![\bigwedge \Gamma]\!]_v, [\![G]\!]_v) \leq [\![\bigvee \Delta]\!]_v$. Consideramos los siguientes casos:

  - Si $[\![G]\!]_v = 0$, entonces trivialmente $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, 1 - [\![G]\!]_v)$.
  - Considere ahora que $[\![G]\!]_v = 1$. Si $[\![\bigwedge \Gamma]\!]_v = 0$ la prueba es trivial. De lo contrario, si $[\![\bigwedge \Gamma]\!]_v = 1$, dado que $min([\![\bigwedge \Gamma]\!]_v, [\![G]\!]_v) \leq [\![\bigvee \Delta]\!]_v$, necesariamente $[\![\bigvee \Delta]\!]_v = 1$ y concluimos que $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, 1 - [\![G]\!]_v)$.
- **Caso $\neg_I$** La derivación termina de la siguiente manera:

$$\frac{\overset{\Xi}{\Gamma \longrightarrow \Delta, G}}{\Gamma, \neg G \longrightarrow \Delta} \neg_I$$

  Por inducción sabemos que $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, [\![G]\!]_v)$. Consideramos los siguientes casos:

  - Si $[\![G]\!]_v = 1$, tenemos que $min([\![\bigwedge \Gamma]\!]_v, 1 - [\![G]\!]_v) = 0$ y trivialmente, $[\![\bigwedge(\Gamma, \neg G)]\!]_v \leq [\![\bigvee \Delta]\!]_v$.
  - De lo contrario, si $[\![G]\!]_v = 0$ consideramos dos casos. Si $[\![\bigvee \Delta]\!]_v = 1$ trivialmente tenemos que $[\![\bigwedge(\Gamma, \neg G)]\!]_v \leq [\![\bigvee \Delta]\!]_v$. Si $[\![\bigvee \Delta]\!]_v = 0$, como $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, [\![G]\!]_v)$, necesariamente $[\![\bigwedge \Gamma]\!]_v = 0$ y podemos concluir que $[\![\bigwedge(\Gamma, \neg G)]\!]_v \leq [\![\bigvee \Delta]\!]_v$.

□

4.6.2. Completitud

Antes de probar el teorema de completitud, necesitamos algunos resultados adicionales.

Lema 4.1 (Conectivos en la derivación)**.** *Defina la complejidad del secuente,* $C(\Gamma \longrightarrow \Delta)$*, como el número de conectivos que aparecen en* Γ *y* Δ [9]*. En una derivación de la forma:*

$$\frac{\Gamma' \longrightarrow \Delta'}{\Gamma \longrightarrow \Delta} \qquad o \qquad \frac{\Gamma' \longrightarrow \Delta' \qquad \Gamma'' \longrightarrow \Delta''}{\Gamma \longrightarrow \Delta}$$

tenemos que $C(\Gamma' \longrightarrow \Delta') < C(\Gamma \longrightarrow \Delta)$ *y* $C(\Gamma'' \longrightarrow \Delta'') < C(\Gamma \longrightarrow \Delta)$

Demostración. Note primero que $\Gamma \longrightarrow \Delta$ no puede ser una instancia de **axioma** o de la regla $\perp_I$. El resultado es trivial observando que en cada una de las otras reglas en la Definición 4.1, el número de conectivos del secuente superior es menor al número de conectivos del secuente de la parte inferior de la regla. □

Como sabemos, un secuente en el sistema de la Definición 4.1 es probable solamente si todas sus hojas son instancias de la regla **axioma** o $\perp_I$. Si la descomposición de todos los conectivos del secuente conduce a una hoja donde ni la regla **axioma** ni la regla $\perp_I$ se pueden aplicar, entonces el secuente es refutable.

Lema 4.2 (Probabilidad de un Secuente)**.** *Un secuente* $\Gamma \longrightarrow \Delta$ *es probable con el sistema de la Definición 4.1 si y solamente si todas las hojas del árbol de la derivación terminan con la regla* **axioma** *o* $\perp_I$*. Si el árbol de la derivación termina con alguna de sus hojas donde al lado derecho e izquierdo del secuente solo hay variables proposicionales y no se puede aplicar la regla* **axioma***, entonces el secuente es refutable.*

Demostración. A partir del secuente $\Gamma \longrightarrow \Delta$, aplique las reglas de la Definición 4.1. Por el Lema 4.1, sabemos que en un número finito de pasos, llegaremos a las hojas del árbol de la derivación. Si todas las hojas son instancias de las regla **axioma** o $\perp_I$ entonces el secuente es probable. Ahora asuma que una de las hojas del árbol es de la forma:

$$P_1, \cdots, P_n \longrightarrow Q_1, \cdots, Q_m$$

donde todo P_i y Q_i son variables proposicionales y los conjuntos $\{P_1, \cdots, P_n\}$ y $\{Q_1, \cdots, Q_m\}$ son disyuntos. La valuación v que asigna `T` a cada P_i y `F` a cada Q_i refuta el secuente $\Gamma \longrightarrow \Delta$. Esto se puede probar fácilmente por inducción en la altura de la derivación. □

Teorema 4.3 (Completitud)**.** *Si el secuente* $\Gamma \longrightarrow \Delta$ *es válido entonces es probable.* [10]

Demostración. Asuma que el secuente $\Gamma \longrightarrow \Delta$ es válido. Construya el árbol de la derivación del secuente $\Gamma \longrightarrow \Delta$ y asuma por contradicción que el secuente no es probable. Por Lema 4.2, el árbol termina con hojas que no son instancias de la regla **axioma** o $\perp_I$ y por tanto, existe una valuación que hace que el secuente no sea válido. Por lo tanto, obtenemos una contradicción. □

Corolario 4.1. $\Gamma \longrightarrow \Delta$ *iff* $\Gamma \models \bigvee \Delta$ *iff* $\models (\bigwedge \Gamma) \supset (\bigvee \Delta)$

4.6.3. Consistencia

La consistencia de un sistema de pruebas se puede definir de la siguiente manera.

Definición 4.9 (Consistencia)**.** *Un sistema de pruebas es consistente si no es posible probar* $\perp$*. Es decir, no es posible probar un teorema* F *y su negación* $\neg F$*.*

La anterior definición también se puede leer como "*un sistema de pruebas consistente está libre de contradicciones*".

Usualmente en matemáticas, la prueba de un teorema procede mediante la prueba de algunos lemas que finalmente conducen a la prueba del teorema, es decir, se utilizan *resultados parciales* para completar la prueba. Esto se puede formalizar mediante la regla de corte:

$$\frac{\Gamma \longrightarrow \Delta, G \quad \Gamma, G \longrightarrow \Delta}{\Gamma \longrightarrow \Delta} \; \texttt{cut}$$

[9] Note que si todas las fórmulas en Γ y Δ son variables proposicionales, entonces $C(\Gamma \longrightarrow \Delta) = 0$.

[10] Dicho de otra forma, toda *tautología* es un teorema del sistema $\longrightarrow$.

La regla anterior se puede leer como: primero probamos G a partir de Γ y luego utilizamos G para probar Δ. A G se le conoce como la fórmula de corte.

Observación 4.3 (Adición de la regla `cut`). *Asuma que adicionamos la regla* `cut` *al sistema de la Definición 4.1. Podemos observar lo siguiente:*

- *Si ya tenemos varios teoremas probados, la regla* `cut` *nos permitiría* reutilizar *dichos resultados y por lo tanto, las pruebas en el sistema podrían ser más cortas.*
- *En una prueba de $\Gamma \longrightarrow \Delta$ utilizando* `cut`*, la fórmula G que introduce dicha regla no necesariamente aparece en Γ o Δ. Es decir, G es una fórmula "nueva" que adicionamos en el secuente. Esto haría que la implementación sea más difícil.*
- *Ya no es tan evidente que el sistema ampliado con la regla* `cut` *no pueda probar $\bot$ a partir del conjunto vacío de premisas:*

$$\frac{\longrightarrow G, \bot \quad G \longrightarrow \bot}{\longrightarrow \bot}\ \texttt{cut}$$

quizás con mucha imaginación alguien podría encontrar la fórmula G que completa la derivación.

La última observación es muy importante en la teoría de las pruebas para probar la consistencia de un sistema. Si probamos que la regla de corte es *admisible* [11], quiere decir que cualquier prueba utilizando `cut` se puede escribir sin ella.

Definición 4.10 (Admisibilidad de una regla). *En un sistema de pruebas, una regla R es admisible para el sistema si todo secuente derivado utilizando R puede ser probado también sin utilizar R. Dicho de otro manera, el conjunto de teoremas del sistema no cambia al adicionar R.*

Teorema 4.4 (Eliminación de `cut`). *La regla* `cut` *es admisible para el sistema de la Definición 4.1.*

Corolario 4.2 (Consistencia). *El sistema de la Definición 4.1 es consistente.*

4.7. Ejercicios

Ejercicio 4.1. *Pruebe cada uno de los secuentes que se encuentran a continuación. Utilice los sistemas de la Definición 4.1 y 4.2 según sea el caso* [12].

1. $\Longrightarrow (P \supset Q \supset R) \supset (P \supset Q) \supset (P \supset R)$
2. $\Longrightarrow (P \supset Q) \supset (Q \supset R) \supset (P \supset R)$.
3. $\Longrightarrow (P \supset P \supset Q) \supset (P \supset Q)$.
4. $\longrightarrow (\neg P \supset P) \supset P$.
5. $\Longrightarrow (P \supset Q) \supset \neg Q \supset \neg P$
6. $\Longrightarrow (P \vee Q \supset R) \supset (P \supset R) \wedge (Q \supset R)$.
7. $\Longrightarrow (\neg P \wedge \neg Q) \supset \neg(P \vee (\neg P \wedge Q))$.
8. $\Longrightarrow (\neg(P \wedge P) \supset (\neg P \vee \neg P))$.
9. $\Longrightarrow (P \supset Q) \supset (P \supset \neg Q) \supset P \supset R$.
10. $\Longrightarrow (P \wedge Q) \vee \neg Q \supset Q \supset (P \wedge Q)$.

[11] Este fue uno de los grandes aportes de Gentzen a la teoría de las pruebas.
[12] Estos ejercicios fueron tomados de http://www.cs.ru.nl/H.Geuvers/onderwijs/provingwithCA/

11. $\longrightarrow P \supset (\neg Q \vee (P \wedge Q))$.

12. $\Longrightarrow ((P \supset Q) \wedge (R \supset E)) \supset (\neg Q \vee \neg E) \supset (\neg P \vee \neg R)$.

13. $\longrightarrow \neg(\neg P \wedge \neg Q) \supset (P \vee Q)$.

Ejercicio 4.2. *Prueba los secuentes del ejercicio anterior con la ayuda de Coq.*

Ejercicio 4.3. *Complete la prueba del Lema 4.2.*

Ejercicio 4.4. *Especifique cada uno de los siguientes enunciados como un teorema en Coq y pruébelo. Por ejemplo:*

La máquina dispensadora requiere de una moneda para dar un café. Introduje una moneda en la máquina. Entonces, obtengo un café.

```
Section MaquinaCafe.
  Variables Moneda Cafe: Prop.
  Hypothesis HMaquina: Moneda -> Cafe.
  Hypothesis HDinero: Moneda.

  Theorem QuieroMiCafe: Cafe.
     exact (HMaquina HDinero).
Qed.
End MaquinaCafe.
```

Alternativamente, se puede especificar la misma situación sin el uso de hipótesis (aunque la especificación con hipótesis resulta más clara).

```
Section MaquinaCafe2.
  Variables Moneda Cafe: Prop.

  Theorem QuieroMiCafe2: (Moneda -> Cafe)-> Moneda-> Cafe.
     intros H1 H2.
     exact (H1 H2).
Qed.
End MaquinaCafe2.
```

1. Un programa tiene dos variables x y z. El programa termina si la variables x es positiva o si la variable z es negativa. Siempre alguna de las variables es negativa. Si la variable x es negativa, necesariamente la variables z es negativa. Entonces, el programa termina.

2. Un bar tiene las siguientes reglas para sus miembros:
   a) Todo miembro que no sea escocés tiene medias.
   b) Todo miembro o usa falda o no usa medias.
   c) Las personas casadas no van los domingos.
   d) Una persona va el domingo si y solamente si es escocés.
   e) Toda persona que usa falda es escocés y está casado.
   f) Todo escocés usa falda.

   Se puede concluir que a este bar no viene nadie.

Ayuda: Si no viene nadie es porque el sistema de reglas es inconsistente, es decir, se puede probar falso. Define "ser escocés" como una cierta variable proposicional. Lo mismo para las otras afirmaciones. No requiere usar lógica clásica.

3. Una persona es "escéptica" si asume como cierto algo totalmente diferente a lo que le dicen. De lo contrario es "confiada". Una persona es "mentirosa" si dice todo lo contrario a lo que sabe. De lo contrario es "honesta". Asuma que hay cinco personas A, B, C, D, E en una fila que pueden ganar un premio si dicen una clave secreta. La clave secreta consiste en decir "si" o "no". A puede hablar solamente B, B puede hablar solamente con C y así sucesivamente. En algún momento, una persona externa M se acerca a A y le dice la clave secreta y el mensaje se comienza a propagar de A a B, de B a C y así sucesivamente hasta llegar a E. Sin emabrgo:

   - A es confiada y honesta.
   - B es escéptica y mentirosa.
   - C es escéptica y honesta.
   - D es confiada y mentirosa.
   - E es confiada y honesta.

   Luego individualmente se le pregunta a cada persona la clave secreta. Asumiendo que M le dijo a A la respuesta correcta, ¿quienes ganan el premio ?

Capítulo 5

Sintaxis y Semántica de la Lógica de Predicados

En este capítulo se define la sintaxis de la lógica de predicados (o Lógica de Primero Orden –FOL–). Introduciremos la noción de cuantificadores, términos, estructuras de primer orden y sustituciones.

5.1. La necesidad de variables y cuantificadores

Asuma que quiere formalizar el hecho que la lista L está ordenada. Un primer intento (mezclando el lenguaje natural y los símbolos de la matemática) podría ser:

La lista L está ordenada si para toda posición $i \in 1..length(L)$, $L[i] \leq L[i+1]$.

En la frase anterior, hay varios aspectos que no pueden ser modelados en la lógica proposicional:

1. L: que se refiere a cualquier lista. Note que L podría ser cualquier otra letra si se reemplazan sus tres ocurrencias en la frase. Por lo tanto, L juega el papel de una *variable*.
2. **Para todo**: que nos permite referirnos a *cualquier* número i tal que $0 < i < length(L)$.
3. **L[i], length(L)**: que se refieren a *funciones* que pueden ser aplicadas a objetos tipo lista. En este ejemplo, también el símbolo $+$ en los enteros es una función.
4. L **está ordenada**: que se refiere a una *propiedad* acerca de L. Igualmente, el símbolo $\leq$ es un predicado sobre los elementos del rango de la lista.

La lógica de predicados nos permite expresar dicha frase de la siguiente forma:

$$\forall L \in LIST.(\forall i \in 1..length(L) - 1.(L[i] \leq L[i+1]) \supset ordered(L))$$

5.2. Términos y Fórmulas

El ejemplo de la sección anterior incluía objetos tipo *lista* y los números naturales. Cuando demos significado (semántica) a dicha fórmula, tenemos que expresar de manera clara cuál es el tipo de las variables y las funciones y predicados que se pueden utilizar para construir fórmulas.

Definición 5.1 (Signatura de Primer Orden)**.** *Una signatura de primer orden $\Sigma = (\mathcal{P}, \mathcal{F}, \mathcal{C})$ está compuesta de:*

1. *$\mathcal{P}$: Un conjunto de símbolos de predicados con su respectiva aridad. Por ejemplo, $ordered/1$, $less_equal/2$, $is_Lista/2$, etc.*

2. *$\mathcal{F}$: Un conjunto de símbolos de funciones con su respectiva aridad. Por ejemplo, $length/2$, $plus/2$, $mother_of/2$, etc.*
3. *$\mathcal{C}$: Un conjunto de símbolos de constantes. Por ejemplo, nil, 4, $juan$, etc.*

La signatura Σ define entonces los símbolos *no-lógicos* de la fórmula y se define de acuerdo al *universo de discurso*. Usualmente, el conjunto de contantes $\mathcal{C}$ se omite puesto que las constantes pueden verse como funciones sin argumentos (de aridad 0).

A partir de los símbolos no-lógicos, y los símbolos propios de la lógica, se construye el alfabeto de FOL.

Definición 5.2 (Alfabeto de FOL)**.** *El alfabeto de la lógica de predicados está conformado por:*

- *Los símbolos lógicos $\wedge$, $\vee$, $\neg$, $\supset$, $\bot$.*
- *Los quantificadores: $\forall$ (para todo) y $\exists$ (existe).*
- *El símbolo de igualdad sintáctica $\doteq$ [1].*
- *Un conjunto infinito (contable) de variables $\mathcal{V} = \{x_1, x_2, ...\}$.*
- *Una signatura Σ de símbolos de predicados, funciones y constantes.*
- *Los símbolos auxiliares "(" y ")".*

A partir de las variables, constantes y símbolos de funciones, se construyen los *términos* del lenguaje.

Definición 5.3 (Términos)**.** *Los términos del lenguaje se definen como:*

- *Una variables es un término.*
- *Toda constante $c \in \mathcal{C}$ es un término.*
- *Si $f/n \in \mathcal{F}$ es un símbolo de función, y $t_1, ..., t_n$ son términos, entonces $f(t_1, ..., t_n)$ es un término.*
- *t es un término solamente si está formado mediante la aplicación de las anteriores reglas.*

Ejemplo 5.1 (Términos)**.** *Estos son algunos ejemplos de términos:*

- *nil.*
- *$cons(3, L)$.*
- *$cons(3, cons(4, nil))$.*

Utilizando la notación de BNF, los términos se definen como:

$$t ::= x \mid c \mid f(t, \cdots, t)$$

donde $x \in \mathcal{V}$, $c \in \mathcal{C}$ y $f \in \mathcal{F}$.

Las fórmulas del lenguaje se construyen con la ayuda de los términos, los predicados y los símbolos propios de la lógica.

Definición 5.4 (Sintaxis de FOL)**.** *Dado una signatura Σ, las fórmulas en FOL se construyen de la siguiente forma:*

1. *$\bot$ es una fórmula.*
2. *$t \doteq t'$ es una fórmula.*

[1] Vamos a distinguir la igualdad sintáctica (por ejemplo, $x \not\doteq z$ y $x \doteq x$ de la igualdad en el metalenguaje (por ejemplo, si $x = 3$ y $z = 3$ entonces $x = z$).

3. *Si* $p/n \in \mathcal{P}$ *entonces* $p(t_1, \cdots, t_n)$ *es una fórmula.*
4. *Si* F *es una fórmula entonces* $(\neg F)$ *es una fórmula.*
5. *Si* F, G *son fórmulas entonces* $(F \wedge G), (F \vee G), (F \supset G), (F \equiv H)$ *son fórmulas.*
6. *Si* F *es una fórmula y* $x \in \mathcal{V}$ *entonces* $(\forall x F)$ *y* $(\exists x F)$ *son fórmulas.*
7. F *es una fórmula solamente si está formada mediante la aplicación de las anteriores reglas.*

Alguna veces escribiremos $\forall x.(F)$ *en vez de* $(\forall x F)$ *para facilitar la lectura de las fórmulas. Igualmente para* $\exists x.(F)$*. Además, una secuencia de términos* $t_1, ..., t_n$ *la escribiremos como* $\vec{t}$*. Igualmente para una secuencia de variables* $x_1, ..., x_n$*, escribiremos* $\vec{x}$*.*

Alternativamente, podemos definir el lenguaje de FOL mediante la siguiente sintaxis:

$$F ::= \bot \mid p(\vec{t}) \mid \neg F \mid (F \wedge F) \mid (F \vee F) \mid (F \supset F) \mid (F \equiv F) \mid (\forall x F) \mid (\exists x F)$$

Los cuantificadores *capturan* las variables que están bajo su alcance y se deben distinguir de las otras ocurrencias de las variables en una fórmula. Por ejemplo, en

$$\forall x.(p(x) \supset q(x)) \wedge r(x)$$

La x en $r(x)$ debe ser distinta de la x en $p(x)$. Por otro lado, la x en $p(x)$ y $q(x)$ se refiere a la misma variable que está universalmente cuantificada. Esta noción se puede definir de manera inductiva de la siguiente forma.

Definición 5.5 (Variables Libres y Ligadas)**.** *Primero definimos el conjunto de variables libres de un término de la siguiente forma:*

$$\begin{array}{lcl} FV(c) & = & \emptyset \text{ si } c \in \mathcal{C} \\ FV(x) & = & \{x\} \text{ si } x \in \mathcal{V} \\ FV(f(t_1, \cdots, t_n)) & = & FV(t_1) \cup \cdots \cup FV(t_n) \text{ si } f \in \mathcal{F} \end{array}$$

Dada una fórmula F*, el conjunto de variables libres de* F *se define como:*

$$\begin{array}{lcl} fv(\bot) & = & \emptyset \\ fv(p(\vec{t})) & = & FV(\vec{t}) \\ fv(\neg F) & = & fv(F) \\ fv(F \otimes F') & = & fv(F) \cup fv(F') \text{ donde } \otimes \in \{\wedge, \vee, \supset, \equiv\} \\ fv(\nabla x.F) & = & fv(F) \setminus \{x\} \text{ donde } \nabla \in \{\forall, \exists\} \end{array}$$

Además, $x \in bv(F)$ *si* x *ocurre en* F *y* $x \notin fv(F)$*.* [2] *El conjunto* $bv(F)$ *corresponde a las variables ligadas (o no libres) de* F*.*

Las variables actúan como parámetros que pueden ser sustituidos por una instancia particular. Por ejemplo, considere la fórmula:

$$\neg \forall x.(ave(x) \supset vuela(x))$$

que se puede leer como "*no es cierto que toda ave puede volar*". Dicho de otro modo:

$$\exists x.(ave(x) \wedge \neg vuela(x))$$

"*existen aves que no vuelan*". En particular, el pingüino es un ave que no vuela. Así que una posible *instancia* de la anterior fórmula es:

$$ave(pinguino) \wedge \neg vuela(pinguino)$$

Lo anterior se puede lograr por medio de una *sustitución* de variables por un término específico.

[2] Defina de manera inductiva el conjunto $bv(\cdot)$ y la propiedad *"la variable ocurre en F"*.

Definición 5.6 (Sustitución)**.** *Dada una fórmula F, una variable x y un término t, se define la fórmula $F[t/x]$ como la fórmula que resulta de sustituir todas las ocurrencias libres de x en F por t. Usualmente las sustituciones las denotaremos con la letra θ. Adicionalmente, podemos sustituir múltiples variables al tiempo. Por ejemplo, sea F la fórmula $p(x, y)$. Luego de aplicar la sustitución $[f(y), z/x, y]$ obtenemos $p(f(y), z)$.*

Las sustituciones deben realizarse de tal manera que no se *capturen* variables ligadas de una fórmula. Por ejemplo, considere la fórmula:

$$x \not\doteq 0 \supset \exists y.(y < x)$$

En los números naturales, la anterior fórmula significa que para todo número natural x diferente de cero [3], existe un y tal que $y < x$. Esto debería ser una fórmula válida en los números naturales puesto que siempre podemos escoger y como 0 que es estrictamente menor a todo número natural diferente de cero. Consideremos algunas sustituciones de este fórmula:

- Si aplicamos la sustitución $[0/x]$, obtenemos $0 \not\doteq 0 \supset \exists y.(y < 0)$. Esta fórmula es trivialmente cierta puesto que $0 \doteq 0$ y por tanto, $0 \not\doteq 0$ debe ser falso [4].
- Considere ahora la susticución $[5, 3/x, y]$, es decir, considere la fórmula $5 \not\doteq 0 \supset 3 < 5$ que sabemos que también es verdadera.
- Finalmente, considere la sustitución $[y/x]$ que conduce a la fórmula $y \not\doteq 0 \supset \exists y.(y < y)$. Note que no importa cuál sea el valor de y, la fórmula $y < y$ siempre va a ser falsa.

Para evitar la captura de variables, utilizaremos el renombramiento de las variables ligadas de una fórmula.

Definición 5.7 (Renombramiento de variables)**.** *Utilizaremos la notación $F(x)$ para denotar que la variable x ocurre libre en F. Si z no ocurre en F, un renombramiento de x por z en $\forall x.(F(x))$ es la fórmula $\forall z.(F(z))$ donde todas las ocurrencias libres de x en F se reemplazan por z. Similarmente, para $\exists x.(F(x))$. A este procedimiento se le conoce también como α-conversión.*

Volviendo al ejemplo anterior, la fórmula $\forall x.(x \not\doteq 0 \supset \exists y(y < x))$ la podemos reescribir como:

$$x \not\doteq 0 \supset \exists z.(z < x)$$

En esta nueva fórmula, una sustitución de x por y no captura la variable ligada z.

5.3. Semántica de la Lógica de Predicados

En la lógica proposicional, para dar valor de verdad a una fórmula F, requeríamos dar valor de verdad a las variables proposicionales por medio de una valuación v. En el caso de FOL, debemos *interpretar* los predicados, las funciones, constantes y las variables libres de la fórmula.

Definición 5.8 (Estructuras)**.** *Dada una signatura Σ, una estructura (o modelo) $\mathcal{M}$ es una pareja (D, I) donde D es un conjunto no vacío e I es una función que asigna valores en D a los elementos de Σ de la siguiente manera:*

- *Para cada $f/n \in \mathcal{F}$, $I(f) : D^n \to D$.*
- *Para cada $c \in \mathcal{C}$, $I(c) \in D$.*
- *Para cada $p/n \in \mathcal{P}$, $I(P) \in D^n \to \texttt{Bool}$.*

[3] $t \not\doteq t'$ es una abreviación de $\neg t \doteq t'$.

[4] Más adelante daremos un significado formal a la noción de *verdad* en la lógica de predicados

Note que si el predicado p es de aridad 0, entonces $I(p) \in \texttt{Bool}$, es decir, I se comporta como las valuaciones en la lógica proposicional.

Ejemplo 5.2 (Números naturales)**.** *Asuma una signatura* Σ *con la constante* $\texttt{cero}$*, la función sucesor* $s/1$ *y el predicado* $\texttt{lq}/2$ *(*$\leq$*). Una posible estructura* (M, I) *para interpretar las fórmulas construidas a partir de* Σ *podría ser:*

$$
\begin{array}{lcl}
M & = & Nat \\
I(\texttt{cero}) & = & 0 \\
I(s) & = & \texttt{fun}\ x \Rightarrow (x+1) \\
I(\texttt{lq}) & = & \texttt{fun}\ (x,y) \Rightarrow \texttt{if}\ x \leq y\ \texttt{then T else F}
\end{array}
$$

Utilizando la anterior signatura, considere la fórmula:

$$\exists y.(y \leq x)$$

Para poder decir si es verdadera o no, necesitamos saber cuál es el valor de la variable x [5]. Por lo tanto, definiremos *asignaciones* para darle significado a las variables libres de la fórmula.

Definición 5.9 (Asignaciones)**.** *Dada una signatura* Σ *y un modelo* $\mathcal{M} = (D, I)$*, la asignación* $s : \mathcal{V} \rightarrow M$ *asigna un valor en* M *a cada variable en* $\mathcal{V}$*. Denotaremos por* $s[x \mapsto a]$ *la asignación* s' *tal que* $s(y) = s'(y)$ *para todo* $y \neq x$ *y* $s'(x) = a$*.*

Las fórmulas de la lógica de predicados se interpretan de la siguiente manera.

Definición 5.10 (Semántica de FOL)**.** *Dada una signatura* $\Sigma = (\mathcal{P}, \mathcal{F}, \mathcal{C})$*, un modelo* $\mathcal{M} = (D, I)$ *y una asignación* s*, definimos la función* $\mathcal{T}[\![\cdot]\!]_s^{\mathcal{M}}$ *como:*

$$
\begin{array}{lcl}
\mathcal{T}[\![c]\!]_s^{\mathcal{M}} & = & I(c)\ \textit{if}\ c \in \mathcal{C} \\
\mathcal{T}[\![x]\!]_s^{\mathcal{M}} & = & s(x)\ \textit{if}\ x \in \mathcal{V} \\
\mathcal{T}[\![f(t_1, \cdots, t_n)]\!]_s^{\mathcal{M}} & = & I(f)(\mathcal{T}[\![t_1]\!]_s^{\mathcal{M}}, \cdots, \mathcal{T}[\![t_n]\!]_s^{\mathcal{M}})\ \textit{if}\ f \in \mathcal{F}
\end{array}
$$

Además, definimos la función $[\![\cdot]\!]_s^{\mathcal{M}}$ *como*

$$
\begin{array}{lcl}
[\![\bot]\!]_s^{\mathcal{M}} & = & \texttt{F} \\
[\![t_1 \doteq t_2]\!]_s^{\mathcal{M}} & = & \texttt{T}\ \textit{sii}\ \mathcal{T}[\![t_1]\!]_s^{\mathcal{M}} = \mathcal{T}[\![t_2]\!]_s^{\mathcal{M}} \\
[\![p(t_1, \cdots, t_m)]\!]_s^{\mathcal{M}} & = & I(p)(\mathcal{T}[\![t_1]\!]_s^{\mathcal{M}}, \cdots, \mathcal{T}[\![t_n]\!]_s^{\mathcal{M}}) \\
[\![\neg F]\!]_s^{\mathcal{M}} & = & \texttt{neg}([\![F]\!]_s^{\mathcal{M}}) \\
[\![F \wedge G]\!]_s^{\mathcal{M}} & = & \texttt{con}([\![F]\!]_s^{\mathcal{M}}, [\![G]\!]_s^{\mathcal{M}}) \\
[\![F \vee G]\!]_s^{\mathcal{M}} & = & \texttt{disj}([\![F]\!]_s^{\mathcal{M}}, [\![G]\!]_s^{\mathcal{M}}) \\
[\![F \supset G]\!]_s^{\mathcal{M}} & = & \texttt{imp}([\![F]\!]_s^{\mathcal{M}}, [\![G]\!]_s^{\mathcal{M}}) \\
[\![F \equiv G]\!]_s^{\mathcal{M}} & = & \texttt{equiv}([\![F]\!]_s^{\mathcal{M}}, [\![G]\!]_s^{\mathcal{M}}) \\
[\![(\forall x F)]\!]_s^{\mathcal{M}} & = & \texttt{T}\ \textit{iff}\ [\![F]\!]_{s[x \mapsto a]}^{\mathcal{M}} = \texttt{T}\ \textit{para todo}\ a \in M \\
[\![(\exists x F)]\!]_s^{\mathcal{M}} & = & \texttt{T}\ \textit{iff}\ [\![F]\!]_{s[x \mapsto a]}^{\mathcal{M}} = \texttt{T}\ \textit{para algun}\ a \in M
\end{array}
$$

donde las funciones $\texttt{neg}(\cdot)$*,* $\texttt{con}(\cdot)$*,* $\texttt{disj}(\cdot)$*,* $\texttt{imp}(\cdot)$*,* $\texttt{equiv}(\cdot)$ *se encuentran en la Definición 3.2. Si* $[\![F]\!]_s^{\mathcal{M}} = \texttt{T}$ *escribimos* $\mathcal{M} \models_s F$ *y decimos que* $\mathcal{M}$*, bajo la asignación* s *es un modelo de* F*. De lo contrario, escribimos* $\mathcal{M} \not\models_s F$*.*

Definición 5.11 (Sentencias)**.** *Decimos que la fórmula* F *es una sentencia si* $fv(F) = \emptyset$*.*

Lema 5.1 (Modelos y variables libres)**.** $\mathcal{M} \models_s F$ *sii* $\mathcal{M} \models_{s'} F$ *si* $s(x) = s'(x)$ *para toda* $x \in fv(F)$*. Si* F *es una sentencia y* $[\![F]\!]_s^{\mathcal{M}} = \texttt{T}$*,* s *puede ser cualquier asignación y escribimos* $\mathcal{M} \models F$*. Igualmente para* $\mathcal{M} \not\models F$*.*

[5] De hecho, en este caso, no importa cuál es el valor de x la fórmula sigue siendo cierta en el modelo del Ejemplo 5.2.

Demostración. Trivialmente por inducción en la estructura de F. □

Ejemplo 5.3 (Cadenas Binarias [HR04]). *Asuma una signatura* $\Sigma = (\mathcal{P}, \mathcal{F}, \mathcal{C})$ *donde* $\mathcal{P} = \{\mathtt{lt}\}$, $\mathcal{F} = \{\mathtt{conc}\}$ *y* $\mathcal{C} = \{e\}$. *Vamos a escoger como modelo* $\mathcal{M} = (D, I)$ *donde:*

$$
\begin{array}{lcl}
D & = & \{0,1\}^* \textit{ (i.e., cadenas binarias)} \\
I(e) & = & \epsilon \textit{ (i.e., la cadena vacía)} \\
I(\mathtt{conc}) & = & \mathtt{fun}\ x, y \Rightarrow x \cdot y \\
I(\mathtt{lt}) & = & \mathtt{fun}\ x, y \Rightarrow \mathtt{isPrefix(x,y)} \textit{ (retorna } \mathtt{T} \textit{ si } x \textit{ es un prefijo de } y\textit{)}
\end{array}
$$

En este caso, $\mathcal{M}$ *es un modelo de las siguientes fórmulas independientemente de la asignación* s*:*

1. $\forall x.(\mathtt{lt}(x, \mathtt{conc}(x, e)) \wedge \mathtt{lt}(\mathtt{conc}(x, e), x))$.
2. $\exists y.\forall x.(\mathtt{lt}(y, x))$.
3. $\forall x.\exists y.(\mathtt{lt}(y, x))$.
4. $\forall x, y, z.(\mathtt{lt}(x, y) \supset (\mathtt{lt}(\mathtt{conc}(z, x), \mathtt{conc}(z, y))))$.
5. $\neg\exists x.\forall y.(\mathtt{lt}(x, y) \supset \mathtt{lt}(y, x))$.

Definición 5.12 (Consecuencia Lógica y Satisfacibilidad). *Dado un conjunto de fórmulas* $\Gamma = \{F_1, \cdots, F_n\}$ *y una fórmula* G *decimos que:*

1. $\mathcal{M} \models_s \Gamma$ *si* $\mathcal{M} \models_s F_i$ *para todo* $F_i \in \Gamma$.
2. Γ *es consistente o satisfacible si existe un modelo* $\mathcal{M}$ *y una asignación* s *tal que* $\mathcal{M} \models_s \Gamma$.
3. G *es consecuencia lógica de* Γ, *notación* $\Gamma \models G$, *si para todo modelo* $\mathcal{M}$ *y asignación* s, $\mathcal{M} \models_s \Gamma$ *implica* $\mathcal{M} \models_s F$.
4. *La fórmula* G *es satisfacible si existe un modelo* $\mathcal{M}$ *y una asignación* s *tal que* $\mathcal{M} \models_s G$. *De lo contrario decimos que* G *es insatisfacible.*
5. *La fórmula* G *es válida (una tautología) y para todo modelo* $\mathcal{M}$ *y asignación* s *se cumple que* $\mathcal{M} \models_s G$. *En este caso escribimos* $\models G$.

Ejemplo 5.4 (Tautologías). *Las siguientes fórmulas son válidas:*

- $\exists x.(F \wedge G) \supset (\exists x.(F) \wedge \exists x.(G))$
- $\forall x.(F \wedge G) \equiv \forall x.(F) \wedge \forall x.(G)$

Ejemplo 5.5 (Inducción). *Considere la fórmula:*

$$(A(\mathtt{cero}) \wedge \forall x.(A(x) \supset A(s(x)))) \supset \forall y.(A(y))$$

En el modelo de los números naturales, esta fórmula corresponde al principio de inducción.

Ejemplo 5.6. *Considere la fórmula* F*:*

$$\forall x.(s(x) \not\equiv \mathtt{cero})$$

Esta fórmula es válida si la interpretamos en el modelo de los números naturales. Sin embargo, considere un modelo $\mathcal{M} = (D, I)$ *sonde* D *es un conjunto unitario. Claramente* $\mathcal{M} \not\models_s F$.

Una alternativa para la semántica de los cuantificadores consiste en extender el conjunto de símbolos de constantes $\mathcal{C}$ para incluir todos los elementos del dominio de interpretación como se define a continuación.

Observación 5.1 (Semántica Alternativa)**.** *Asuma un modelo* $\mathcal{M} = (D, I)$ *y una signatura* $\Sigma = (\mathcal{P}, \mathcal{F}, \mathcal{C})$*. Considere el modelo* $\mathcal{M}'$ *con el conjunto de símbolos de constantes* $\mathcal{C}' \supseteq \mathcal{C}$ *tal que para todo* $d_i \in D$ *existe un símbolo* $c_i \in \mathcal{C}'$ *tal que* $\mathcal{T}[\![c_i]\!]_s^{\mathcal{M}} = d_i$*. Entonces:*

$$\begin{array}{lll} \mathcal{M} \models \forall x.F & \textit{iff} & \mathcal{M} \models F[c_i/x] \textit{ para todo } c_i \in \mathcal{C}' \\ \mathcal{M} \models \exists x.F & \textit{iff} & \mathcal{M} \models F[c_i/x] \textit{ para algún } c_i \in \mathcal{C}' \end{array}$$

Observación 5.2. *En general no existe un algoritmo para evaluar* $[\![\cdot]\!]_s^{\mathcal{M}}$*. Por ejemplo considere la fórmula* $\forall x.(F)$*. Verificar si* $[\![\forall x.(F)]\!]_s^{\mathcal{M}} = \mathtt{T}$ *requiere verificar que* $[\![F]\!]_{s[x \mapsto a]}^{\mathcal{M}}$ *para cualquier elemento "a" en el dominio de la interpretación.*

Una posibilidad para generar tautologías en FOL es tomar tautologías de la lógica proposicional y reemplazar las variables proposicionales por fórmulas en FOL. Por ejemplo, considere la tautología $P \supset P$ de la lógica proposicional. La siguientes fórmula es una tautología en FOL:

$$(\forall x.A \supset \exists x.A) \supset (\forall x.A \supset \exists x.A)$$

Sin embargo, no toda tautología de FOL se puede obtener por este método.

Notation 5.1 (Esquemas de Fórmulas)**.** *Asuma que* $F, G, ...$ *son variables que denotan fórmulas en FOL. Una fórmula esquema es una fórmula donde las variables de esquema* $F, G, ..$ *se pueden sustituir por fórmulas específicas. Por ejemplo, el esquema:*

$$\forall x.(F \wedge G) \equiv \forall x.(F) \wedge \forall x.(G)$$

se puede instanciar en la fórmula:

$$\forall x.(\exists x.(p(x, y)) \wedge q(x)) \equiv \forall x.(\exists x.(p(x, y))) \wedge \forall x.(q(x))$$

donde p y q son símbolos de predicados.

5.4. Ejercicios

Ejercicio 5.1 (Sustituciones)**.** *Para cada fórmula F y sustitución* θ*, calcule* $F\theta$*.* p, q, r *denotan símbolos de predicados y* f, g *denotan símbolos de funciones.*

1. $F = p(x) \wedge \forall x.(p(x) \supset q(x))$, $\theta = [y/x]$
2. $F = p(f(x)) \equiv \exists y.(q(x), r(x, f(x)))$, $\theta = [g(f(y))/x]$
3. $F = \forall y.(p(x) \wedge \exists x.(p(y) \supset q(f(x, y))))$, $\theta = [y/x]$

Ejercicio 5.2 (Aritmética de Peano)**.** *Asuma la constante* `cero` *y la función sucesor* $s/1$*. Exprese cada uno de los siguientes axiomas como fórmulas en la lógica de predicados.*

1. *El predicado "=" es una relación de equivalencia, es decir, es una relación simétrica, reflexiva y transitiva.*
2. *Para cualquier número natural* n, $n \neq s(n)$.
3. *Si* $s(m) = s(n)$ *entonces* $m = n$.

Ahora definamos las funciones "+" y "×" que debe satisfacer los siguientes axiomas:

1. 0 *es el elemento neutro para* $+$.
2. $n + s(m)$ *es igual a* $s(n + m)$.

3. 1 *es el elemento neutro para* $\times$.
4. *La multiplicación distribuye sobre la suma.*
5. 0 *es el elemento absorbente para* $\times$.

Defina ahora los siguientes axiomas para la relación de orden $\leq$*:*

1. $\leq$ *es un orden total, es decir, es una relación reflexiva, antisimétrica y transitiva. Además, dado dos números* n *y* m, $n \leq m$ *o* $m \leq n$.
2. *Asuma que* $m \leq n$. *Entonces, se cumple que para cualquier* a, $m+a \leq b+a$ *y* $m \times a \leq b \times a$.
3. `cero` *es el elemento más pequeño con respecto al orden* $\leq$.
4. *Para todo* n *existe un número que es más grande que él mismo.*

Finalmente, defina el predicado $<$. *¿Que axiomas necesitaría ?*

Ejercicio 5.3. *Utilizando los predicados* `padre`(x, y) *(x es el padre de y),* `madre`/2, `esposo`/2 *formalice las siguientes propiedades:*

1. *Todos tenemos un padre y una madre.* [6]
2. *Defina los axiomas necesarios para especificar la relación* `abuelo`.
3. *Defina los axiomas necesarios para especificar la relación* `hermano`.
4. *El padre de una persona no puede ser la madre de esa o de cualquier otra persona.*

Ejercicio 5.4 ([HR04])**.** *Traduzca la siguiente frase como una fórmula en FOL.*

Si existe algún matemático, entonces todos los lógicos son matemáticos. Si existe un informático, entonces todos los matemáticos son informáticos. Así que si existe un informático-matemático, entonces todos los lógicos son informáticos.

Ejercicio 5.5 (The drinker formula)**.** *Suponga que* x *es un cliente de algún bar y asuma que* $B(x)$ *significa que* x *bebe. En el bar se cumple que existe una persona que si empieza a beber, entonces todos beben. Formalice esa frase como una fórmula en FOL.*

Ejercicio 5.6 (Tautologías)**.** *Muestre que las siguientes fórmulas son tautologías. Asuma que* A *y* B *son fórmulas donde* x *ocurre libre,* t *es un término y* $x \notin fv(C)$*:*

1. $\forall x.(A) \supset A[t/x]$
2. $A[t/x] \supset \exists x.(A)$
3. $(\forall x.(F) \vee \forall x.(G)) \supset (\forall x.(F \vee G))$
4. $\forall x.(F) \supset \exists x.(F)$
5. $(\exists x.\forall y.(A)) \supset (\forall y.\exists x.(A))$
6. $C \wedge \forall x.(A) \equiv \forall x.(C \wedge A)$
7. $C \vee \exists x.(A) \equiv \exists x.(C \vee A)$
8. $(\exists x.(A) \supset \forall x.(B)) \supset \forall x.(A \supset B)$
9. $(\forall x.(A) \supset \exists x.(B)) \supset \exists x.(A \supset B)$

Ejercicio 5.7 (Fórmulas no válidas [Gal03])**.** *Muestre que las siguientes fórmulas* **no** *son válidas. Asuma que* $x \in fv(A)$ *y* $x \in fv(B)$*:*

[6] Este axioma puede ser problemático. ¿Como debería ser un modelo para este axioma ? ¿Como podría cambiarse el axioma para que sea más real el modelo del sistema ?

1. $\exists x.(A) \supset \forall x.(A)$
2. $(\forall x.\exists y.A) \supset (\exists x.\forall y.A)$
3. $(\exists x.A \wedge \exists x.B) \supset \exists x.(A \wedge B)$
4. $(\forall x.(A \vee B)) \supset (\forall x.(A) \vee \forall x.(B))$

Ejercicio 5.8. *Asuma que $x \in fv(A)$. Pruebe que:*

$$A \textit{ es válida iff } \forall x.(A) \textit{ es válida}$$

Capítulo 6

Sistemas de Prueba para Lógica de Predicados

Como vimos en el capítulo anterior, la evaluación de $[\![\cdot]\!]_s^{\mathcal{M}}$ no es trivial por las posibles infinitas instancias de la forma $F[t/x]$ cuando se evalúa la fórmula $\forall x.F$. En este capítulo mostraremos el cálculo de secuentes para la lógica de primer orden que extiende el sistema clásico e intuicionista para la lógica proposicional del Capítulo 4. También mostraremos como llevar a cabo pruebas semi-automáticas en Coq cuando se consideran teoremas de la lógica de primer orden. Luego estudiaremos el algoritmo de resolución y algunos ejemplos de programas en Prolog. Finalmente, mostraremos que el cálculo de secuentes es correcto y completo para la lógica de predicados pero que, en general, el problema de validez de una fórmula no es decidible.

6.1. Sistema Clásico

El cálculo de secuentes para la lógica de primer orden clásica extiende el sistema de la Definición 4.1 con las reglas de inferencia para introducir los cuantificadores a ambos lados del secuente.

Definición 6.1 (Cálculo de Secuentes)**.** *A continuación utilizaremos Δ, Γ para denotar conjuntos de fórmulas de primer orden. Las reglas de inferencia del cálculo de secuentes para los cuantificadores son las siguientes: [NvP01]:*

$$\frac{\Gamma, \forall x.F, F[t/x] \longrightarrow \Delta}{\Gamma, \forall x.F \longrightarrow \Delta}\ \forall_I \qquad \frac{\Gamma \longrightarrow \Delta, F[x'/x]}{\Gamma \longrightarrow \Delta, \forall x.F}\ \forall_D$$

$$\frac{\Gamma, F[x'/x] \longrightarrow \Delta}{\Gamma, \exists x.F \longrightarrow \Delta}\ \exists_I \qquad \frac{\Gamma \longrightarrow \Delta, \exists x.F, F[t/x]}{\Gamma \longrightarrow \Delta, \exists x.F}\ \exists_D$$

donde x' es una variables que no ocurren libre en las fórmulas del secuente inferior. A esta variable se le conoce como eigenvariable*. Además, t es un término tal que la sustitución $[t/x]$ no captura variables (ver Definición 5.7).*

Veamos algunos ejemplos.

Ejemplo 6.1. *La siguiente es la derivación para probar que $\forall x.p(x) \supset \exists x.p(x)$ donde p es un*

predicado unario.

$$\dfrac{\dfrac{\dfrac{\overline{\forall x.p(x), p(t) \longrightarrow p(t), \exists x.p(x)}}{\forall x.p(x) \longrightarrow p(t), \exists x.p(x)}\ \forall_I}{\forall x.p(x) \longrightarrow \exists x.p(x)}\ \exists_D}{\longrightarrow \forall x.p(x) \supset \exists x.p(x)}\ \supset_D$$

Ejemplo 6.2. *Asuma que A y B son fórmulas donde x ocurre libre.* [1]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{A(x') \longrightarrow B(x'), A(x'), \exists x.(A)}}{A(x') \longrightarrow B(x'), \exists x.(A)}\ \exists_D \qquad \dfrac{\overline{\forall x.(B), B(x'), A(x') \longrightarrow B(x')}}{\forall x.(B), A(x') \longrightarrow B(x')}\ \forall_I}{\exists x.(A) \supset \forall x.(B), A(x') \longrightarrow B(x')}\ \supset_I}{\exists x.(A) \supset \forall x.(B) \longrightarrow A(x') \supset B(x')}\ \supset_D}{\exists x.(A) \supset \forall x.(B) \longrightarrow \forall x.(A \supset B)}\ \forall_D}{\longrightarrow (\exists x.(A) \supset \forall x.(B)) \supset \forall .x(A \supset B)}\ \supset_D$$

Ejemplo 6.3. *Sea F la fórmula $\exists x.(p(x) \supset \forall z.(p(z)))$. Podemos concluir que F es válida por medio de la siguiente derivación:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{p(t), p(z') \longrightarrow F, p(z'), \forall z.(p(z))}}{p(t) \longrightarrow F, p(z'), p(z') \supset \forall z.(p(z))}\ \supset_D}{p(t) \longrightarrow F, p(z')}\ \exists_D}{p(t) \longrightarrow F, \forall z.(p(z))}\ \forall_D}{\longrightarrow F, p(t) \supset \forall z.(p(z))}\ \supset_D}{\longrightarrow \exists x.(p(x) \supset \forall z.(p(z)))}\ \exists_D$$

6.2. Sistema Intuicionista

El cálculo de secuentes para la lógica de primer orden intuicionista extiende el sistema de la Definición 4.2 con las reglas de inferencia para introducir los cuantificadores a ambos lados del secuente.

Definición 6.2 (Cálculo de Secuentes Lógica Intuicionista)**.** *A continuación utilizaremos Δ, Γ para denotar conjuntos de fórmulas de primer orden y G para denotar una fórmula de primer orden. Las reglas para los cuantificadores son las siguientes: [NvP01]:*

$$\dfrac{\Gamma, \forall x.F, F[t/x] \Longrightarrow G}{\Gamma, \forall x.F \Longrightarrow G}\ \forall_I \qquad \dfrac{\Gamma \Longrightarrow F[x'/x]}{\Gamma \Longrightarrow \forall x.F}\ \forall_D$$

$$\dfrac{\Gamma, F[x'/x] \Longrightarrow G}{\Gamma, \exists x.F \Longrightarrow G}\ \exists_I \qquad \dfrac{\Gamma \Longrightarrow F[t/x]}{\Gamma \Longrightarrow \exists x.F}\ \exists_D$$

donde x' es una variables que no ocurren libre en las fórmulas del secuente inferior. Además, t es un término tal que la sustitución $[t/x]$ no captura variables (ver Definición 5.7).

Como en el caso de la lógica proposicional, algunos secuentes pueden ser probados con el sistema de la Definición 6.1 y no son probables con el sistema intuicionista.

[1] El resultado de la sustitución $A[x'/x]$ lo representamos como $A(x')$

Ejemplo 6.4. *Los siguientes secuentes son probables clásicamente pero no lo son con el sistema intuicionista.*

1. $\longrightarrow \neg(\forall x.(\neg A)) \supset \exists x.(A)$
2. $\longrightarrow \neg\neg\forall x(A \vee \neg A)$

Definición 6.3 (Formal Normal Prenexa)**.** *Una fórmula está en forma normal prenexa si toma la forma* $\nabla x_1.\nabla x_2....\nabla x_n F$ *donde* F *está libre de cuantificadores y* $\nabla \in \{\forall, \exists\}$.

Dada una fórmula F, una simple transformación nos permite encontrar su forma prenexa.

Definición 6.4. *Asuma que* $x \notin fv(G)$ *y considere las siguientes transformaciones:*

- $\forall x.(F) \wedge G \Rrightarrow \forall x.(F \wedge G)$
- $\forall x.(F) \vee G \Rrightarrow \forall x.(F \vee G)$
- $\exists x.(F) \wedge G \Rrightarrow \exists x.(F \wedge G)$
- $\exists x.(F) \vee G \Rrightarrow \exists x.(F \vee G)$
- $\neg\forall x.(F) \Rrightarrow \exists x.(\neg F)$
- $\neg\exists x.(F) \Rrightarrow \forall x.(\neg F)$
- $\forall x.(F) \supset G \Rrightarrow \exists x.(F \supset G)$
- $\exists x.(F) \supset G \Rrightarrow \forall x.(F \supset G)$
- $G \supset \exists x.(F) \Rrightarrow \exists x.(G \supset F)$
- $G \supset \forall x.(F) \Rrightarrow \forall x.(G \supset F)$

Teorema 6.1. *La* F *es equivalente en lógica clásica a su versión prenexa* F'*. Es decir, el secuente* $\longrightarrow F \equiv F'$ *es probable.*

Demostración. Basta con probar con el sistema $\longrightarrow$ la equivalencia de las transformaciones en la Definición 6.4. Por ejemplo:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{F(x') \longrightarrow F(x'), \exists x.(\neg F)}}{\longrightarrow F(x'), \neg F(x'), \exists x.(\neg F)}\,\neg_D}{\longrightarrow F(x'), \exists x.(\neg F)}\,\exists_D}{\longrightarrow \forall x.(F), \exists x.(\neg F)}\,\forall_D}{\neg\forall x.(F) \longrightarrow \exists x.(\neg F)}\,\neg_I}{\longrightarrow \neg\forall x.(F) \supset \exists x.(\neg F)}\,\supset_D \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\forall x.(F), F(x') \longrightarrow F(x')}}{\forall x.(F) \longrightarrow F(x')}\,\forall_I}{\neg F(x'), \forall x.(F) \longrightarrow}\,\neg_I}{\exists x.(\neg F), \forall x.(F) \longrightarrow}\,\exists_I}{\exists x.(\neg F) \longrightarrow \neg\forall x.(F)}\,\neg_D}{\longrightarrow \exists x.(\neg F) \supset \neg\forall x.(F)}\,\supset_D}{\longrightarrow \neg\forall x.(F) \equiv \exists x.(\neg F)}\,\wedge_D$$

□

Teorema 6.2. *Asuma que* $x \notin fv(F)$*. Los siguientes secuentes no son probables en lógica intuicionista:*

- $\Longrightarrow \forall x(F \vee G) \supset F \vee \forall x.(G)$
- $\Longrightarrow (F \supset \exists x.(G)) \supset \exists(F \supset G)$
- $\Longrightarrow (\forall x.(G) \supset F) \supset \exists x.(G \supset F)$

Corolario 6.1. *No toda fórmula* F *es equivalte a su forma normal prenexa en lógica intuicionista.*

6.3. FOL en Coq

Para poder probar teoremas de la Lógica de Predicados en Coq, primero debemos definir la signatura con las funciones, predicados y símbolos de constantes que serán utilizados.

```
(* Ejemplo de FOL *)
Section Amigos.

  Variable Persona: Set.
  (* Amigo es un predicado Binario *)
  Variables Amigo: Persona -> Persona -> Prop.
```

Se pueden definir propiedades sobre los predicados, por ejemplo:

```
  (* La relacion amigo no es reflexiva *)
  Hypothesis Amigo_no_reflex: forall x:Persona, ~ Amigo x x.
  (* La relacion amigo es simetrica *)
  Hypothesis Amigo_symetric: forall x y :Persona, Amigo x y -> Amigo y x.
  (* La relacion amigo es transitiva *)
  Hypothesis Amigo_trans: forall x y z:Persona, Amigo x y ->
                                        Amigo y z -> Amigo x z.
```

Los teoremas se establecen y se prueban de manera similar como en la lógica proposicional.

```
 (* Si x es amigo de y, entonces existe un amigo para y *)
  Theorem T1:  forall x y:Persona, Amigo x y -> exists z :Persona, Amigo y z.
     intros x y. (* regla para forall a la derecha *)
     intro H.
     (* como amigo x y, por simetria amigo y x. Entonces, el testigo del exists
         puede ser x *)
     exists x.
     apply Amigo_symetric.
     assumption.
  Qed.
```

En general, las tácticas para introducir y eliminar los cuantificadores en una prueba son:

- $\forall_D$: Si x no aparece en las hipótesis:

$$\frac{\Gamma, x : D \Longrightarrow F}{\Gamma \Longrightarrow \forall x : D.F}\ \texttt{intro x}$$

  Si x aparece en Γ, simplemente se escoge otra varaible para eliminar el cuantificador, por ejemplo, `into` x'.

- $\exists_D$: En este caso, el término t debe aparecer en las hipótesis Γ:

$$\frac{\Gamma \Longrightarrow F[t/x]}{\Gamma \Longrightarrow \exists x : D.F}\ \texttt{exists}\ t$$

- $\forall_I$: Si se quiere instanciar la fórmula $\forall x.F$ para un término particular, primero se debe introducir la fórmula $F[t/x]$ por medio de `assert`. Este subgoal se prueba fácilmente por medio de `apply`:

$$\dfrac{\dfrac{}{\Gamma, H \Longrightarrow F\theta}\ \texttt{apply}\ H \qquad \dfrac{}{\Gamma, H, H' : F\theta \longrightarrow G}}{\Gamma, H : \forall x : D.F \Longrightarrow G}\ \texttt{assert}\ (H' : F\theta)$$

  En algunas ocasiones, `apply` no puede deducir la sustitución necesaria para probar $F\theta$. En estos casos se puede indicar de manera explícita la sustitución de los parámetros:

```
H: forall x y:D, p x y.
------------
p t1 t2
```

  El goal $p\ t1\ t1$ se puede probar mediante

```
apply H with (x:= t1) (x:= t2).
```

- $\exists_I$: Si la variable z no aparece en las hipótesis:

$$\dfrac{\Gamma, z : D, H' : F[z/x]}{\Gamma, H : \exists x : D.F \longrightarrow G}\ \texttt{destruct}\ H\ \texttt{as[}\ z\ H'\ \texttt{]}$$

6.4. Especifación y Verificación

En esta sección mostraremos algunos ejemplos de especificación de sistemas. Además, mostraremos cómo se pueden verificar propiedades (simples) de dichos sistemas.

Ejemplo 6.5 (Cursos de una Universidad)**.** *El sistema de registro de una universidad tiene los siguientes requerimientos:*

Cada curso está dictado por un único profesor. Todo profesor debe dictar al menos un curso. Los profesores se clasifican en hora cátedra y tiempo completo.

Un modelo de este sistema se encuentra en la Figura 6.1.

Ejemplo 6.6 (Definición de Relaciones)**.**

Toda persona tiene un padre. El abuelo de una persona se define como el padre del padre de una persona. Pruebe que toda persona tiene un abuelo.

En la figura 6.2 se encuentra una posible solución a este enunciado.

Ejemplo 6.7 (Conferencia)**.** *Los requerimientos para un sistema de conferencias son:*

Los autores envían artículos. Un artículo puede tener varios autores. Un autor puede aparecer en varios artículos. Las personas se inscriben en la conferencia. Los artículos se programan para presentación en la conferencia. Solamente se programa un artículo si al menos uno de sus autores se ha inscrito en la conferencia. Alguno de los autores presenta cada artículo. Las presentaciones se dividen en sesiones. Cada sesión tiene un moderador, que debe estar inscrito en el congreso. Si una persona presenta un artículo de una sesión, no puede ser moderador de la sesión. La misma persona no puede ser moderador de sesiones diferentes.

En la Figura 6.3 se encuentra la especificación de dicho sistema en Coq.

Ejemplo 6.8 (Oficina Burocrática)**.** *Una oficina de trámites funciona de la siguiente manera:*

> Se observa que las personas entran y salen. Algunas personas tienen el $sello_1$, otras el $sello_2$ y otras el $sello_3$. Las reglas de la oficina dictan que para obtener el $Sello_{i+1}$ se debe obtener primero el $Sello_i$. Además, para obtener un sello necesariamente se debe estar dentro de la oficina.

El modelo para este problema se encuentra en la Figura 6.4. Para entender mejor las definiciones de Coq utilizadas en este ejemplo, hay que referirse a la librería Ensemble *(http://coq.inria.fr/stdlib/Coq.Sets.Ensembles.html).Note que un conjunto de elementos de cierto tipo U se define en Coq como:*

```
Definition Ensemble := U -> Prop.
```

Ensemble es entonces una función que toma como parámetro un elemento tipo U y retorna verdadero (si el elemento pertenece) o $falso$ si el elemento no pertenece al conjunto.
Dado un conjunto A, la definición de $x \in A$ es entonces:

```
Definition In (A:Ensemble) (x:U) : Prop := A x.
```

y la definición de $A \subseteq B$ es:

```
Definition Included (B C:Ensemble) : Prop := forall x:U, In B x -> In C x.
```

Las definiciones de Ensemble *están contenidas en una sección que define el tipo del conjunto:*

```
Variable U : Type.
```

Así que todas las definiciones son paramétricas con respecto al tipo U:

```
Check Included.
---------
Included
     : forall U : Type, Ensemble U -> Ensemble U -> Prop
---------
Check In.
---------
In
     : forall U : Type, Ensemble U -> U -> Prop
---------
```

Así que si definimos:

```
Variable A B :Ensemble PERSONAS.
```

y queremos adicionar como hipótesis que $A \subseteq B$, debemos instanciar *el parámetro U en la definición de* Included*:*

```
  (Included PERSONAS) A B
```

Para evitar esto, podemos definir lo siguiente:

```
Definition INC_PER := Included PERSONAS.
```

y utilizar simplemente:

```
  INC_PER A B
```

Por otro lado, es posible, a partir de los invariantes definidos, expresar y probar la siguiente propiedad:

Si una persona tiene uno de los sellos es porque está dentro de la oficina.

(ver Lema "sello_implica_in" al final de la Figura 6.4.

Ejemplo 6.9 (Centro de Votación)**.** *A continuación se encuentran los requerimientos para un sistema de votación:*

- Un centro de votación es un lugar en el que la gente entra a depositar su voto.
- En el centro de votación hay mesas.
- Las personas no puede votar en cualquier mesa. Cada mesa tiene una lista de votantes asignados a esa mesa.
- Solo las personas que están inscritas para votar se les asigna una mesa.
- Se debe controlar las personas que están dentro del centro de votación
- Una persona no puede entrar al centro si no está registrada en alguna de las mesas del mismo.

La especificación del sistema se encuentra en la Figura 6.5.

Ejemplo 6.10 (Accesos y Permisos)**.** *Se desea modelar un sistema que permita controlar el acceso a ciertos espacios. Los requerimientos son los siguientes:*

- Un centro de investigación está conformado por una serie de laboratorios.
- Las personas que trabajan para el centro tienen permiso para entrar a algunos de las laboratorios y a otros no.
- Por cada laboratorio hay al menos una persona que tiene acceso para entrar en él.
- Todas las personas que pertenecen al centro tienen acceso para entrar en al menos uno de los laboratorios.
- Se observan las personas que están dentro del centro y las personas que se encuentran en cada laboratorio.
- Una persona no puede estar en el centro si no pertenece al centro.
- Así mismo, una persona no puede estar en un laboratorio si no tiene permiso para ello.
- En el sistema también se observan grupos de personas, por ejemplo, el grupo de los ingenieros, el grupo de los químicos, etc.
- Las personas pueden pertenecer a varios grupos.
- Por cada grupo se conoce los laboratorios a los cuales sus miembros tienen acceso.
- Cada grupo tiene acceso a por lo menos uno de los laboratorios.
- Se debe controlar que si una persona pertenece a un grupo, entonces debe tener acceso a todas los laboratorios que el grupo tiene acceso.

El modelo de este sistema se encuentra en al Figura 6.6

Ejemplo 6.11 (Red Social)**.** *A continuación se presentan los requerimientos para una red social:*

Las personas deben registrarse para hacer uso de la red social. Los usuarios registrados establecen relaciones de amistad con otros usuarios. La relación de amistad es simétrica y no necesariamente transitiva. Los usuarios publican nuevos contenidos en la red. Los contenidos se clasifican en tres tipos:

- Privado: Solamente quien creó el contenido puede verlo.
- Compartido : Solamente los amigos de quien creo el contenido tienen acceso a este.
- Público: Todos los amigos de los amigos (clausura transitiva) tienen acceso al contenido.

El modelo del sistema se encuentra en la Figura 6.7.

Se desean verificar las siguientes propiedades:

1. *Si todo usuario tiene al menos un contenido entonces todo usuario tiene acceso a al menos un contenido.*
2. *Asuma ahora que p1 publica un contenido "público" y que p1 es amigo de p2. A su vez, p2 es amigo de p3. Se debe poder mostrar que p1,p2 y p3 pueden ver el contenido.*
3. *Privacidad: Si p1 publica algo privado, entonces nadie diferente a él mismo lo puede ver.*

La especificación de estas propiedades y su prueba se encuentran en la Figura 6.8. El código en esta figura debe copiarse dentro de la Sección RedSocial *creada en el código de la Figura 6.7.*

```
Section Cursos_Profesores.
(********* tipos *******)
(* Tipo de los cursos *)
Variable Curso:Set.
(* Profesores *)
Variable Profesor:Set.
(* Clasificacion de los profesores *)
Variable Tipo:Set.
(* Defincion de tiempo completo y H. catedra *)
Variables tc hc : Tipo.
(* Definicion de la funcion que asigna una clasificacion
   a los profesores *)
Variable Tipo_Profesor: Profesor -> Tipo.

(****** predicados ******)
(* Dicta es un predicado binario donde
    Dicta P C significa que el profesor P
    dicta el curso C. *)
Variable Dicta:Profesor -> Curso -> Prop.

(* Todo curso debe estar dictado por un profesor *)
Definition H1: Prop :=  forall c:Curso, exists p:Profesor, Dicta p c.
(*  El profesor que dicta el curso es unico *)
Definition H2:Prop :=  forall c:Curso, exists p1 p2:Profesor,
   Dicta p1 c/\ Dicta p2 c -> p1=p2.
(* Todo profesor debe dictar al menos un curso *)
Definition H3:Prop :=  forall p:Profesor, exists c:Curso, Dicta p c.

(* Todo profesor debe tener un tipo *)
Hypothesis H4: forall x, Tipo_Profesor x = tc \/ Tipo_Profesor x = hc.

End Cursos_Profesores.
```

Figura 6.1: Especificación del sistema de cursos (ver Ejemplo 6.5).

```
Section Padre_Abuelo.
(*****  Tipos  *****)
Variable Persona:Set.

(* Definicion de  Papa
    Papa x y significa que x es el papa de y
*)
Variable Papa: Persona -> Persona -> Prop.

(* Toda persona tiene un papa   *)
Hypothesis H1: forall x:Persona, exists y,  Papa y x .
(* El papa de una persona no puede ser la misma persona *)
Hypothesis H1': forall x:Persona, ~Papa x x.

(* La relacion Papa no puede ser simetrica *)
Hypothesis H1'' : forall x y:Persona, Papa x y -> ~Papa y x.

(* Definicion de Abuelo
    Abuelo x y significa que x es el abuelo de y
*)
Definition Abuelo  (x y: Persona): Prop :=  exists z, Papa z y /\ Papa x z.

Theorem Todos_tienen_abuelo: forall x:Persona, exists y, Abuelo y x.
  intro x.
  assert (exists papa_x:Persona, Papa papa_x x).
  apply H1.
  destruct H as [papa_x H'].
  assert (exists papa_papa_x:Persona, Papa papa_papa_x papa_x).
  apply H1.
  destruct H as [papa_papa_x H''].
  exists papa_papa_x.
  unfold Abuelo. (* Aplicar la definicion de abuelo *)
  exists papa_x.
  split;assumption.
Qed.

End Padre_Abuelo.
```

Figura 6.2: Solución al problema de padres y abuelos (ver Ejemplo 6.6.

```
Section Conferencia.
  (* Definicion de los tipos  *)
  Variables (PERSONAS: Type) (* Autores y participantes de la conferencia *)
           (ARTICULOS: Type) (* Papers *)
           (SESIONES: Type) (* Sesiones de la conferencia *)
           (TIME: Type). (* Time-slots de la conferencia *)

(* Definicion de las variables del modelo e Invariantes *)
 Variable autores_articulos: PERSONAS -> ARTICULOS -> Prop. (*Autores de los articulos *)
 Variable inscritos: PERSONAS -> Prop. (* Personas inscritas a la conferencia *)
 Variable sesion_articulo: ARTICULOS -> SESIONES. (* La sesion asignada a cada articulo *)
 Variable presenta_articulo : ARTICULOS -> PERSONAS. (* Autor que presenta el articulo *)
 Variable moderador: SESIONES -> PERSONAS. (* El moderador de la sesion *)

(* Un articulo se programa solamente si uno de los autores se inscribio en la conferencia *)
 Definition INV_1:Prop :=  forall (a:ARTICULOS) (s: SESIONES) , s = sesion_articulo a  ->  exists p:PERSONAS, inscritos p /\ autores_articulos p a.

(* El articulo lo debe presentar uno de los autores que esta inscrito *)
 Definition INV_2  :Prop := forall (a:ARTICULOS) (p:PERSONAS),   p = presenta_articulo a ->inscritos p /\ autores_articulos p a.

(* Note que INV_1 e INV_2 podrian expresarse como:
 Definition INV_1_2  :Prop :=  forall (a:ARTICULOS)  (s: SESIONES) ,
   s = sesion_articulo a  ->   exists p:PERSONAS, p = presenta_articulo a /\ inscritos p  /\ autores_articulos p a. *)

(* El moderador de cada sesion debe estar inscrito al congreso *)
  Definition INV_3 :Prop := forall (s:SESIONES) (p:PERSONAS),   p = moderador s -> inscritos p.

(* El moderador no puede ser uno de los que presentan articulos en la sesion *)
  Definition INV_4  :Prop := forall (a:ARTICULOS) (p:PERSONAS) (s:SESIONES),
     p = presenta_articulo a -> p <> moderador (sesion_articulo a).
(* Una persona no puede ser moderador de dos sesiones diferentes *)
  Definition INV_5  :Prop :=forall (s s':SESIONES),  s<>s' -> moderador s <> moderador s'.

(* Los articulos se presentan en un horario *)
  Variable horario_sesion : SESIONES -> TIME.
(* Si una persona va a presentar dos articulos, esos articulos estan en la misma sesion
  o pertenecen a sesiones con horarios diferentes --no coincidencia de horario-- *)
  Definition INV_6  : Prop := forall a a':ARTICULOS,  a <> a' -> presenta_articulo a = presenta_articulo a' ->
    sesion_articulo a <> sesion_articulo a' -> horario_sesion (sesion_articulo a) <> horario_sesion (sesion_articulo a').

  (* El moderador no puede presentar un articulo en la misma sesion *)
  (* Si asumimos INV_4, el moderador no puede presentar articulos en la sesion que esta moderando *)
  Definition INV_7  : Prop := forall (p:PERSONAS) (s:SESIONES) (a:ARTICULOS),  p = moderador s ->  p = presenta_articulo a ->
    horario_sesion s <>  horario_sesion (sesion_articulo a).

  (* El invariante del sistema es la conjuncion de todas las definiciones anteriores.  *)
  Definition INVARIANTE (P:Prop) :Prop :=  INV_1 -> INV_2  -> INV_3  -> INV_4 -> INV_5 -> INV_6 -> INV_7 -> P.

(* Asumamos que hay un cientifico muy brillante que es coautor en todos los papers de la conferencia *)
 Variable genio: PERSONAS.
  Hypothesis H1: forall (a: ARTICULOS), autores_articulos genio a.

 (* Asumamos que existen dos sesiones --diferentes-- y paralelas *)
  Variable sesion1 sesion2 : SESIONES.
  Hypothesis H2: horario_sesion sesion1 = horario_sesion sesion2.
  Hypothesis H2': sesion1 <> sesion2.

 (* Asumamos tambien que el cientifico presenta un articulo en la sesion 1. *)
 Variable articulo:ARTICULOS.
  Hypothesis H3: genio = presenta_articulo articulo.
  Hypothesis H3': sesion1 = sesion_articulo articulo.

 (* Podemos probar que el cientifico efectivamente esta inscrito a la conferencia *)
 Lemma En_conferencia: INVARIANTE (inscritos genio).
   unfold INVARIANTE.
   intros I1 I2 I3 I4 I5 I6 I7. (* aqui tenemos todos los invaraintes como hipotesis *)
   apply I2 with (a:= articulo) (p:=genio). (* La sustitucion de "a" y "p" en el forall se hace explicita *)
   exact H3.
 Qed.

 (* Podemos probar tambien que el cientifico no puede presentar ninguno de los articulos programados en la sesion 2*)
 Lemma No_Sesion2: INVARIANTE (forall art:ARTICULOS, art <> articulo -> sesion2 = sesion_articulo art -> genio <> presenta_articulo art).
   unfold INVARIANTE.
   (* Introduccion de todas las hipotesis *)
   intros I1 I2 I3 I4 I5 I6 I7. intro art H4 H5 H6.
  (* Tenemos como hipotesis que el cientifico presenta el articulo "art" y el articulo "articulo". Podemos utilizar el INV6 para probar esto *)
   absurd  (horario_sesion sesion1 = horario_sesion sesion2).
   unfold INV_6 in I6.
 (* El INV6 se puede aplicar sobre una expresion de la forma    horario_sesion (sesion_articulo a) <> horario_sesion (sesion_articulo a')
      Sin embargo, lo que tenemos como goal es horario_sesion sesion1 <> horario_sesion sesion2. Entonces primero debemos sustituir en el goal
     la expresion sesion 1 para poder utilizar el INV6 *)
  replace sesion1 with (sesion_articulo articulo).
  replace sesion2 with (sesion_articulo art).
   apply I6  with (a:=articulo) (a' := art). (* Aplicacion del invariante 6 *)
   auto. (* prueba articulo <> art utilizando  H4 *)
   (* La prueba de presenta_articulo articulo = presenta_articulo art es simple puesto que quien presenta "articulo" y "art" es el cientifico *)
  replace (presenta_articulo articulo) with genio.
  replace (presenta_articulo art) with genio.
  trivial.
  (* la prueba de sesion_articulo articulo <> sesion_articulo art tambien es simple puesto que una sesion corresponde a la sesion 1 y la otra a la
   sesion2 -- que son diferentes por hipotesis -- *)
  replace (sesion_articulo articulo) with sesion1.
  replace (sesion_articulo art) with sesion2.
  exact H2'.
 (* Ahora se prueba el otro lado del "absurd" que se tiene como hipotesis *)
  assumption.
Qed.
End Conferencia.
```

Figura 6.3: Especificación del Sistema de Conferencias (ver Ejemplo 6.7).

```
Require Import Coq.Sets.Ensembles.
(* Un conjunto --ensable-- no es mas que un predicado de la forma:
  Definition Ensemble := U -> Prop.*)
Variable PERSONAS : Type.
Section Oficina.
  (* Definicion de las personas que estan dentro de la oficina *)
  Variable per_in: Ensemble PERSONAS. (*Print per_in. *)
  (* Personas que tienen el sello_i *)
  Variable sello1 : Ensemble PERSONAS.
  Variable sello2 : Ensemble PERSONAS.
  Variable sello3 : Ensemble PERSONAS.

  (* Para simplificar la notacion, podemos introducir la siguiente definicion *)
  Definition SET_PER := Ensemble PERSONAS.

  (* Si una persona tiene un sello, esta dentro de la oficina *)
  (* Ademas, si tiene el selloi, tiene el sello anterior. *)

  (* La formula subconjunto(sello3,sello2) se puede definir como:
 forall x:PERSONAS, sello3 x -> sello2 x. *)

  (* Esta definicion se encuentra en la libreria Ensamble:
  Definition Included (B C:Ensemble) : Prop := forall x:U, In B x -> In C x.*)

  Definition INV_1:Prop := Included PERSONAS sello3 sello2.
  Definition INV_2:Prop := Included PERSONAS sello2 sello1.
  Definition INV_3:Prop := Included PERSONAS sello1 per_in.

  (* Para evitar el uso de "Included PERSONAS", podemos adicionar una nueva definicion *)
  Definition INC_PER := Included PERSONAS.

  (* Se asume como hipotesis los tres invariantes *)
  Hypotheses (H1: INV_1)  (H2: INV_2)  (H3: INV_3).

  (* Se puede probar facilmente que si una persona tiene un sello, entonces
     la persona esta dentro de la oficina. Utilizaremos la siguiente definicion de la libreria:
     Definition In (A:Ensemble) (x:U) : Prop := A x.*)
  Definition IN_PER := In PERSONAS.
  Lemma sello_implica_in : forall x:PERSONAS,
     (IN_PER sello1 x \/  IN_PER sello2 x \/ IN_PER sello3 x) -> IN_PER per_in x.
    intro per. intro H4.
    destruct H4 as [H41 | H41']. (* Si tiene el sello 1 se puede utilizar el INV_3 *)
    apply H3; assumption.
    destruct H41' as [H42 | H42'].
    apply H3; apply H2 ; assumption.
    apply H3 ; apply H2; apply H1; assumption.
  Qed.
End Oficina.
```

Figura 6.4: Especificación de la Oficina Burocrática (ver Ejemplo 6.8) .

```
Require Import Coq.Sets.Ensembles.

Section CentroVotacion.
  (* Definicion de Tipos *)
  Variable PERSONA CENTRO MESA : Set.

  (* Personas Inscritas para votar *)
  Variable Inscritos : Ensemble PERSONA.

  (* Personas dentro del centro *)
  Variable Per_In : PERSONA -> CENTRO -> Prop.

  (* Ubicaciones de las mesas en los centros de votacion *)
  Variable Ubicacion: MESA -> CENTRO.

  (* Por cada persona inscrita para votar se debe conocer en que mesa lo debe hacer*)
  Variable Mesa_Persona : PERSONA -> MESA -> Prop.

  (* Se registra las personas que ya votaron *)
  Variable Votantes : Ensemble PERSONA.

  (* Una persona no puede estar dentro de dos centros diferentes *)
  Definition INV_1 : Prop :=
    forall (p:PERSONA) (c c' : CENTRO), Per_In p c -> Per_In p c' -> c = c'.

  (* Una persona no puede estar dentro del centro si no esta inscrita en el mismo *)
  Definition INV_2: Prop :=
    forall (p: PERSONA) (c: CENTRO),
      Per_In p c ->
      (In PERSONA Inscritos p /\  exists m:MESA, c = Ubicacion m /\  Mesa_Persona p m).

  (* Cada una de las personas inscritas para votar debe tener una mesa asignada *)
  Definition INV_3 : Prop :=
    forall p:PERSONA, In PERSONA Inscritos p <-> exists m:MESA, Mesa_Persona p m.

  (* La mesa que se asigna a una persona es unica *)
  Definition INV_4 : Prop:=
    forall (p:PERSONA) (m m': MESA), Mesa_Persona p m -> Mesa_Persona p m' -> m=m'.

  (* Si una persona voto es porque esta inscrita *)
  Definition INV_5 :Prop := Included PERSONA Votantes Inscritos.

End CentroVotacion.
```

Figura 6.5: Especificación del centro de votación (ver Ejemplo 6.9) .

```
Variable PERSONAS LABORATORIO GRUPO : Set. (* TIPOS *)
Section Sistema_Seguridad.
  (* Personas que pertenecen al centro *)
  Variable Miembro : PERSONAS -> Prop.

  (* Acceso a un  laboratorio*)
  Variable Acceso: PERSONAS -> LABORATORIO -> Prop.

  (* Las personas que pertenecen a un grupo *)
  Variable In_Grupo : PERSONAS -> GRUPO -> Prop.

  (* Los labotaorios que tiene acceso el grupo de personas *)
  Variable Permisos_Grupo : GRUPO -> LABORATORIO -> Prop.

  (* Personas dentro del centro y dentro de un laboratorio *)
  Variable In_Centro : PERSONAS -> Prop.
  Variable In_Laboratorio: PERSONAS -> LABORATORIO -> Prop.

  (* A cada laboratorio tiene al menos acceso una persona y
   cada persona tiene acceso a almenos un  laboratorio *)
  Definition INV_1 :Prop := forall p:PERSONAS, Miembro p->
              exists l:LABORATORIO, Acceso p l.
  Definition INV_2 :Prop := exists l:LABORATORIO, forall p:PERSONAS,  Acceso p l.

  (* Si la persona esta en el centro es porque pertenece al centro *)
  Definition INV_3 : Prop := forall p:PERSONAS, In_Centro p -> Miembro p.

  (* Si la persona esta en un laboratorio, debe estar dentro dentro del centro *)
  Definition INV_4:Prop := forall (p:PERSONAS) (l:LABORATORIO),
          In_Laboratorio p l ->  In_Centro p.

  (* Si la persona esta en un laboratorio debe tener permiso para ello *)
  Definition INV_5: Prop := forall (p:PERSONAS) (l:LABORATORIO),
      In_Laboratorio p l ->  Acceso p l.

  (* El grupo debe tener acceso a almenos un laboratorio *)
  Definition INV_6: Prop := forall g:GRUPO, exists l:LABORATORIO, Permisos_Grupo g l.

  (* Si la persona pertenece a un grupo,
     entonces tiene acceso a todas los laboratorios del grupo *)
  Definition INV_7: Prop:= forall (p:PERSONAS) (g:GRUPO),
      In_Grupo p g ->  forall l:LABORATORIO, Permisos_Grupo g l -> Acceso p l.

  (* Una persona no puede estar en dos laboratorios al tiempo *)
  Definition INV_8: Prop := forall (p:PERSONAS) (l l': LABORATORIO),
         (In_Laboratorio p l /\ In_Laboratorio p l') -> l=l'.
End Sistema_Seguridad.
```

Figura 6.6: Especificación del sistema de control de acceso (ver Ejemplo 6.10) .

```
Variable PERSONA CONTENIDO TIPO_CONTENIDO : Set.

Section RedSocial.
  (* Definicion de predicados *)
  Variables (Usuarios : PERSONA -> Prop) (* Usuarios registrados *)
            (Amigos : PERSONA -> PERSONA -> Prop) (* Relacion de amistad *)
            (Contenidos : PERSONA -> CONTENIDO -> Prop) (* Contenidos publicados por un usuario *)
            (TipoContenido : CONTENIDO -> TIPO_CONTENIDO) (* Cada contenido tiene un tipo *)
            (Acceso : PERSONA -> CONTENIDO -> Prop). (* Si un ususario tiene acceso a un contenido *)

  Variable privado compartido publico : TIPO_CONTENIDO. (* Los unicos posibles tipos de contenido *)

  (* Solamente los usuarios registrados pueden aparecer en el dominio (y/o rango) de Amigos, Contenidos y Acceso *)
  Definition INV_1 : Prop := forall p p':PERSONA, Amigos p p' -> (Usuarios p /\ Usuarios p').
  Definition INV_2 : Prop := forall p: PERSONA, forall c:CONTENIDO, Contenidos p c -> Usuarios p.
  Definition INV_3 : Prop := forall p: PERSONA, forall c:CONTENIDO, Acceso p c -> Usuarios p.

  (* Un contenido no puede ser adicionado por dos usuarios distintos (i.e., Contenidos es una funcion) *)
  Definition INV_4 : Prop := forall p p':PERSONA, forall c: CONTENIDO,
                              Contenidos p c -> Contenidos p' c -> p=p'.

  (* La relacion de amistad es simetrica *)
  Definition INV_5 :Prop := forall p p': PERSONA, Amigos p p' -> Amigos p' p.

  (* Un contenido solo puede ser privado, compartido o publico *)
  Definition INV_6 : Prop := forall c:CONTENIDO, forall t: TIPO_CONTENIDO,
                              t= TipoContenido c -> (t = privado \/ t=publico \/ t=compartido).

  (* p es la unica persona con acceso al contenido c *)
  Definition Acceso_Unico (c: CONTENIDO) (p: PERSONA) := forall  p':PERSONA, Acceso p' c -> p=p'.
  (* Si el contenido es privado, solamente lo puede ver quien lo creo *)
  Definition INV_7: Prop := forall (p:PERSONA) (c:CONTENIDO),
                             Contenidos p c -> privado = TipoContenido c ->  (Acceso p c /\ Acceso_Unico c p).
  (* Los amigos de p tienen acceso al contenido c *)
  Definition Acceso_Amigos (c: CONTENIDO) (p: PERSONA) := forall  p':PERSONA, Amigos p p' -> Acceso p' c.
  Definition INV_8: Prop := forall (p:PERSONA) (c:CONTENIDO),
                             Contenidos p c -> compartido = TipoContenido c ->  (Acceso p c /\ Acceso_Amigos c p).

  (* Ahora definimos la clausura transitiva de la relacion Amigos *)
  Inductive Amigos_Amigos  (p: PERSONA) : PERSONA -> Prop :=
    | base (p': PERSONA) : Amigos p p' -> Amigos_Amigos p p'
    | paso_ind (p' p'' : PERSONA): Amigos_Amigos p p' -> Amigos_Amigos p' p'' -> Amigos_Amigos p p''.
  (* Este tipo de definiciones se pueden encuentrar en la libreria Coq.Relations.Relation_Operators *)
  (* Los amigos de los amigos de p tienen acceso al contenido c *)
  Definition Acceso_Amigos_Amigos (c: CONTENIDO) (p: PERSONA) :=
    forall  p':PERSONA, Amigos_Amigos p p' -> Acceso p' c.
  Definition INV_9:Prop := forall (p:PERSONA) (c:CONTENIDO),
                            Contenidos p c -> publico = TipoContenido c ->  (Acceso p c /\ Acceso_Amigos_Amigos c p).

  Hypotheses (H1 : INV_1) (H2 : INV_2) (H3 : INV_3) (H4 : INV_4) (H5 : INV_5)
            (H6 : INV_6) (H7 : INV_7) (H8 : INV_8) (H9 : INV_9).
... (* Propiedades *)
End RedSocial.
```

Figura 6.7: Especificación del sistema de la red social (ver Ejemplo 6.11) .

```
....Seccion RedSocial ...
(* Asuma que todo usuario tiene al menos un contenido.
Entonces, todo usuario tiene al menos acceso a un contenido *)

  Theorem T1: (forall p:PERSONA, Usuarios p -> exists c:CONTENIDO, Contenidos p c) ->
              forall p:PERSONA, Usuarios  p -> exists c:CONTENIDO, Acceso p c.
    intros H10 p H11.
    assert (H12: exists c:CONTENIDO, Contenidos p c).
    auto.
    destruct H12 as [c H13].
    exists c.
    assert (H14: TipoContenido c = privado \/ TipoContenido c=publico \/ TipoContenido c=compartido).
    apply H6 with (c:=c) (t:=TipoContenido c).
    trivial.
    destruct H14 as [H15 | H16].
    apply H7.
    assumption. eauto.
    destruct H16 as [H17 | H18].
    apply H9.
    assumption. eauto.
    apply H8.
    assumption. eauto.
  Qed.

  (* Asuma ahora que p1 publica un contenido publico y que p1 es amigo de p2.
  A su vez, p2 es amigo de p3. Se debe poder mostrar que p3 puede ver el contenido *)
  Section Transitividad.
    Variable p1 p2 p3: PERSONA.
    Variable c: CONTENIDO.
    Hypotheses (H1': Contenidos p1 c)
               (H2': Amigos p1 p2)
               (H3': Amigos p2 p3)
               (H4': publico = TipoContenido c).

    Theorem T2: Acceso p1 c /\ Acceso p2 c /\ Acceso p3 c.
      split.
      apply H9;auto.
      assert (H5':Amigos_Amigos p1 p2).
      constructor;assumption.
      assert (H6':Amigos_Amigos p1 p3).
      constructor 2 with (p':=p2).
      eauto.
      constructor; assumption.
      assert (H7': Acceso_Amigos_Amigos c p1).
      apply H9;assumption.
      split; apply H7';assumption.
    Qed.
  End Transitividad.

    (* Privacidad: Si p1 publica algo privado, entonces nadie diferente a el mismo lo puede ver *)
  Section Privacidad.
    Variable p1: PERSONA.
    Variable c: CONTENIDO.
    Hypotheses (H1': Contenidos p1 c)
               (H2': privado = TipoContenido c).

    Theorem T3: forall p:PERSONA, p<>p1 -> ~ Acceso p c.
      intros p H3'.
      intro H4'.
      assert (H5': Acceso_Unico c p1).
      apply H7; assumption.
      unfold Acceso_Unico in H5'.
      absurd (p1=p);eauto.
    Qed.
  End Privacidad.
```

Figura 6.8: Propiedades de la red social (ver Ejemplo 6.11) .

`induction` x	x debe ser una variable de un tipo inductivo T. La táctica genera los casos de acuerdo a los constructores de T.
`induction` P	P es un predicado definido inductivamente. La táctica genera los casos de acuerdo a los constructores de P.
`induction` H	H es el identificador de una de las hipótesis que contiene una definición inductiva.
`simple induction` x	la variable x es de tipo T y está cuantificada universalmente. Esta táctica introduce la variable x (eliminando el $\forall$) y genera los casos necesarios.
`inversion` H	H es el identificador de una de las hipótesis que contiene una definición inductiva. Genera los casos de acuerdo a la definición inductiva en H.
`discriminate`	Termina cualquier prueba cuando se tiene como hipótesis una igualdad de la forma $t = t'$ donde t y t' son del mismo tipo pero generados a partir de diferentes constructores. Por ejemplo, $0 = s(x)$ para los números naturales o $nil = a :: x$ para las listas.
`injection` H	H es una hipótesis de la forma $t = t'$. La táctica introduce como hipótesis las igualdades que se pueden deducir a partir de $t = t'$. Por ejemplo, de $s(x) = s(s(y))$ se puede deducir que $x = s(y)$.

Cuadro 6.1: Tácticas utilizadas en pruebas inductivas.

6.5. Tipos Inductivos

En esta sección definiremos algunas estructuras de datos y operaciones sobre las mismas y probaremos la correctitud de dichas operaciones. Además, estudiaremos algunas técnicas de prueba por inducción en Coq (ver resumen en la Tabla 6.1).

Empezaremos por un ejemplo simple de definición inductiva de un tipo. En el siguiente código se declara el tipo inductivo *color_sem* que tiene 3 posibles constructores:

```
Inductive color_sem : Set :=
  rojo
| amarillo
| verde.
```

Con esta declaración se define *color_sem_ind* que permite realizar pruebas inductivas que incluyan variables tipo *color_sem*:

```
Check color_sem_ind.
----
color_sem_ind
     : forall P : color_sem -> Prop,
       P rojo -> P amarillo -> P verde -> forall c : color_sem, P c
```

Dicha definición se puede leer como: "*Para todo predicado P que recibe como argumento un objeto tipo color_sem, si se prueba que P es verdadero para rojo, amarillo y verde entonces se concluye que P es cierto para todo x de tipo color_sem*". Esto claramente refleja nuestra intuición que para probar que algo es cierto acerca de un objeto tipo *color_sem*, debemos probarlo para cada uno de sus posibles valores ("constructores").

Podemos probar fácilmente que cualquier objeto tipo *color_sem* solamente puede tomar tres valores diferentes.

```
Theorem Color_Sem_Values : forall x:color_sem, x=rojo \/ x= verde \/ x = amarillo.
  simple induction x.
  (* Genera los casos que se deben tener en cuenta
     de acuerdo a los constructores del tipo color_sem *)
  (* Primer caso: x= rojo *)
  left.
  trivial.
  (* Segundo caso: x=amarillo. *)
  right; right; trivial.
  (* Tercer caso: x=verde *)
  right;left;trivial.
Qed.
```

De hecho, la anterior prueba se puede realizar automáticamente con la táctica `auto`:

```
Theorem Color_Sem_Values' : forall x:color_sem, x=rojo \/ x= verde \/ x = amarillo.
  simple induction x;auto.
Qed.
```

6.5.1. Números Naturales

Los números naturales se puede definir a partir de la constante *cero* y la función *sucesor*:

```
 Inductive NAT : Set :=
   zero:NAT
 | suc (n':NAT): NAT.

 Check NAT_ind.
```

De hecho, esta definición ya existe en Coq como "nat".

```
Check nat.
Check nat_ind.
```

Podemos definir funciones (recursivas) que operan sobre tipos inductivos. Por ejemplo, la siguiente función calcula la suma de dos naturales:

```
Fixpoint Suma (n m :nat): nat :=
  match n with
      O => m
    | S n'  => S ( Suma n' m)
  end.

(* Ejemplo de uso *)
Compute Suma 3 2.
```

Utilizando inducción, podemos probar algunas propiedades de la función *Suma*:

```
(* Para todo n, n+0 = n *)
Lemma plus_n_O : forall n:nat, n = n + 0.
  intro n.
  induction n. (* Casos para n *)
  simpl. (* simplica la expresion --realizando los calculos respectivos *)
  trivial.
  trivial.
Qed.
```

También es posible probar este teorema con la táctica auto.

```
Lemma plus_n_O' : forall n:nat, n = n + 0.
  simple induction n;auto.
Qed.
```

Varios teoremas sobre los números naturales ya se encuentran probados y pueden ser utilizados durante las pruebas [2]:

```
Require Import Arith.

Check eq_S. (* S preserva igualdad *)
----
  eq_S
       : forall x y : nat, x = y -> S x = S y
----

Search nat.  (* definiciones sobre nat *)
----
S: nat -> nat
O: nat
tail_plus: nat -> nat -> nat
tail_mult: nat -> nat -> nat
NPeano.square: nat -> nat
NPeano.sqrt_iter: nat -> nat -> nat -> nat -> nat
NPeano.sqrt: nat -> nat
NPeano.shiftr: nat -> nat -> nat
NPeano.shiftl: nat -> nat -> nat
.....
----

SearchPattern (_ + _ = _).
----
plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
plus_assoc: forall n m p : nat, n + (m + p) = n + m + p
plus_Snm_nSm: forall n m : nat, S n + m = n + S m
plus_Sn_m: forall n m : nat, S n + m = S (n + m)
plus_O_n: forall n : nat, 0 + n = n
plus_0_r: forall n : nat, n + 0 = n
....
----
```

[2]Ver por ejemplo http://coq.inria.fr/distrib/V8.4/stdlib/Coq.Arith.Plus.html

Si se quiere que un teorema ya probado sea utilizado por la táctica `auto`, debe utilizarse la siguiente instrucción:

```
Hint Resolve plus_n_O.
```

Por ejemplo, aquí primero probamos un resultado auxiliar que luego se utiliza en la prueba del teorema *plus_com*.

```
Lemma plus_n_S : forall n m:nat, S (n + m) = n + S m.
  intros n m.
  induction n.
  auto.
  simpl.
  auto.
Qed.

Hint Resolve plus_n_S .
Lemma plus_com : forall n m:nat, n + m = m + n.
  intros n m.
  induction n.
  auto.
  simpl.
  rewrite IHn. (* ahora queda igual que plus_n_s *)
  auto.
Qed.
Hint Resolve plus_com.
```

Ejercicio 6.1 (Pruebas sobre los números naturales)**.** *Pruebe los siguientes teoremas en Coq:*

```
Lemma plus_plus_n_m: forall n m:nat, S n + S m = S (S (n + m)).
Lemma conmutatity: forall x y z: nat, x + (y + z) = (x + y) + z.
```

La siguiente función retorna *true* cuando el argumento es cero:

```
Definition Is_zero (n:nat) : Prop :=
  match n with
      O => True
    | S n' => False
  end.
```

y podemos probar los siguientes teoremas:

```
Theorem zero_is_zero : Is_zero O.
  simpl.
  trivial.
Qed.

(* Is_Zero del sucesor de n debe ser falso *)
(*  simple induction n;auto.  tambien prueba este teorema *)
Lemma S_Is_not_zero : forall n:nat, ~Is_zero (S n).
  intro n.
  intro H.
  simpl in H. (* Simplificar alguna de las hipotesis.
```

```
                    En particular, Is_zero (S n)  retorna falso *)
  trivial.
Qed.
```

De manera similar, podemos definir una función que solamente retorna falso cuando el argumento es cero:

```
Definition Is_S (n:nat) := match n with
                               0 => False
                             | S n' => True
                           end.
```

y podemos probar lo siguiente utilizando la táctica `discriminate`:

```
Lemma no_confusion : forall n:nat, 0 <> S n.
  intro n.
  intro H.
  (* En este caso, se tiene como hipotesis 0 = S n.
    Como se utilizan dos constructores diferentes en una
    igualdad, es decir, 0 = S n, la tactica discriminate
    prueba cualquier objetivo
    (puesto que 0 = S n implica falso *)
  discriminate.
Qed.
```

Así como definimos inductivamente tipos, podemos definir predicados de manera inductiva. Por ejemplo, la definición inductiva de $\leq$ en los números naturales es la siguiente:

```
Inductive menor_igual (n:nat) : nat -> Prop :=
   le_n : menor_igual n n
  | le_S : forall m:nat, menor_igual n m -> menor_igual n (S m).
```

Esta definición ya existe como "*le*" en Coq. A continuación un ejemple de una prueba utilizando dicha definición:

```
Lemma le_n_S : forall n m:nat, le n m -> le (S n) (S m).
  intros n m.
  intro H.
  (* Para expandir la definicion inductiva de la hipotesis H: *)
  induction H.
  constructor 1.
   (* alternativamente, se puede utilizar "apply le_n", es decir, el
       nombre del constructor. *)
  (* para probar que le (s n) (s n), basta con utilizar el primer
      constructor de le *)
  (* la prueba de le (S n) (S ( S m)) resulta de utilizar el 2do constructor *)
  constructor 2.
  trivial.
Qed.
```

6.5.2. Estructuras de Datos

En Coq las listas están definidas en el módulo List:

```
Require Import Coq.Lists.List.
Search list.
```

Al igual que en las secciones anteriores, podemos definir funciones inductivas sobre listas:

```
(* Reverso de una lista *)
Fixpoint Rev (l : list nat) :list nat :=
  match l with
      nil => nil
    | a::l' =>  (Rev l') ++  (a::nil) end.

(* Ejemplo de uso *)
Compute Rev (1::2::3::4::nil).

(* Numero de elementos en la lista *)
Fixpoint Length (l: list nat) : nat:=
  match l with
      nil => 0
    | _::l' => 1 + Length l'
  end.

(* Ejemplo de uso *)
Compute Length (1::5::10::nil).
```

La definición de reverso está definida en Coq como *reverse* y la longitud de una lista como *length*.

Podemos probar propiedades sobre las lista utilizando su definición inductiva:

```
Lemma S_Cons: forall l: list nat, forall n: nat,  length (n::l) = S (length l).
  intros l n.
  simpl.
  trivial.
Qed.

(* l++l' representa la concatención de las listas l y l'
   Este teorema prueba que la longitud de la concatenacion de dos listas
    es la suma de las longitudes de la lista *)
Lemma Append_L: forall l l' : list nat, length (l++l') = length l + length l'.
  simple induction l.
  intros.
  auto.
  intros.
  simpl.
  (* auto funcion aqui por el teorema eq_S: Si x = x' entonces S x = S x' *)
  auto.
Qed.
```

Ejercicio 6.2 (Pruebas sobre listas)**.** *Pruebe los siguientes teoremas:*

```
Lemma Add_Sides: forall l: list nat, forall a a':nat,
                             length (a::l) = length (l ++ (a'::nil)).
```

Ahora incluimos los anteriores teoremas para que sean tenidos en cuenta por la táctica `auto`.

```
Hint Resolve S_Cons Append_L Add_Sides.
```

Con los resultados anteriores podemos probar que la lista l y su reverso tienen la misma longitud.

```
Lemma Igual_Ln: forall l: list nat, length l = length (Rev l).
  intro l.
  induction l.
  auto.
  simpl.
  induction Rev.
  simpl.
  auto.
  simpl.
  rewrite IHl. (* auto requiere primero que se reescriba la expresion *)
  auto.
Qed.
```

Ahora vamos a definir una función que retorna *true* si un elemento x está en la lista l. Esta función corresponde a la función In de la librería de Coq:

```
Fixpoint Is_In_L (l: list nat)(n: nat) : Prop :=
  match l with
      nil => False
    | m::l' => m=n \/ Is_In_L l' n
  end.
```

En la siguiente definición inductiva, especificamos cuando una lista es prefijo de otra.

```
Inductive prefix (list1 list2 : list nat): Prop :=
  sl_b : list1=nil -> prefix list1 list2
| sl_r (list1' list2' :list nat) (n:nat) :
  list1=n::list1' -> list2=n::list2' -> prefix list1' list2'
                       -> prefix list1 list2.
```

Esta definición establece las condiciones para determinar que $list1$ es un prefijo de $list2$ y puede ser utilizada en las pruebas. Por el contrario, la función (recursiva) Is_Prefix que aparece a continuación retorna *true* o $false$ si el primer argumento es un prefijo del segundo:

```
Fixpoint Is_Prefix (list1 list2 : list nat): Prop :=
  match list1, list2 with
      nil,nil => True
    | nil, m::list2' => True
    | n::list1' , nil => False
    | n::list1' , m::list2' => n=m /\ Is_Prefix  list1' list2'
  end.
```

Note la diferencia entre $prefix$ que solo considera los casos "verdaderos" mientras que Is_Prefix, para que esté bien definida, debe considerar los casos "verdaderos" y los casos "falsos". La definición inductiva puede considerarse como la **especificación** mientras que la segunda como una **implementación**. Más adelante mostraremos la correctitud de la implementación con respecto a su especificación.

A continuación probamos que toda lista es prefijo de si misma.

```
Theorem Prefix_ID: forall l : list nat, prefix l l.
  intro l.
  induction l.
  (* El constructor 1 de sublist resuelve este caso *)
  constructor 1;trivial.
   (* El segundo constructor necesita informacion
      explicita de las sustituciones que debe realizar *)
  constructor 2 with (list1':=l) (list2':=l) (n:=a); trivial.
Qed.
```

Ejercicio 6.3. *Pruebe los siguientes teoremas:*

```
Theorem prefix_append: forall l l' : list nat, prefix l (l++l').
Theorem length_nil : forall l:list nat, length (nil:list nat) <= length l.
```

y adiciónelos para que sean consideramos con la táctica ***auto****:*

```
Hint Resolve Prefix_ID prefix_append length_nil.
```

El siguiente teorema establece la relación entre los prefijos de una lista y sus longitudes:

```
Theorem prefix_length: forall l l': list nat, prefix l l' ->
                           length l <= length l'.
  intros l l' H.
  induction H.
  rewrite H.
  auto.
  rewrite H.
  rewrite H0.
  simpl.
  apply le_n_S with (n:= (length list1')) (m:= (length list2')).
  trivial.
Qed.
```

Por otro lado, podemos probar lo siguiente:

```
Theorem prefix_preservation : forall x y x' y': list nat, forall n:nat,
                        x = n::x' -> y= n::y' -> prefix x y -> prefix x' y'.
  intros x y x' y' n H1 H2 H3.
  induction H3.
  rewrite H in H1. (* para obtener nil = n::x'*)
  discriminate.
  rewrite H in H1.
  assert (x'=list1').
  injection H1;auto.
```

```
  assert (y'=list2').
  rewrite H2 in H0.
  injection H0;auto.
  rewrite H4, H5; assumption.
Qed.

Hint Resolve prefix_preservation.
```

Ejercicio 6.4. *Pruebe el siguiente teorema utilizando inducción sobre las dos listas:*

```
Theorem prefix_prefix_eq: forall l l': list nat,
             prefix l l' -> prefix l' l -> l = l'.
```

Ahora podemos probar la correctitud de la implementación de "is_prefix con respecto a su especificación, es decir:

$$Is_Prefix(l_1, l_2) \equiv prefix(l_1, l_2)$$

Primero la implicación de derecha a izquierda:

```
Theorem Correctness_prefix: forall l1 l2:list nat, prefix l1 l2 -> Is_Prefix l1 l2.
  intros l1 l2.
  intro H.
  induction H.
  rewrite H.
  induction list2.
  simpl;trivial.
  simpl;trivial.
  rewrite H. rewrite H0.
  simpl.
  auto.
  Qed.
```

Ahora la implicación de izquierda a derecha:

```
Theorem Soundness_Prefix: forall l1 l2:list nat,  Is_Prefix l1 l2 -> prefix l1 l2.
  simple induction l1.
  intros.
  apply sl_b;trivial.
  intros.
  induction l2.
  (* Calcular la funcion Is_prefix *)
  vm_compute in H0.
  tauto.
  (* Como H0:  Is_Prefix (a :: l) (a0 :: l2)
     entonces a debe ser igual a a0 *)
  assert (a=a0).
  vm_compute in H0.
  tauto.
  rewrite H1.
  apply sl_r with (list1' := l) (list2' := l2) (n:=a0);auto.
  apply H.
  rewrite H1 in H0.
  vm_compute in H0.
  intuition.
Qed.
```

Ejercicio 6.5. *Defina inductivamente el predicado* `sorted`/1 *que determina si una lista de números naturales está ordenada.*

Árboles Binarios El tipo de un árbol binario se puede definir de manera inductiva como se muestra a continuación:

```
Inductive tree : Set :=
| vacio (* Caso base *)
(* Un arbol a partir del subarbol derecho e izquierdo *)
| nodo : tree -> nat -> tree -> tree.
```

También, podemos definir de manera inductiva si un elemento se encuentra en el árbol (que no está necesariamente ordenado):

```
Inductive is_in_T (x:nat)  : tree -> Prop :=
| t_base : forall (l r :tree), is_in_T x (nodo l x r)
| t_ind_l : forall (l r: tree) (y:nat), is_in_T x l -> is_in_T x (nodo l y r)
| t_ind_r : forall (l r: tree) (y:nat), is_in_T x r -> is_in_T x (nodo l y r).
```

Podemos escribir una función que dado un árbol determina si está vacío o no [3]:

```
Definition is_vacio (t:tree):Prop :=
  match t with
      vacio => True
    | nodo _ _ _ => False
  end.
```

Note que esta es una función que determina si el árbol está vacío o no y **no es** un predicado inductivo que determina cuando un árbol está vacío. Dicho predicado se definiría de la siguiente manera:

```
Inductive pred_is_vacio (t:tree): Prop:=
  t_vacio : t=vacio -> pred_is_vacio t.
```

Nuevamente, "*is_vacio*" es la implementación (que considera todos los casos) y "*pred_is_vacio*" es un predicado que nos sirve como especificación.

Ahora definimos de manera inductiva cuando un árbol binario se encuentra ordenado con el criterio usual de ordenamiento de árboles:

```
Inductive ordered_tree: tree -> Prop :=
  o_vacio : ordered_tree vacio
| o_ind   : forall (l r:tree) (x: nat), ordered_tree l -> ordered_tree r ->
                                        (forall y:nat, is_in_T y r -> x<y )
                                        -> (forall y:nat, is_in_T y l -> y<x )
                                        -> ordered_tree (nodo l x r ).
```

Una manera alternativa es proveer de manera explícita el argumento "t":

[3] El símbolo "_" representa cualquier valor.

```
Inductive ordered_t (t:tree):  Prop :=
    o_v : t = vacio -> ordered_t t
  | o_n (l r:tree) (x: nat) : t = nodo l x r -> ordered_t l  -> ordered_t r ->
                              (forall y:nat, is_in_T y r -> x<y )
                              -> (forall y:nat, is_in_T y l -> y<x ) ->
                              ordered_t t.
```

Ejercicio 6.6. *Defina de manera inductiva el predicado "altura_t" y la función getAltura. Además, pruebe los siguientes teoremas:*

```
Theorem altura_vacio: altura_t vacio 0.

Theorem altura_no_vacio: forall t:tree, forall h:nat,
                                        ~is_vacio t -> altura_t t h -> h>0.

Hint Resolve altura_vacio altura_no_vacio.
```

A continuación probamos que la altura de un árbol debe ser menor que la altura de sus subárboles. Para ello, vamos a requerir algunos resultados auxiliares acerca de la función " max":

```
Lemma max_s : forall n m:nat, n< S( max n m).
  intros n m.
  assert (n <= max n m).
  apply Max.le_max_l.
  apply le_lt_n_Sm;assumption.
Qed.

Lemma max_s_g_l : forall n m:nat, S( max n m) > n.
  intros n m.
  assert (n< S( max n m)).
  apply max_s.
  auto.
Qed.

Hint Resolve Max.max_comm.
Lemma max_s_g_r : forall n m:nat, S( max n m) > m.
  intros n m.
  assert (max n m = max m n).
  auto.
  rewrite H.
  assert (m< S( max m n)).
  apply max_s.
  auto.
Qed.
```

Ahora si probamos el teorema que relaciona la altura de los subárboles:

```
Theorem h_subarbol: forall t l r:tree, forall x hr h hl :nat,
            t= nodo l x r -> altura_t l hl ->
           altura_t r hr -> altura_t t h -> (h>hr /\ h > hl).
  intros.
  induction H2.
  rewrite H2 in H; discriminate.
```

```
  rewrite H2 in H;injection H; intros.
  assert (hl = hl0).
  rewrite H6 in H2_.
  apply h_eq with (t:=l);assumption.
  assert (hr = hr0).
  rewrite H4 in H2_0.
  apply h_eq with (t:=r);assumption.
  rewrite H3,H7,H8.
  simpl.
  split.
  apply max_s_g_r with (n:=hl0)(m:=hr0).
  apply max_s_g_l with (n:=hl0)(m:=hr0).
Qed.
```

Ejercicio 6.7. *Pruebe la correctitud de la función getAltura con respecto a su especificación altura_t (ver Ejercicio 6.6).*

6.6. Ejercicios

Por cada uno de los siguientes enunciados,

1. Defina claramente la signatura Σ que necesitaría para modelar la situación, así como los tipos de datos que requiere.
2. Explique brevemente el significado que le daría a los símbolos de Σ.
3. Describa la situación como un conjunto de fórmulas en FOL. Explique brevemente las fórmulas que propuso.
4. Escriba su modelo en Coq.

En algunos ejercicios se enuncia una proposición que debe ser establecida como teorema en Coq y probada.

Ejercicio 6.8 (Agentes y Tokens)**.**

Un sistema está conformado de agentes, e.j., $agentes = \{Alice, Bob\}$ y de piezas (o tokens) de información, e.j, $tokens = \{msg1, msg2, msg3\}$. Los agentes adicionan tokens de información e interesa saber quién adicioné cuál token. Dados dos mensajes m_1 y m_2 se dice que son de la misma fuente si los adicionó al sistema el mismo usuario. Un mensaje no puede ser adicionado por dos usuarios diferentes, es decir, los mensajes emitidos por los usuarios son diferentes a todos los otros mensajes. Se dice que el sistema es mono-usuario si todos los mensajes del sistema los ha puesto un único usuario.

Ejercicio 6.9 (Improvisación Musical)**.**

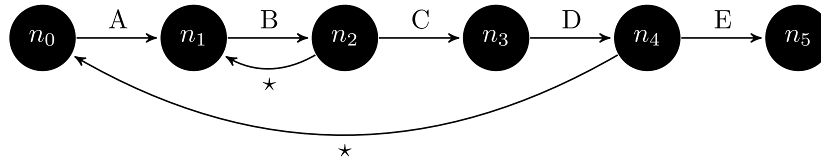


Figura 6.9: Ejemplo de un Factor Oracle.

En sistemas de improvisación musical asistida por computador, una estructura de datos que se utiliza frecuentemente es el *Factor Oracle* (FO) (ver Figura 6.9). En esta estructura, las notas se representan por las letras C,D,E,F,G,A,B. Cada nodo representa un estado y los arcos, etiquetados con notas, representan una nota tocada por el músico. Por ejemplo, el músico tocó B en el estado n_1. A estos arcos se les conoce como *Forward Links*. Además, el grafo se completa con *Backward links* que son arcos sin etiquetas y relacionan cualquier par de nodos del grafo. En realidad, los *backward links* satisfacen algunas otras propiedades que no vamos a especificar aquí. Los *forward link* satisfacen las siguientes propiedades:

1. Todo nodo es el origen o el destino de un *forward link*. Es decir, no hay nodos desconectados.
2. Existe un nodo al que no llega ningún *forward link* (n_0 en la figura)
3. Existe un nodo del que no salen *forward links* (n_5 en la figura).
4. De un nodo a lo más sale un *forward link*.
5. A un nodo llega a lo más un *forward link*.

El FO representa el *estilo* que ha aprendido el computador de acuerdo a los notas tocadas por el músico.

Ejercicio 6.10 (Recursos y Usuarios)**.**

Un sistema que controla recursos compartidos está conformado por usuarios y recursos. Los usuarios pueden hacer uso de varios recursos y un recurso siempre está siendo utilizado por un usuario. El uso de los recursos puede hacerse en dos modos o permisos:

- `unique`: Se debe garantizar que el usuario es el único usuario en el sistema haciendo uso del recurso.
- `share`: El recurso puede estar siendo utilizado (compartido) por varios usuarios al mismo tiempo.

Un recurso no puede ser accedido en dos modos distintos.

Ejercicio 6.11 (Vuelos y Conexiones)**.**

Se tiene información de los vuelos directos en un país. Se dice que dos ciudades están conectadas si hay un vuelo directamente entre ellas. Obviamente toda ciudad está conectada consigo misma (no hay necesidad de vuelo). Además, dos ciudades se conectan si hay un vuelo con escalas entre ellas. Se dice que una ciudad es un *hub* si está conectada con vuelos directos a todas las ciudades. Pruebe que para cualquier par de *hubs* c_1 y c_2 del país existe un vuelo directo entre c_1 y c_2.

Ejercicio 6.12 (Trazas). *Asuma dos programas P_1 y P_2 que están conformados por una lista no vacía de sentencias de la siguiente forma:*

$$P1 \;=\; \begin{array}{l} st_{11} \\ st_{12} \\ st_{13} \end{array} \qquad P2 \;=\; \begin{array}{l} st_{21} \\ st_{22} \\ st_{23} \\ st_{24} \end{array}$$

Note que los programas no necesariamente tienen la misma longitud (número de sentencias).

Una ejecución concurrente de P_1 y P_2 da como resultado una traza que contiene sentencias tanto de P_1 como de P_2. Para nuestro ejemplo, una posible traza sería:

$$[st_{11}\ st_{12}\ st_{21}\ st_{22}\ st_{13}\ st_{23}\ st_{24}]$$

Note que:

1. *El orden de las sentencias de cada programa se mantiene.*
2. *Las sentencias de los programas aparecen intercaladas, aunque una traza válida podría ser*

$$[st_{11}\ st_{12}\ st_{13}\ st_{21}\ st_{22}\ \ st_{23}\ st_{24}]$$

3. *Todas las sentencias de ambos programas se deben ejecutar.*

A manera de ejemplo, considere las siguientes secuencias que NO son trazas válidas para una ejecución concurrente de P_1 y P_2:

- $[st_{11}\ st_{13}\ st_{21}\ st_{22}\ st_{12}\ st_{23}\ st_{24}]$. *Aquí no se respeta el orden de P_1 ($st_{13} \rightarrow st_{12}$).*
- $[st_{11}\ st_{12}\ st_{21}\ st_{22}\ st_{23}\ st_{24}]$. *Aquí st_{13} no aparece.*

A partir de este enunciado debe:

1. *Definir un nuevo tipo para las sentencias.*
2. *Definir el tipo de los programas como listas de sentencias.*
3. *Definir (de manera inductiva) un predicado "$isTrace\ \ p_1\ \ p_2\ \ l_3$" que determine cuando l_3 es una traza valida a partir de los programas p_1 y p_2.*
4. *Especifique y pruebe en Coq las siguientes propiedades:*
   a) $isTrace\ \ p_1\ \ nil\ \ p_1$.
   b) $isTrace\ \ nil\ \ p_2\ \ p_2$.
   c) *Dados dos programas p_1 y p_2, su concatenación ("$p_1 ++ p_2$") es una traza válida.*
   d) *Invariante de longitud: El número de sentencias en la traza l_3 debe ser igual al número de sentencias en p_1 más el número de sentencias en p_2.*
   e) *Invariante de elementos: Todo elemento en p_1 tiene que aparecer en la traza l_3.*

6.7. Propiedades de los Sistemas de Pruebas en FOL

En esta sección probaremos algunas propiedades de los sistemas de pruebas de la Definición 6.1. Seguiremos los métodos de prueba utilizados en [NvP01].

6.7.1. Correctitud

Para probar que toda teorema del sistema $\longrightarrow$ es una tautología de la lógica clásica, debemos extender la noción de valuación en la Definición 4.7 para el caso de los cuantificadores.

Definición 6.5 (Valuación)**.** *Considere una función de valuación v que asigna un valor de verdad 0/1 a los símbolos de predicado del lenguaje.* [4] *Es decir, $v(p(x_1, ..., x_n)) = 0$ o $v(p(x_1, ..., x_n)) = 1$ si p es un símbolo de predicado de aridad n. Extendemos la función $[\![\cdot]\!]$ en la Definición 4.7 con los siguientes casos:*

$$\begin{array}{lcl}
[\![\bot]\!]_v & = & 0 \\
[\![p(x_1, ..., x_n)]\!]_v & = & v(x_1, ..., x_n), \ \textit{si } p \textit{ es una predicado de aridad } n \\
[\![\neg F]\!]_v & = & 1 - [\![F]\!]_v \\
[\![(F_1 \wedge F_2)]\!]_v & = & min([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \vee F_2)]\!]_v & = & max([\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![(F_1 \supset F_2)]\!]_v & = & max(1 - [\![F_1]\!]_v, [\![F_2]\!]_v) \\
[\![\forall x.F]\!]_v & = & inf(F[x_i/x]) \\
[\![\exists x.F]\!]_v & = & sup(F[x_i/x])
\end{array}$$

donde inf es el infimo y sup es el supremo.

De manera similar al caso proposicional, lo que debemos probar es que si el secuente $\Gamma \longrightarrow \Delta$ es probable entonces, para toda valuación v, $[\![\bigwedge \Gamma]\!]_v \leq [\![\bigvee \Delta]\!]_v$ (ver Definición 4.8).

Teorema 6.3 (Correctitud –soundness–)**.** *Si el secuente $\Gamma \longrightarrow \Delta$ es derivable, entonces es válido.*

Demostración. La prueba procede por inducción en la altura de la derivación y un análisis de casos de la última regla aplicada en la derivación. Los casos para todos los conectivos (diferentes a los cuantificadores) es igual a la prueba del Teorema 4.2 en la lógica proposicional. Los casos de los cuantificadores son los siguientes:

- **Caso** $\forall_I$. Si la prueba termina de la siguiente manera:

  $$\frac{\forall x.F, F[y/x], \Gamma \longrightarrow \Delta}{\forall x.F, \Gamma \longrightarrow \Delta} \ \forall_I$$

  Podemos asumir por hipótesis inductiva que

  $$min([\![\forall x.F]\!]_v, [\![F[y/x]]\!]_v, [\![\Gamma]\!]_v) \leq [\![\bigvee \Delta]\!]_v$$

  Por definición de $[\![\forall x.F]\!]_v$, sabemos que:

  $$[\![\forall x.F]\!]_v \leq [\![F[y/x]]\!]_v$$

  y por lo tanto, $min([\![\forall x.F]\!]_v, [\![\Gamma]\!]_v) \leq [\![\bigvee \Delta]\!]_v$.

- **Caso** $\forall_D$. La derivación termina de la siguiente forma:

  $$\frac{\Gamma \longrightarrow \Delta, F[y/x]}{\Gamma \longrightarrow \Delta, \forall x.F} \ \forall_D$$

  donde y no ocurre ni en Γ ni en Δ. Por inducción sabemos que

  $$[\![\bigwedge \Gamma]\!]_v \leq max(\bigvee \Delta, [\![F[y/x]]\!]_v)$$

[4] En esta definición no se consideran símbolos de funciones y constantes en la signatura. Por tanto, los términos corresponden únicamente a variables.

Como y no ocurren en Γ, la relación $\leq$ se preserva así consideremos el mínimos de todas las posibles valuaciones de $F[y/x]$. Es decir,

$$[\![\bigwedge \Gamma]\!]_v \leq inf_y(max([\![\bigvee \Delta]\!]_v, [\![F[y/x]]\!]_v))$$

Como y tampoco ocurre en Δ, concluimos que $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, inf_y[\![F[y/x]]\!]_v)$ y por tanto, $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, [\![\forall x.F]\!]_v)$.

- **Caso** $\exists_I$. La derivación termina con:

$$\frac{F[y/x], \Gamma \longrightarrow \Delta}{\exists x.F, \Gamma \longrightarrow \Delta} \exists_I$$

donde y no ocurre ni en Γ ni en Δ. Por inducción sabemos que $min([\![F[y/x]]\!]_v, [\![\Gamma]\!]_v) \leq [\![\bigvee \Delta]\!]_v$. Similar al caso $\forall_D$, como y no ocurre ni en Γ ni en Δ, podemos escoger cualquier valor arbitrario para $F[y/x]$ y la desigualdad $min([\![F[y/x]]\!]_v, [\![\Gamma]\!]_v) \leq [\![\bigvee \Delta]\!]_v$ se conserva. Por lo tanto, $min(sup_y([\![F[y/x]]\!]_v), [\![\Gamma]\!]_v) \leq [\![\Delta]\!]_v$ y por tanto, $min([\![\exists x.F]\!]_v, [\![\Gamma]\!]_v) \leq [\![\bigvee \Delta]\!]_v$.

- **Caso** $\exists_D$. Por lo tanto, la derivación debe terminar con:

$$\frac{\Gamma \longrightarrow \Delta, \exists x.F, F[y/x]}{\Gamma \longrightarrow \Delta, \exists x.F} \exists_D$$

y por hipótesis inductiva, $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, sup([\![F[z/x]]\!]_v), [\![F[y/x]]\!]_v)$. Trivialmente entonces tenemos $[\![\bigwedge \Gamma]\!]_v \leq max([\![\bigvee \Delta]\!]_v, sup([\![F[z/x]]\!]_v))$.

□

6.7.2. Completitud

Para la lógica proposicional logramos tener un método efectivo para determinar si una fórmula era válida o no. Dicho procedimiento fue utilizado en la prueba de correctitud del cálculo de secuentes para dicha lógica de la siguiente manera:

1. Subir en la derivación y en cada paso, eliminar el conectivo principal.
2. Como el número de conectivos es finito y se reduce en cada derivación, eventualmente se obtienen hojas sin conectivos, es decir, con variables proposicionales y/o el símbolo $\perp$.
3. Si todas las hojas son instancias de la regla **axioma** o $\perp_I$, entonces el secuente es probable. De lo contrario, la hoja de la forma:

$$P_1, ..., P_n \longrightarrow Q_1, ..., Q_m$$

donde los conjuntos $P_1, ..., P_n$ y $Q_1, ..., Q_m$ nos permite encontrar una valuación [5] que refutaba el secuente.

Dicho procedimiento no funciona en general para la lógica de primer orden. En particular, la condición de terminación "*el número de conectivos se reduce en cada derivación*" en (2) no necesariamente se cumple en la lógica de predicados. Así que el resultado de completitud para FOL afirma que si la fórmula es válida, el sistema es capaz de probarlo. Si no lo es, el procedimiento probablemente no termina y por tanto, no obtenemos un algortimo efectivo para decidir si una fórmula es válida o no.

Los pasos entonces que vamos a seguir son:

1. Construir el árbol de la derivación de acuerdo con una estrategia.
2. Si el árbol es finito, y todas sus hojas son instancias de **axioma** o $\perp_I$, el secuente es probable.
3. Si el árbol tiene una rama **infinita**, se obtiene una valuación que refuta el secuente.

[5] Dicha valuación asigna 1 a cada P_i y 0 a cada Q_i de tal manera que $[\![\bigwedge P_i]\!]_v > [\![\bigvee Q_i]\!]_v$.

Construcción del Árbol Una posible estrategia para la construcción del árbol de la derivación se presenta a continuación:

1. Parar cuando todas las hojas del secuente son instancias de **axioma** o $\bot_I$.
2. Si hay una hoja (con conectivos) que no es una instancia de **axioma** o $\bot_I$ aplicar las reglas de $\longrightarrow$:
    a) Aplicar $\wedge_I$ y luego $\wedge_D$.
    b) Aplicar $\vee_I$ y luego $\vee_D$.
    c) Aplicar $\supset_R$ y luego aplicar $\supset_I$.
    d) Aplicar $\forall_D$ y luego $\forall_I$ instanciando $\forall x.F$ con $F[y/x]$ por cada variable y que aparezca (libre) en el secuente.
    e) Aplicar $\exists_I$ y luego $\exists_L$ por cada variable que ocurra (libre) en el secuente. [6]
3. Repetir (2) hasta que todas las hojas sean instancias de **axioma** o $\bot_I$.

Si el paso (3) se cumple, el secuente es probable.

Árboles Infinitos y Refutación Asumamos que el procedimiento anterior no termina, es decir, obtenemos un árbol **infinito** de la derivación. Es importante notar que cada nodo del árbol tiene un número finito de sucesores. Así que por el Lema de König, debe existir una rama infinita[7]. Asuma que dicha rama infinita es de la forma:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_2 \longrightarrow \Delta_2}}{\Gamma_1 \longrightarrow \Delta_1}}{\Gamma_0 \longrightarrow \Delta_0}$$

Si se consideran los conjuntos de fórmulas:

$$\begin{aligned} \Gamma &= \bigcup_{i \geq 0} \Gamma_i \\ \Delta &= \bigcup_{i \geq 0} \Delta_i \end{aligned}$$

Una valuación que asigna 1 a todas las fórmulas en Γ y 0 a todas las fórmulas en Δ refuta el secuente $\Gamma_0 \longrightarrow \Delta_0$.

Ejemplo 6.12 (Derivación infinita y derivación)**.** *Considere un símbolo de función $f/1$ y el secuente $q(z), \forall x.(q(x) \supset q(f(x))) \longrightarrow \forall y.p(y)$. Claramente este secuente no es probable y debe existir una valuación que lo refuta. Si construimos el árbol de la derivación obtenemos lo siguiente:*

$$\dfrac{\dfrac{\dfrac{}{..., q(z) \longrightarrow q(z), p(y_1)} \quad \dfrac{\dfrac{\dfrac{}{..., q(f(z)) \longrightarrow q(f(z)), p(y_1)} \quad \dfrac{\dfrac{\vdots}{q(z), q(f(z)), q(f(f(z))), \forall x.F, F[y_1/x], F[f(f(z))/x] \longrightarrow p(y_1)}}{q(z), q(f(z)), q(f(f(z))), \forall x.F, F[y_1/x]] \longrightarrow p(y_1)}\,\forall_I}{q(z), q(f(z)), \forall x.F, F[y_1/x], F[f(z)/x] \longrightarrow p(y_1)}\,\supset_I}{q(z), q(f(z)), \forall x.F, F[y_1/x] \longrightarrow p(y_1)}\,\forall_I}{q(z), \forall x.F, F[z/x], F[y_1/x] \longrightarrow p(y_1)}\,\supset_I}{q(z), \forall x.F \longrightarrow \forall y.p(y)}\,\forall_D$$

donde F es la fórmula $q(x) \supset q(f(x))$. Así que la valuación v talque $v(p(z)) = 0$ para cualquier z y $v(q(x)) = 1$ para cualquier x conduce a $[\![min([\![q(z)]\!]_v, [\![\forall x.F]\!]_v) > [\![\forall y.p(y)]\!]_v$ refutando el secuente.

Teorema 6.4 (Completitud)**.** *Si el secuente $\Gamma \longrightarrow \Delta$ es válido entonces es derivable.*

[6] Recuerde que estamos considerando una signatura sin símbolos de función y símbolos de constantes.

[7] El Lema de König afirma que un árbol infinito cuyas derivaciones son finitas (*finitely branching*) debe tener una rama infinita.

6.7.3. Consistencia

Recuerde que la regla `cut` nos permite introducir lemas en una prueba tal como sucede en las pruebas matemáticas. Sin embargo, en una prueba automática, esto implica que debemos "adivinar" las fórmulas que debemos introducir en el corte (ver Sección 4.6.3). Para el sistema de la lógica de primer orden, también es posible mostrar que la regla `cut` es admisible y por tanto, en toda derivación, las fórmulas que aparecen en la parte de arriba del secuente son *subfórmulas* de la parte inferior del secuente.

Teorema 6.5 (Eliminación de `cut`)**.** *La regla* `cut` *es admisible para el sistema de la Definición 6.1.*

Demostración. La prueba procede por inducción en la altura de la derivación con una análisis de caso en la última regla aplicada. La idea es mostrar que la regla de corte siempre se puede desplazar a la parte superior del secuente y por tanto, toda prueba utilizando `cut` se puede obtener sin utilizar dicha regla. Solo presentaremos uno de los casos. La prueba completa se encuentra, por ejemplo, en [NvP01]. Considere la siguiente derivación:

$$\dfrac{\dfrac{\dfrac{\vdots}{F[t/x], \forall x.F, \Gamma \longrightarrow C}}{\forall x.F, \Gamma \longrightarrow C}\ \forall_I \qquad \dfrac{\vdots}{C, \Gamma \longrightarrow \Delta}}{\forall x.F, \Gamma \longrightarrow \Delta}\ \texttt{cut}$$

donde la regla $\forall_I$ se utiliza para derivar la fórmula C. Podemos entonces tener un corte de **menor** altura (y por tanto podemos utilizar la hipótesis inductiva) transformando la anterior derivación de la siguiente manera:

$$\dfrac{\dfrac{\dfrac{\vdots}{F[t/x], \forall x.F, \Gamma \longrightarrow C} \qquad \dfrac{\vdots}{C, \Gamma \longrightarrow \Delta}}{F[t/x], \forall x.F, \Gamma \longrightarrow \Delta}\ \texttt{cut}}{\forall x.F, \Gamma \longrightarrow \Delta}\ \forall_I$$

□

Como en una derivación (sin `cut`) se cumple que en todo el secuente solamente hay subformulas del secuente inicial, podemos probar lo siguiente.

Teorema 6.6 (Consistencia)**.** *El secuente* $\longrightarrow \bot$ *no es probable.*

6.8. Indecidibilidad de FOL

Como pudimos notar, el procedimiento de la Sección 6.7.2 no es un algoritmo que nos permita decidir si una fórmula en FOL es válida o no. De hecho, podemos probar, como es estándar en computación, que el problema no es decidible realizando una transformación del problema de validez en FOL al problema de terminación (*halting problem*) en máquina de Turing que sabemos que es indecidible.

> *The following question now arises as a fundamental problem: Is it possible to determine whether or not a given statement pertaining to a field of knowledge is a consequence of the axioms? (David Hilbert)*

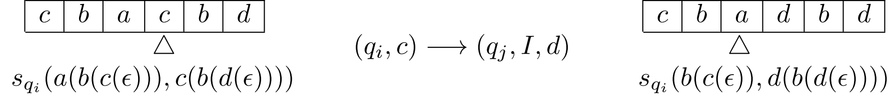


Figura 6.10: Estado de la máquina M y su representación con fórmulas de FOL. Se asume que el alfabeto de la máquina es $\Sigma = \{a, b, c, d\}$. En $s_{q_i}(x, y)$ x se asume que está en orden inverso.

Máquina de Turing Una máquina de Turing es un tupla:

$$\langle Q, \Sigma, \delta, q_i, q_f \rangle$$

donde:

- $Q = \{q_1, ..., q_n\} \cup \{q_i, q_f\}$ es un conjunto de estados.
- q_i es el estado inicial y q_f es el estado final.
- Sigma es un conjunto finito de símbolos (el alfabeto de la máquina)
- $\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma \times \{D, I, C\})$ es una función que indica, dado el estado actual y el símbolo en la cabeza de la máquina, cuál es el nuevo estado de la máquina y cuales son las operaciones sobre la cinta (desplazarse a la (I)zquierda, (D)erecha, o no desplazarse (C)).

Las posibles instrucciones de la máquina son entonces:

- $(q_x, a_i) \longrightarrow (q_y, I, a_j)$: Estando en el estado q_x y con el símbolo a_i en la cabeza de la máquina M, el nuevo estado es q_y, se escribe el símbolo a_j y la cabeza se desplaza a la izquierda.
- $(q_x, a_i) \longrightarrow (q_y, C, a_j)$: igual que el caso anterior pero la cabeza no se desplaza.
- $(q_x, a_i) \longrightarrow (q_y, D, a_j)$: igual que el primer caso pero la cabeza de desplaza a la derecha.

Turing probó que no es posible construir un algoritmo que decida si una máquina M se detiene, es decir, si siguiendo las instrucciones de δ se puede alcanzar el estado q_f.

Una forma de probar la indecidibilidad de FOL es construir una fórmula F a partir de una máquina M tal que M se detiene si y solamente si F es válida. Para esta construcción se requiere:

- Una constante ϵ representando la secuencia vacía.
- Un símbolo de función unario $f_a(\cdot)$ por cada elemento a en el alfabeto de la máquina. Intuitivamente, $f_a(w)$ denota la secuencia $a \cdot w$.
- Por cada estado $q \in Q$, un símbolo de predicado binario $s_q(x, y)$ indicando que se ha alcanzado el estado q con la cinta $x \cdot y$ donde la cabeza está en la primera posición de la secuencia y y que al lado izquierdo de la cabeza se encuentra la secuencia x en orden inverso (ver Figura 6.10). Por ejemplo, $s_{q_i}(\epsilon, w)$ indica que el estado inicial se puede alcanzar donde la cinta del lado derecho es w y al lado izquierdo no hay nada escrito.

Las transiciones de la máquina pueden ser codificadas de la siguiente manera [8]

- $\forall x.\forall y.(s_{q_i}(x, a_i(y)) \supset s_{q_j}(a_j(x), y))$. Esta fórmula representa la lectura del símbolo a_i, la escritura de a_j y el movimiento de la cabeza a la derecha.
- $\forall x.\forall y.(s_{q_i}(b(x), a_i(y)) \supset s_{q_j}(x, b(a_j(y))))$. Esta fórmula representa la lectura del símbolo a_i, la escritura de a_j y el movimiento de la cabeza a la izquierda (ver Figura 6.10).

[8] Algunas fórmulas auxiliares son requeridas para tener en cuenta la secuencia vacía ϵ. Aquí solo mostramos la idea general del encoding.

Con la codificación de las instrucciones y los estados de la máquina, asuma la siguiente fórmula:

$$s_{q_i}(\epsilon, w) \supset \exists x.\exists y.(s_{q_f}(x, y))$$

Intuitivamente esta fórmula dice que si estando en el estado inicial q_i y la entrada w, es posible alcanzar el estado q_f terminando con alguna cinta de la forma $x \cdot y$. Esta fórmula es válida si y solamente si M es capaz de alcanzar el estado final q_f. Como no hay un algoritmo para determinar que M se detiene, entonces no existe un algoritmo para determinar si una fórmula en FOL es válida. [9]

[9] Algunos fragmentos de FOL son de hecho decidibles. Por ejemplo, el fragmento con solamente símbolos unarios (monadic FOL) es decidible. Note que el encoding de máquinas de Turing requiere predicados binarios.

Capítulo 7

Resolución y Programación Lógica

En la Sección 4.5 estudiamos un fragmento de la lógica proposicional que admite procedimientos de decisión eficientes. En este capítulo extendemos dicho fragmento con cuantificadores y exploraremos el método de resolución que sirve de base a la programación lógica (Prolog).

7.1. Formas Normales

Recuerde que una fórmula F está en forma normal prenexa si toma la forma $\nabla x_1.\nabla x_2....\nabla x_n F'$ donde $\nabla \in \{\exists, \forall\}$ y F' está libre de cuantificadores (ver Definición 4.5). Además, F está en forma prenexa-normal-conjuntiva (PFNC) si F' es una conjunción de cláusulas, es decir, una conjunción de disyunciones de literales (ver Definición 4.4). Para el caso de la lógica de primer orden, un literal es una fórmula de la forma $p(\vec{t})$ o $\neg p(\vec{t})$ donde p es un símbolo de predicado.

Definición 7.1 (Cláusulas *ground*). *Decimos que C es una cláusula* ground *si no contiene variables. Dicho de otro modo, C se obtiene a partir de C' sustituyendo las variables de C' por términos* ground *(constantes o funciones aplicadas a constantes).*

En la definición anterior, nos referimos a los "téminos ground". El conjunto de dichos términos se puede obtener de manera inductiva y conforman el universo de Herbrand.

Definición 7.2 (Universo de Herbrand). *Asuma una signatura de primer orden $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{P})$. El universo de Herbrand $\mathcal{H}$ es el conjunto de términos más pequeño que se genera utilizando las siguientes reglas:*

1. *$c \in \mathcal{H}$ para todo $c \in \mathcal{C}$*
2. *$f(t_1, ..., t_n) \in \mathcal{H}$ para toda $f/n \in \mathcal{F}$ y $t_i \in \mathcal{H}$*

Ejemplo 7.1 (Universo de Herbrand). . *Asuma la constante `cero` y la función `suc` (sucesor). Para esta signatura, $\mathcal{H}$ se define como el conjunto:*

$$\{\mathit{cero}, \mathit{suc}(\mathit{cero}), \mathit{suc}(\mathit{suc}(\mathit{cero})), \mathit{suc}(\mathit{suc}(\mathit{suc}(\mathit{cero}))), ...\}$$

Note que si el conjunto de símbolos de funciones $\mathcal{F}$ en una signatura no es vacío, entonces $\mathcal{H}$ es infinito puesto que se pueden formar términos de la forma $f(f(f(.....f(a))))$ como en el ejemplo anterior.

Para utilizar el algoritmo de resolución requeriremos que las clásulas se encuentren en la siguiente forma.

Definición 7.3 (Forma Clausal)**.** *Decimos que F está en forma clausal (*clausal form*), si (1) F es una fórmula* cerrada*, es decir, $fv(F) = \emptyset$; (2) F está en PFNC; y (3) F solo contiene cuantificadores universales.*

Note que la definición anterior requiere la eliminación de los cuantificadores existenciales de una fórmula para ello, utilizaremos el proceso de skolemización.

Definición 7.4 (Skolemización)**.** *Asuma que F se encuentra en forma normal prenexa:*

$$F = \forall x_1.\forall x_n.\exists y.\nabla x_{n+1}...\nabla x_m F'$$

Asuma que f es un (nuevo) símbolo de función de n argumentos que no aparece en F. El cuantificador universal $\exists y$ se puede eliminar de F (preservando equivalencia) si toda ocurrencia de y en F' se reemplaza por $f(x_1, ..., x_n)$.

En la definición anterior, el nuevo símbolo de función f se conoce como *función de Skolem.* Si el número de cuantificadores universales que preceden a $\exists y$ es vacío, simplemente se reemplaza la variable y por un nuevo símbolo de constante (i.e., una función de cero argumentos).

Ejemplo 7.2. *Asuma la siguiente fórmula en PFNC:*

$$\exists x.\forall y.\exists z((p(x,y) \vee \neg q(y,z)) \wedge (\neg p(y,z) \vee q(x,z)))$$

La anterior fórmula se puede convertir a forma clausal eliminando los dos cuantificadores existenciales:

- *Eliminación de $\exists x$:*

$$\forall y.\exists z((p(a,y) \vee \neg q(y,z)) \wedge (\neg p(y,z) \vee q(a,z)))$$

  donde a es un símbolo de constante.

- *Eliminación de $\exists z$:*

$$\forall y.((p(a,y) \vee \neg q(y,f(y))) \wedge (\neg p(y,f(y)) \vee q(a,f(y))))$$

Teorema 7.1 (Skolem)**.** *Para toda fórmula cerrada F existe una fórmula F' en forma clausal tal que $F \equiv F'$.*

Para simplificar la escritura de las fórmulas en forma clausal, utilizaremos la siguiente notación.

Notation 7.1 (Formas Clausales)**.** *Asuma que F está en forma clausal. Para simplificar la notación, se omitirán los cuantificadores universales y por tanto, se asume que toda variable está universalmente cuantificada. Por otro lado, tal como hicimos en la Sección 4.5.2, escribiremos la fórmula F como un conjunto de cláusulas y una cláusula C como un conjunto de literales. Por ejemplo, la siguiente fórmula (que contiene 3 cláusulas):*

$$\forall x.\forall y.\forall z.(\overbrace{(p(x,y) \vee \neg q(x))} \wedge \overbrace{(\neg p(y,z) \vee \neg r(x,y,z))} \wedge \overbrace{(r(x,x,z)})$$

se escribirá como:

$$\{p(x,y), \overline{q(x)}\}, \{\overline{p(y,z)}, \overline{r(x,y,z)}\}, r(x,x,z)$$

7.2. Modelos de Herbrand

Para extender el procedimiento de resolución de la lógica proposicional a la lógica de primer orden, debemos formalizar primero algunos resultados acerca de sustituciones (ver Definición 5.6).

Definición 7.5 (Sustituciones simultáneas)**.** *Utilizaremos la letra θ para denotar una sustitución. En una sustitución simultánea de la forma*

$$\theta = [t_1/x_1, ..., t_n/x_n]$$

las variables $x_1, ..., x_n$ son todas distintas y se reemplaza de manera simultánea *cada una de las variables x_i por el término t_i. Por ejemplo, si $F = p(x, f(y))$ y $\theta = [g(y)/x, a/y]$ entonces $F\theta = p(g(y), f(a))$. Note que x se reemplaza por $g(y)$ y no por $g(a)$.*

A partir de los términos *ground* (el universo de Herbrand) se puede generar el conjunto de predicados *ground* sobre una signatura dada. Este conjunto nos servirá para determinar si una cláusula (o un conjunto de cláusulas) es satisfacible o no, es decir, si tiene un modelo o no.

Definición 7.6 (Base de Herbrand)**.** *Asuma una signatura $\Sigma = (\mathcal{C}, \mathcal{F}, \mathcal{P})$ y el universo de Herbrand $\mathcal{H}_\Sigma$ generado a partir de Σ. La base Herbrand, notación $\mathcal{H}_{B\Sigma}$, es el conjunto de todos los literales construidos a partir de los símbolos de predicados en $\mathcal{P}$ y los términos* ground *en $\mathcal{H}_\Sigma$. Un modelo de Herbrand es un subconjunto de $\mathcal{H}_\Sigma$, es decir, es el conjunto de literales que el modelo interpreta como verdadero (T).*

Ejemplo 7.3 (Modelo de Herbrand)**.** *Asuma la signatura del Ejemplo 7.1 y los predicados $\mathtt{esPar}(\cdot)$ y $\mathtt{esImpar}(\cdot)$. Considere la fórmula $\forall x.(\mathtt{esPar}(x) \supset \mathtt{esImpar}(\mathtt{suc}(x)))$. Note que esta fórmula no es una forma clausual. Sin embargo, se puede reescribir como la cláusula*

$$\{\overline{\mathtt{esPar}(x)}, \mathtt{esImpar}(\mathtt{suc}(x))\}$$

Note que en todo modelo de Herbrand para esta cláusula que incluya el literal `esPar(n)` donde n es el término $\mathtt{suc}^n(\mathtt{cero})$, necesariamente el literal $\mathtt{esImpar}(\mathtt{suc}^{n+1}(\mathtt{cero}))$ debe estar también en el modelo.

Teorema 7.2. *Un conjunto de cláusulas tiene un modelo si y solamente si tiene un modelo de Herbrand.*

Teorema 7.3 (Teorema de Herbrand)**.** *Una fórmula F es insatisfacible si y solamente si un subconjunto finito de cláusulas* ground *de F es insatisfacible.*

El anterior teorema nos permite verificar si una fórmula es insatisfacible encontrando un modelo de Herbrand que hace insatisfacible un subconjunto de sus cláusulas. Recuerde que una fórmula es una tautología si y solamente si su negación es insatisfacible (ver Teorema 3.1).

Ejemplo 7.4. *Sabemos que la fórmula $\forall x.(p(x) \supset p(x))$ es una tautología. Considere su negación representada en forma clausal:*

$$\{p(x)\}, \{\overline{p(x)}\}$$

Una instancia ground *de esta cláusula es*

$$\{p(a)\}, \{\overline{p(a)}\}$$

donde a es una constance. Como este conjunto de cláusulas es insatisfacible, tenemos una prueba que F es una tautología.

Ahora considere la fórmula

$$(\exists x.(p(x) \wedge \forall x.(p(x) \supset q(x))) \supset \exists x(p(x) \wedge q(x))$$

Dicha fórmula también es una tautología. Si transformamos su negación a forma clausal obtenemos:

$$\{\overline{p(x)}, q(x)\}, \{p(a)\}, \{\overline{p(y)}, \overline{q(y)}\}$$

Considere la instancia de dicha cláusula substituyendo tanto x como y por la constante a:

$$\{\overline{p(a)}, q(a)\}, \{p(a)\}, \{\overline{p(a)}, \overline{q(a)}\}$$

Note que dicho conjunto de cláusulas es insatisfacible: debemos asignar `T` *a $p(a)$, y por tanto, $\overline{p(a)}$ es* `F`*. Así que para hacer la primera cláusula verdadera, $q(a)$ debe ser evaluado a* `T`*. Pero dicha asignación hace falsa la tercera cláusula.*

7.3. Resolución

En la sección anterior buscamos instancias (sustituciones) de las cláusulas que contuvieran el literal l y su negación $\bar{l}$ para determinar si el conjunto de cláusulas era insatisfacible. En todos los casos, consideramos cláusulas *ground*. Considere por ejemplo el conjunto de cláusulas:

$$\{p(x,y)\}, \{\overline{p(g(a), f(g(b)))}\}$$

El conjunto de cláusulas es insatisfacible si consideramos la sustitución

$$\theta = [g(a)/x, f(g(b)/y]$$

en este caso, si aplicamos θ a las cláusulas obtenemos:

$$\{p(x,y)\}\theta, \{\overline{p(g(a), f(g(b)))}\}\theta = \{p(g(a), f(g(b)))\}, \{\overline{p(g(a), f(g(b)))}\}$$

que es un conjunto de cláusulas insatisfacible.

Definición 7.7 (Unificador)**.** *Dado un conjunto de literales, un unificador θ es una sustitución que hace todos los literales iguales. Se dice que θ es el unificador más general (MGU) si cualquier otro unificador θ' se puede obtener como la composición de θ con otra sustitución λ, es decir, $\theta' = \theta\lambda$.*

Ejemplo 7.5 (MGU)**.** *Asuma el conjunto de cláusulas*

$$\{p(f(x), g(y))\}, \{\overline{f(f(a)), g(z)}\}$$

Note que las siguientes sustituciones son unificadores:

$$\begin{array}{rcl} \theta_1 & = & [f(a)/x, f(a)/y, f(a)/z] \\ \theta_2 & = & [f(a)/x, f(f(a))/y, f(f(a))/z] \\ \theta_3 & = & [f(a)/x, z/y] \end{array}$$

Como se puede apreciar, es necesario sustituir x por $f(a)$ para poder igualar los dos literales. Sin embargo, solo con sustituir "y" por "z" (o viceversa) los dos literales son iguales. Así que θ_3 es el MGU.

Note que algunos literales no pueden ser unificados. Por ejemplo, $p(f(x))$ no se puede unificar con $p(g(x))$. De la misma manera, $p(x)$ no se puede unificar con $q(x)$.

Definición 7.8 (Algoritmo de Unificación)**.** *Para unificar dos literales de la forma* $p(t_1, ..., t_n)$ *y* $p(t'_1, ..., t'_n)$ *(sin importar si están negados o no), procedemos así:*

1. *Generar el conjunto de ecuaciones (sobre términos)* $\{t_1 = t'_1, ..., t_n = t'_n\}$.
2. *Transformar toda ecuación de la forma* $t = x$ *por* $x = t$.
3. *Eliminar las ecuaciones de la forma* $x = x$.
4. *Dada una ecuación de la forma* $t = t'$ *donde ninguno de los dos términos es una variable.*
   a) *Si* t *y* t' *son símbolos de función diferentes, terminar con mensaje de error (los términos no son unificables).*
   b) *Si la ecuación es de la forma* $f(t_1, ..., t_m) = f(t'_1, ..., t'_m)$*, reemplaza la ecuación* $t = t'$ *por el conjunto de ecuaciones* $\{t_1 = t'_1, ..., t_m = t'_m\}$.
5. *Dada una ecuación de la forma* $x = t$.
   a) *Si* x *ocurre en* t*, terminar con mensaje de error puesto que los términos no se pueden unificar.*
   b) *De lo contrario, reemplazar toda ocurrencia de* x *por* t *en el resto de ecuaciones.*

Ejemplo 7.6 (Cálculo del MGU)**.** *Considere los literales*

$$p(x, f(y), g(x))$$
$$p(f(c), f(g(f(a))), g(w))$$

El cálculo del MGU seguiría los siguientes pasos:

1. $\{x = f(c), f(y) = f(g(f(a))), g(x) = g(w)\}$
2. $\{x = f(c), y = g(f(a)), x = w\}$
3. $\{x = f(c), y = g(f(a)), w = f(c)\}$

Una forma más clara de ver como se unifican dos términos es observando su estructura en un árbol y comparando cuales son los nodos en los cuales los árboles difieren.

Ejemplo 7.7 (Unificación de términos)**.** *Considere los términos:*

$$t_1 = f(x, g(x, y), h(f(b)))$$
$$t_2 = f(w, g(a, f(z)), h(f(z)))$$

En la Figure 7.1 se muestran los pasos para unificarlos:

1. *La primera diferencia entre los dos arboles es la hoja* a *y la hoja* w*. La unificación produce la sustitución* $[a/w]$.
2. *Similar al paso anterior,* x *debe unificar con* a.
3. *Ahora se debe unificar* y *y* $f(z)$.
4. *Finalmente,* z *debe unificarse con el término* b.

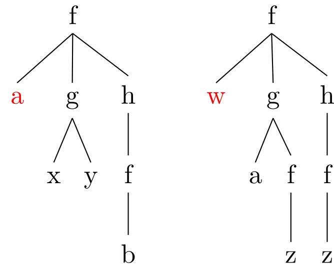
Ya que hemos definido el mecanismo de unificación entre cláusulas, podemos extender el algoritmo de resolución para fórmulas en la lógica de primer orden.

Definición 7.9 (Resolvants)**.** *Asuma dos cláusulas* $C_1 = \{l_1, ..., l_n\}$ *y* $C_2 = \{k_1, ..., k_m\}$ *tal que existe* $C'_1 \subseteq C_1$ *y* $C'_2 \subseteq C_2$ *donde el literal* $l \in C'_1$ *si y solamente si el literal* $\bar{l} \in C'_2$*. Si existe un unificador* θ *para* C'_1 *y* C'_2 *la forma* resuelta *(resolvant) para* C'_1 *y* C'_2 *es la cláusula:*

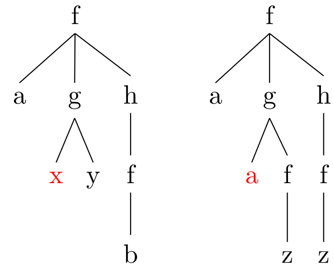
$$C = (C_1 \setminus C'_1\theta) \cup (C_2 \setminus C'_2\theta)$$

En este caso, se dice que C_1 *y* C_2 *están en conflicto (*clash*).*

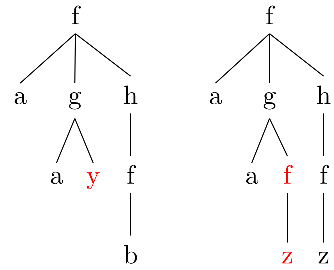
Paso 1



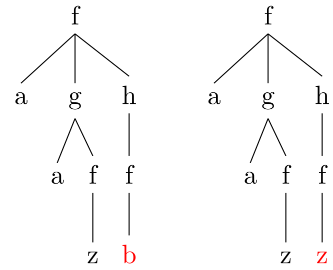
Paso 2: [a/w]



Paso 3: [a/w,a/x]



Paso 4: [a/w,a/x,f(z)/y]



Paso 5: [a/w,a/x,f(z)/y,b/z]

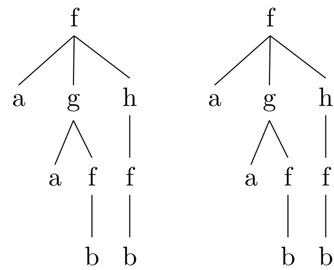


Figura 7.1: MGU para el Ejemplo 7.7

Definición 7.10 (Algoritmo de resolución)**.** *Asuma un conjunto de cláusulas S. Tome $S_0 = S$. Escoja dos cláusulas en conflicto $C_1, C_2 \in S_i$:*

1. *Si la forma resuelta C de C_1 y C_2 es la cláusula vacía ($\Box$), el algoritmo termina y se concluye que S es insatisfacible.*
2. *De lo contrario, $S_{i+1} = S_i \cup \{C\}$. Si $S_{i+1} = S_i$ para cualquier par de cláusulas en conflicto, el algoritmo termina con el resultado que S es satisfacible.*

Ejemplo 7.8 (Resolución)**.** *Considere el siguiente enunciado:*

Si x es el padre de y, entonces x es pariente de y. Se sabe que a es el padre de b. Por lo tanto, a es pariente de b.

El anterior enunciado se puede expresar como las siguientes fórmulas:

$$\begin{array}{lcl} F_1 & = & \forall x, y.(padre(x,y) \supset pariente(x,y)) \\ F_2 & = & padre(a,b) \\ F_3 & = & pariente(a,b) \\ F & = & (F_1 \wedge F_2) \supset F_3 \end{array}$$

La fórmula F intuitivamente debería ser verdadera. Utilizando el algoritmo de resolución podemos probar que efectivamente $\neg F$ es insatisfacible. Consideramos entonces las siguientes cláusulas:

$$\begin{array}{lcl} C_1 & = & \{\overline{padre(x,y)}, pariente(x,y)\} \\ C_2 & = & \{padre(a,b)\} \\ C_3 & = & \{\overline{pariente(a,b)}\} \end{array}$$

C_2 C_1

$\overline{padre(a,b)}$ $padre(a,b)$

$\Box$

Ejemplo 7.9 (Resolución)**.** *Considere el siguiente enunciado:*

Si la ciudad x está conectada con la ciudad y y la ciudad y se conecta con z, entonces x está conectada con z. Se sabe que la ciudad a está conectada con b y que b se conecta con c. Por lo tanto, a está conectada con c.

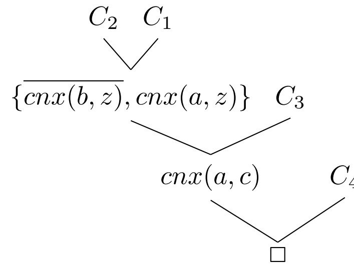
Igual que en el ejemplo anterior, deberíamos poder probar que la conclusión es válida a partir de las premisas. Para ello, modelamos primero el problema como fórmulas en FOL:

$$\begin{array}{lcl} F_1 & = & \forall x, y, z.((cnx(x,y) \wedge cnx(y,z)) \supset cnx(x,z)) \\ F_2 & = & cnx(a,b) \\ F_3 & = & cnx(b,c) \\ F_4 & = & cnx(a,c) \\ F & = & (F_1 \wedge F_2 \wedge F_3) \supset F_4 \end{array}$$

y obtenemos las siguientes cláusulas:

$$
\begin{array}{lcl}
C_1 & = & \{\overline{cnx(x,y)}, \overline{cnx(y,z)}, cnx(x,z)\} \\
C_2 & = & cnx(a,b) \\
C_3 & = & cnx(b,c) \\
C_4 & = & \overline{cnx(a,c)}
\end{array}
$$

y obtenemos la siguiente refutación utilizando el algoritmo de resolución:



7.4. Programación Lógica

Resolución es el mecanismo que utiliza Prolog para concluir si un *query* se puede deducir a partir de un conjunto de hipótesis. En el caso del Ejemplo 7.9, la fórmula F_1 se puede escribir de la siguiente manera (implicación invertida): [1]

```
cnx(X,Z) :- cnx(X,Y), cnx(Y,Z).
```

y se considera como una regla de inferencia:

Para probar que $cnx(x,z)$ se debe probar que $cnx(x,y)$ y que $cnx(y,z)$

Las fórmulas F_2 y F_3 se denominan hechos (*facts*) y se escriben como:

```
cnx(a,b).
cnx(b,c).
```

Finalmente, F_4 se conoce como el *query*, o hecho que queremos probar como verdadero:

```
?: cnx(a,c).
```

En GNU Prolog [2], podemos probar el goal o query del Ejemplo 7.9 [3]:

```
[user].
compiling user for byte code...
cnx(a,b).
cnx(b,c).
cnx(X,Z) :- cnx(X,Y), cnx(Y,Z).
```

después de presionar ctrl+D, se puede expresar el query:

[1]Note el uso de mayúsculas para especificar que X, Y, Z son variables
[2]http://www.gprolog.org
[3]Note que primero se escriben los hechos (*facts*) antes de la definición recursiva de *cnx*.

```
cnx(a,c).
yes
```

Ejemplo 7.10 (Trazas). *Considere el Ejercicio 6.12. Las cláusulas para la definición del predicado* `esTraza` *son las siguientes:*

```
esTraza([], L, L).
esTraza(L, [], L).
esTraza([H1|T1] , L2, [H1|T3]) :- esTraza(T1, L2, T3).
esTraza(L1 , [H2|T2], [H2|T3]) :- esTraza(L1, T2, T3).
```

A continuación el resultado del algunos queries *al sistema:*

```
| ?- esTraza([], [a,b,c],[a,b,c]).
yes
| ?- esTraza([a,b], [c,d],[a,b,c,d]).
yes
| ?-
esTraza([a,b], [c,d],[a,c,d,b]).
yes
| ?- esTraza([a,b], [c,d],[b,a,c,d]).
no
| ?- esTraza([a,b], [c,d],[a,d,b,c]).
no
```

Un uso más interesante del predicado esTraza *es permitir que Prolog, mediante unificación, encuentre una traza válida a partir de los programas.* [4]

esTraza([a,b], [d,e],L). L = [a,b,d,e] ? ;

L = [a,d,b,e] ? ;

L = [a,d,e,b] ? ;

L = [d,a,b,e] ? ;

L = [d,a,e,b] ? ;

L = [d,e,a,b] ? ;

De hecho, es posible instanciar la traza (el tercer argumento) y preguntar por los posibles programas que generan dicha traza:

```
esTraza(P1,P2,[a,b,c,d]).

P1 = []
P2 = [a,b,c,d] ? ;

P1 = [a,b,c,d]
P2 = [] ? ;

P1 = [a]
P2 = [b,c,d] ? ;
```

[4]Para obtener una nueva instanciación de la variable L, se debe digitar ";" en GNU Prolog.

```
P1 = [a,b]
P2 = [c,d] ? ;

P1 = [a,b,c]
P2 = [d] ? ;

P1 = [a,b,d]
P2 = [c] ? ;

...
```

Es posible observar las instancias de los predicados que invoca Prolog utilizando el comando `trace`.

Ejemplo 7.11 (Números Naturales)**.** *A continuación se encuentran las cláusulas para definir algunas operaciones aritméticas.*

- *Factorial de un número.*

  ```
  fact(0,1).
  fact(N,X) :- fact(N2,X2),N is N2 + 1, X is N * X2.
  ```

  Algunos queries:

  ```
  ?- fact(3,X).

  X = 6 ?
  yes.

  ?- fact(X,Y).

  X = 0
  Y = 1 ? ;

  X = 1
  Y = 1 ? ;

  X = 2
  Y = 2 ? ;

  X = 3
  Y = 6 ? ;

  X = 4
  Y = 24 ? ;

  X = 5
  Y = 120 ?
  ```

- *N-ésimo elemento de la seria Fibonacci,*

  ```
  fibonacci(1,1).
  fibonacci(2,1).
  fibonacci(N,X) :- N1 is N-1, N2 is N-2, fibonacci(N1,X1), fibonacci(N2,X2),X is X1 + X2.
  ```

Una consulta:

```
| ?- fibonacci(10,X).
X = 55 ?
yes
```

- *Operaciones sobre listas.*
  - *nelem(N,L,X) : el n-esimo elemento de L es X.*

    ```
    nelem(_,[],_) :- fail. %sin importar N o X, con la lista vacia siempre falla.
    nelem(1,[H|_],H). %El primer elemento siempre es H
    nelem(N,[_|T],X) :- N1 is N-1, nelem(N1,T,X). % Caso inductivo
    ```

    Ejemplos de uso:

    ```
    | ?- nelem(2,[a,b,c],X).
    X = b ?
    yes

    nelem(5,[a,b,c],X).
    no
    ```
  - *reversa(X,Y): la lista Y es la lista X invertida.*

    ```
    reversa([],[]).
    reversa([H],[H]).
    reversa([H|T],L) :- reversa(T,L1), append(L1,[H],L).
    ```

    De hecho, lo que acabamos de definir es el predicado reverse *que se nativo en Prolog.*
  - *palindromo(X): La lista X es un palíndromo.* [5]

    ```
    palindromo(L1) :- reverse(L1, L1).
    ```
  - *elimduplicados(L1,L2): elimina los elementos duplicados continuos de L1, por ejemplo, elimduplicados([a,a,a,b,b,c,a],[a,b,c,a]).*

    ```
    % Predicado auxiliar
    elim(_,[],[]).
    elim(X,[X|T],L) :- elim(X,T,L).
    elim(X,[H|T],[H|L]) :- X\=H, elim(H,T,L).
    elimduplicados([],[]).
    elimduplicados([H|T],[H|T1]) :- elim(H,T,T1).
    ```
  - *duplicar(L1,N,L2) : L2 resulta de duplicar N veces cada uno de los elementos de la lista L1.*

    ```
    %Predicado auxuliar
    gen_N(_,0,[]).
    gen_N(X,N,[X | L]) :- N>0, N1 is N - 1, gen_N(X,N1,L).
    duplicar([],_,[]).
    duplicar([X | T],N,L) :-
       gen_N(X,N,L1), duplicar(T,N,L2) , append(L1,L2,L).
    ```

    Otra alternativa:

    ```
      dup([],_,[],_).
      dup([_|T],0,L,N) :- dup(T,N,L,N).
      dup([H|T],M,[H|L],N) :-
                     M>0,M1 is M-1, dup([H|T],M1,L,N).
       duplicar(L,N,X) :- dup(L,N,X,N).
    ```
- *Árboles Binarios. Un árbol binario se puede representar como el término tree(L,R,X). "empty" corresponde al árbol vacío. Primero construimos un predicado que determine si una*

[5]Intente ejecutar el goal `Palindromo(X)`

variable es un árbol binario y luego podemos realizar ciertas operaciones sobre dichas estructuras.

- *is_Tree(X): determina si X es un árbol binario.*

```
is_Tree(empty). %Caso base
is_Tree(tree(L,R,_)) :- is_Tree(L), is_Tree(R).
```

  Por ejemplo:

```
?- is_Tree(empty).
yes
?- is_Tree(tree(tree(empty,empty,a),tree(empty,empty,b),c)).
        yes
```

- *altura(T,N): determina la altura del árbol T.*

```
altura(empty,0).
altura(tree(L,R,_), N) :- altura(L,NL),altura(R,NR), N is 1+ max(NL,NR).
```

- *generateBT(N,T,Label): genera un árbol totalmente balanceado de altura N donde todos los nodos tienen la etiqueta Label.*

```
generateBT(0,empty,_).
generateBT(N,tree(L,R,Label),Label) :- N1 is N-1,
generateBT(N1,L,Label),generateBT(N1,R,Label).
```

- *isOrdered(T): determina si T es un arbol binario ordenado.*

```
isOrderedAux(empty,_,_).
isOrderedAux(tree(L,R,X),Lb,Ub) :- X>Lb, X<Ub,
          isOrderedAux(L,Lb,X),isOrderedAux(R,X,Ub).

isOrdered(T) :- fd_max_integer(N), isOrderedAux(T,0,N).
```

- *searchT(X,T) : determina si X se encuentra en el árbol T. Se asume que T está ordenado.*

```
searchT(X, tree(_,_,X)).
searchT(X,tree(L,_,Y)) :- X<Y, searchT(X,L).
searchT(X,tree(_,R,Y)) :- X>Y, searchT(X,R).
```

- *inOrder(T,L) : L es el recorrido inorder de un árbol ordenado T:*

```
inOrder(empty, []).
inOrder(tree(L,R,X), Li) :- inOrder(L, LiL),
                          inOrder(R,LiR),append(LiL,[X],LL), append(LL,LiR,Li).
```

  Por ejemplo:

```
        inOrder(tree(
                          tree(
                             tree(empty,empty,1),
                             tree(empty,empty,7),5),
                          tree(
                             tree(empty,empty,15),
                             tree(empty,empty,30),20),  10),L).
        L = [1,5,7,10,15,20,30]
        yes
```

- *Adicionar un elemento a un árbol ordenado.*

```
        addBT(X,empty,tree(empty,empty,X)).
        addBT(X, tree(L,R,Y), tree(L,R2,Y)) :- X>Y, addBT(X,R,R2).
```

```
addBT(X, tree(L,R,Y), tree(L2,R,Y)) :- X<Y, addBT(X,L,L2).
```

Por ejemplo:

```
| ?-addBT(10,empty,T1),addBT(20,T1,T2),addBT(5,T2,T3),addBT(7,T3,T4).

T1 = tree(empty,empty,10)
T2 = tree(empty,tree(empty,empty,20),10)
T3 = tree(tree(empty,empty,5),tree(empty,empty,20),10)
T4 = tree(tree(empty,tree(empty,empty,7),5),tree(empty,empty,20),10) ?
yes.
```

- *Crear un árbol ordenado a partir de una lista de números.*

```
generateOBT([],empty).
generateOBT([H|Tail],T) :- generateOBT(Tail,T1),addBT(H,T1,T).
generate(L,T) :- reverse(L,LR), generateOBT(LR,T).
```

  Por ejemplo:

```
generate([9,5,7,14,6,3,20],T),inOrder(T,L).
L = [3,5,6,7,9,14,20]
T = tree(tree(tree(empty,empty,3),tree(tree(empty,empty,6),empty,7),5),
      tree(empty,tree(empty,empty,20),14),9) ?

yes
```

- *Generar la lista de las hojas de un árbol:*

```
hojas(empty,[]).
hojas(tree(empty,empty,X),[X]).
hojas(tree(L,R,X),Li) :- hojas(L,LiL),hojas(R,LiR),append(LiL,LiR,Li).
```

Ejercicio 7.1. *Asuma una lista de números naturales. Por ejemplo:*

$$[1, 3, 2, 6, 5, 7, 5, 6, 4, 3, 5, 7, 1]$$

Se trata de encontrar la posición del 3 *que está más cercano a un* 7*. En el ejemplo anterior, el resultado es:*

$$\begin{aligned} I3 &= 10 \\ I7 &= 12 \end{aligned}$$

puesto que la distancia entre dicha ocurrencia de 3 *y de* 7 *solo los separa un número (*5 *en este caso). Si la secuencia no contiene un* 3 *el valor de* $I3$ *debe ser* 0*. De manera similar para* $I7$*.*

A continuación se presentan algunos ejemplos de entradas y salidas:

```
[3,7]:  I3 = 1, I7 = 2.
[1,7,2,3,2]: I3=4,I7=2.
[1,3,2,4,1,4,7,2,2,3]: I3=10, I7=7.
[1,2,4,5]: I3 =0 , I7 = 0.
[1,7,4,5]: I3 =0 , I7 = 2.
```

Para resolver el problema debe seguir los pasos que se describen a continuación:

1. *Postcondición. Asuma las siguientes variables:*

$$\begin{array}{lll} l & : & \textit{la lista de enteros} \\ n & : & \textit{la longitud de la lista} \\ i_3 & : & \textit{la posición donde se encuentra el 3} \\ i_7 & : & \textit{la posición donde se encuentra el 7} \end{array}$$

Exprese mediante una fórmula lógica, que relacione las variables anteriores (y otras si lo considera necesario), la postcondición del problema que estamos resolviendo, es decir, cuando el 3 *en* i_3 *es el* 3 *más cercano al* 7 *que se encuentra en* i_7*. Recuerde que si no hay ocurrencias de* 3 *en la secuencia,* i_3 *toma el valor de cero y de manera similar para* i_7*. Explique la fórmula que propuso. Note que aquí está definiendo* cuando i_3 *e* i_7 *son una solución al problema, no está definiendo* como *va a resolver el problema.*

2. *Caso(s) Base. Ahora vamos a definir un predicado en Prolog que nos permita calcular* i_3 *e* i_7*. Primero defina el (los) caso(s) base y explíquelos.*

3. *Casos Inductivos. Defina cuáles serían los casos inductivos y explíquelos brevemente.*

4. *Con base en las observaciones anteriores escriba completamente el programa en Prolog. Si lo considera necesario, puede declarar predicados auxiliares.*

Bibliografía

[Abr10] J-R. Abrial. *Modeling in Event-B: System and Software Engineerin.* Cambridge University Press, 2010.

[BA01] M. Ben-Ari. *Mathematical Logic for Computer Science. 2nd. Edition.* Springer, 2001.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. CoqÁrt: The Calculus of Inductive Constructions.* Springer, 2004.

[Ber10] Yves Bertot. Coq in a hurry, 2010.

[Gal03] J. Gallier. *Logic For Computer Science: Foundations of Automatic Theorem Proving.* 2003.

[Gal11] Jean Gallier. *Discrete Mathematics.* Springer, 2011.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* Cambridge University Press, 2004.

[MN12] D. Miller and G. Nadathur. *Programming with Higher-Order Logic.* Cambridge University Press, 2012.

[NvP01] Sara Negri and Jan von Plato. *Structural Proof Theory.* Cambridge University Press, 2001.

[Ros04] K. Rosen. *Matemática Discreta y sus Aplicaciones. 5ta Edición.* McGraw Hill, 2004.

Índice alfabético

Álgebras Booleanas, 22

Admisibilidad (de una regla), 42, 90
Algoritmo de Resolución, 99
Aritmética de Peano, 53

Base de Herbrand, 95

Cálculo de Secuentes, 27
  Completitud, 41, 88
  Consistencia, 42, 90
  Correctitud, 39, 87
  Lógica de Primer Orden, 57
  Lógica Proposicional, 28
Cláusulas, 93
Cláusulas de Horn, 36
Completitud, 41
Conectivos Funcionalmente Completos, 20
Conjunto bien ordenado, 7
Conjunto Parcialmente Ordenado, 7
Consecuencia Lógica, 19
Consistencia, 42
Coq, 31
  ;, 31
  Árboles Binarios, 82
  absurd, 31
  apply, 31
  assert, 31
  assumption, 31
  classic, 35
  contradiction, 31
  destruct, 31
  discriminate, 73
  Ensemble, 62
  Especificación de Sistemas, 61
  Estructura de Datos, 78
  exact, 35
  exists, 60
  Fixpoint, 74
  forall, 60
  Hint Resolve, 76
  induction, 73
  Inductive, 74
  inductive, 73
  intro, 31
  intros, 31
  inversion, 73
  Listas, 78
  rewrite, 76
  right, 31
  simple induction, 73
  split, 31
  Tipos Inductivos, 73
  unfold, 31
Correctitud, 39
Cut elimination, 42, 90

Especificación, 61
  de Estructura de Datos, 78
  de Sistemas, 61
Especificación vs Implementación, 80
Esquemas de Fórmulas, 53
Estructuras y Modelos, 50

Forma Clausal, 94
Forma Normal Conjuntiva (FNC), 38
Formal Normal Prenexa, 59, 94

Indecidibilidad de FOL, 90
Inducción, 7, 52
  Función definida inductivamente, 10, 74
  Principio de, 7
  Pruebas Inductivas, 10
  Tipos Inductivos, 73

König (Lema), 89

Lógica de Primer Orden, 47
  Alfabeto, 48
  Cálculo de Secuentes, 57
  Completitud, 88
  Consecuencia Lógica, 52
  Consistencia, 90
  Correctitud, 86
  Estructuras, 50
  Formas Normales, 93

Indecidibilidad, 90
Modelos, 51
Refutación, 89
Resolución, 93, 96
Satisfacibilidad, 52
Semántica, 50
Signatura, 47
Sintaxis, 48
Sistema Intuicionista, 58
Términos, 48
Tautología, 52
Lógica Intuicionista, 32
Cálculo de Secuentes, 32
Lógica Proposicional, 9
Alfabeto, 9
Cálculo de Secuentes, 28
Equivalencias, 20
Fórmulas bien Formadas, 10
Intuicionista, 32
Resolución, 36
Semántica, 17
Sintaxis, 9

Máquina de Turing, 91
MGU, 96
Modelo, 17
Modelo de una Fórmula, 18
Modelo de Herbrand, 95
Modelos de Herbrand, 94

Ordenamientos, 7
Conjuntos Bien Ordenados, 7
Parciales, 7
Parejas, 14

Prenexa (Forma Normal), 59
Programación Lógica, 100
Prolog, 37, 100
Árboles Binarios, 103
Listas, 103

Reflexividad, 7
Regla de corte, 42, 90
Relación de Equivalencia, 7
Renombramiento de Variables, 50
Resolución, 36, 96
Algoritmo, 99
Cláusulas de Horn, 36
Forma Clausal, 94
Lógica Proposicional, 36
MGU, 96
Sistema de Pruebas, 36
Unificador, 96
Unificador más general, 96
Resolvants, 97

Satisfacibilidad, 18
Sentencias, 51
Signatura, 47
Simetría, 7
Sistemas à la Hilbert, 27
Skolemización, 94
SLD Resolution, 37
Sustitución, 49
Sustituciones, 95

Términos, 48
Tautología, 18, 52
Teorema de Herbrand, 95
Transitividad, 7

Unificador, 96
Unificador más general, 96
Universo de Herbrand, 93

Validez, 18
Valuación, 17
Variables Libres, 49
Variables Ligadas, 49
Variables Renombramiento, 50
Verificación, 61

Well-founded set, 7