TRABAJO FIN DE MÁSTER

Máster Universitario en Matemáticas

Facultad de Matemáticas

Departamento de Ciencias de la Computación e Inteligencia Artificial

---

FORMALIZACIÓN DE CÁLCULOS LÓGICOS EN ISABELLE/HOL

Realizado por:
María Dolores Mateo Ceballos

Supervisado por:
Dª. María José Hidalgo Doblado

Índice general

Abstract

Natural deduction is a sound and complete proof procedure for propositional logic, that is, it only proves valid formulas and it proves every valid formula. In this work we establish the theory of propositional logic, and we prove the soundness and completeness theorems for natural deduction in propositional logic, following the Melving Fitting's book *First-Order Logic and Automated Theorem Proving*.

We also present a formalization of this theory in Isabelle/HOL. The formalization covers the sintax and semantic of propositional logic, the model existence theorem, and a natural deduction proof calculus together with a proof of soundness and completeness. For this purpose, we introduce Isabelle/HOL system in this work and the main concepts that we use in the above formalization.

Capítulo 1

Introducción

La lógica proposicional estudia las proposiciones, llamadas fórmulas lógicas, formadas a partir de otras proposiciones, que pueden ser verdaderas o falsas, y las conectivas lógicas.

La lógica nace hace más de dos milenios en la Antigua Grecia con Aristóteles y su investigación acerca de los principios del razonamiento válido o correcto, la cual se recoge principalmente en *Órganon*. Sin embargo, no es hasta finales del siglo XVII cuando la lógica matemática comienza a desarrollarse, gracias principalmente a la aportación del filósofo y matemático Gottfried Leibniz (1646-1716), aunque parte de su trabajo no es conocido por la comunidad lógica hasta más tarde. Es por ello que George Boole (1815-1864) y Augustus De Morgan (1806-1871) recreean trabajos de Leibniz de manera independiente. Sin embargo, también hacen aportaciones novedosas.

El primero es el creador de la conocida Algebra de Boole, recogida inicialmente en *Análisis Matemático de la Lógica*, publicado en 1847, y extendido en 1854 en *Investigación sobre las Leyes del Pensamiento*.

Augustus De Morgan es conocido, entre otros, por formular las llamadas Leyes de De Morgan. Su obra principal en el campo de la lógica es *Lógica formal*, publicado en 1847.

Entre el final del siglo XIX y el comienzo del siglo XX se sientan las bases de la lógica matemática moderna, en particular de la lógica de predicados de Gottlob Frege. Aún así, la lógica proposicional se sigue desarrollando. Por ejemplo, la deducción natural fue introducida por Gerhard Gentzen en su trabajo *Investigaciones sobre la inferencia lógica*, publicado en 1934.

Uno de los objetivos de la lógica es determinar qué fórmulas son verdaderas siempre, sin importar si las proposiciones a partir de las cuales están formadas son verdaderas o falsas. Estas fórmulas son denominadas válidas.

También es interesante saber qué fórmulas son consecuencia de otras. Se dice que una fórmula F es consecuencia lógica de un conjunto de fórmulas S si F es cierta cuando todos los miembros de S lo son, sin importar, al igual que en el caso

de la validez lógica, si las proposiciones que forman dichas fórmulas son verdaderas o falsas.

Sin embargo, no sólo basta con la definición de validez y consecuencia lógica. Es de gran interés definir algoritmos que establezcan qué fórmulas lógicas son válidas o consecuencia lógica de ciertos conjuntos de fórmulas. Estos algoritmos se denominan procedimientos de prueba y sólo manipulan las fórmulas como símbolos o cadenas, sin hacer uso de su significado o del de las proposiciones que las forman. Un procedimiento de prueba debe verificar:

1. sólo prueba fórmulas válidas (es adecuado);

2. prueba todas las fórmulas válidas (es completo).

El objetivo de la primera parte de este trabajo es establecer la adecuación y completitud de un procedimiento de prueba para la lógica proposicional: la deducción natural. Esto lo haremos en el Capítulo 3, definiendo en primer lugar la sintaxis y semántica de la lógica proposicional; en las siguientes secciones nos centraremos en probar el Teorema de existencia de modelos y todos los teoremas necesarios para ello, estableciendo así un argumento de completitud "estándar". En la sección 3.7 expondremos las reglas de deducción natural y probaremos su adecuación y completitud, usando dicho argumento de completitud.

Esta primera parte teórica está basada en el libro *First-Order Logic and Automated Theorem Proving* [4] de Melvin Fitting, aunque ha sido complementado con el libro *Logic in Computer Science: Modelling and reasoning about systems* [5] de Huth y Ryan.

El avance de las ciencias de la computación ha hecho posible la construcción de sistemas en los que representar el conocimiento matemático, de forma rigurosa y reutilizable. El razonamiento automático nace en el año 1954 cuando Martin Davis programó el algoritmo de Presburger. Posteriormente, no es hasta el año 1965 cuando Robinson presenta el principio de resolución, lo que constituye un avance importante en esta línea. Ya en la década de los 70, empiezan a desarrollarse demostradores especializados. Entre ellos cabe destacar Nuprl, Coq, PVS, ACL2 e Isabelle.

Los sistemas de razonamiento automático se usan en distintas áreas de conocimiento, como lógica, ciencias de la computación, matemáticas e ingeniería. Se usan tanto para la formalización de teorías como para la verificación de software y hardware.

El objetivo de la segunda parte de este trabajo es la verificación formal en Isabelle/HOL de la adecuación y completitud de la deducción natural en lógica proposicional, lo cual llevaremos a cabo en el Capítulo 4.

Isabelle es un demostrador interactivo de teoremas desarrollado por Lawrence Paulson de la Universidad de Cambridge y Tobias Nipkow de la Universidad Técnica de Múnich, implementado en el lenguaje de programación funcional ML.

Isabelle se distribuyó por primera vez en 1986, aunque desde entonces sigue en permanente desarrollo. Isabelle/HOL es una especialización de Isabelle para HOL. En el Capítulo 2 se presenta una introducción a Isabelle/HOL, con los principales conceptos que utilizaremos en el desarrollo del Capítulo 4.

Existe un trabajo relacionado con el nuestro en Coq: *Propositional Calculus in Coq* [10], desarrollado por Floris van Doorn, y otro en Isabelle/HOL: *First-Order Logic According to Fitting* [3], desarrollado por Stefan Berghofer, en el que se inspira la verificación formal de nuestro trabajo.

El trabajo de Berghofer consiste en la formalización de la adecuación y completitud de la deducción natural para la lógica de primer orden, siguiendo el libro de Fitting. Nuestro trabajo ha sido desarrollado para la lógica proposicional, lo que marca una diferencia que empieza con la formalización de las fórmulas en Isabelle, que no poseen cuantificadores en la lógica proposicional.

Las pruebas formales en Isabelle se han realizado de forma interactiva, y en un futuro podría continuarse este trabajo realizando las pruebas en el lenguaje de prueba estructurado Isar. Al final del Capítulo 2 dedicaremos una sección a este lenguaje.

Asimismo, como la prueba de la completitud de la deducción natural para la lógica proposicional hace uso de un argumento de completitud "estándar", podría ser usado para probar la completitud de otros procedimientos de prueba, como los tableros semánticos, lo cual sería relativamente fácil. Además, podríamos extender este trabajo para la lógica de primer orden sin igualdad.

En este trabajo se han estudiado, por tanto, los teoremas de adecuación y completitud de métodos de razonamiento para la lógica proposicional y se ha profundizado en el conocimiento y uso de sistemas de verificación y razonamiento automático, logrando el desarrollo parcial de una teoría matemática y la verificación formal de los teoremas de dicha teoría en Isabelle/HOL.

Capítulo 2

Introducción a Isabelle/HOL

Isabelle ([7], [8]) es un demostrador interactivo de teoremas desarrollado por Lawrence Paulson de la Universidad de Cambridge y Tobias Nipkow de la Universidad Técnica de Múnich, implementado en el lenguaje de programación funcional ML.

Isabelle soporta el razonamiento formal en varias lógicas objeto: lógicas de primer orden constructiva y clásica (FOL), lógicas de orden superior (HOL), teoría de conjuntos de Zermelo–Fraenkel, teoría de tipos de Martin Löf, lógicas modales, etc.

Isabelle/HOL es una especialización de Isabelle para HOL, abreviatura de Higher-Order Logic.

En Archive of Formal Proofs [1] se encuentran disponibles una variedad de teorías que corresponden a ejemplos y desarrollos científicos que han sido formalmente verificados en Isabelle y, en particular, en Isabelle/HOL.

Por otro lado, Isabelle/Isar [11] es una extensión de Isabelle que permite pruebas estructuradas en el lenguaje Isar (Intelligible semi-automated reasoning). Isar ofrece un entorno versátil para pruebas estructuradas de teoremas, y proporciona documentos de prueba fácilmente legibles por los humanos.

En este capítulo introduciremos los elementos básicos de Isabelle que usaremos más adelante en el capítulo 4. En concreto, introduciremos las pruebas de teoremas, tanto en el lenguaje de prueba estructurado Isar como de forma interactiva, aunque en el capítulo 4 sólo realizaremos las pruebas de forma interactiva. Más información se encuentra disponible en [9]. Podemos descargar e instalar Isabelle desde su página principal [2].

2.1. Teorías

Para trabajar con Isabelle debemos definir teorías. Una teoría es un conjunto de tipos, funciones y teoremas. El formato general en Isabelle/HOL de una teoría `T` es

```
theory T
imports T1 ... Tn
begin
   declaraciones, definiciones y pruebas
end
```

donde `T1 ... Tn` son los nombres de teorías existentes en las que se basa `T`, y *declaraciones, definiciones y pruebas* representa los conceptos nuevos añadidos y las pruebas sobre ellos. Cada teoría `T` debe residir en un archivo llamado `T.thy`.

HOL contiene una teoría `Main`, la unión de todas las teorías básicas predefinidas como aritmética, listas, conjuntos, etc.

2.2. Tipos, términos y fórmulas

El sistema de tipos de HOL se asemeja al de los lenguajes funcionales como ML o Haskell. Estos son:

- los tipos básicos, como `bool`, el tipo de los valores de verdad, o `nat`, el tipo de los naturales;
- los constructores de tipo, como `list`, el tipo de las listas, o `set`, el tipo de los conjuntos, que se construyen a partir de otros tipos;
- el tipo de las funciones, denotado por $\Rightarrow$, que sólo representa las funciones totales;
- las variables de tipo, denotadas por `'a`, `'b`, etc.

Los **términos** se forman aplicando funciones a argumentos. Si `f` es una función de tipo $\tau_1 \Rightarrow \tau_2$ y `t` es un término de tipo τ_1 entonces `f t` es un término de tipo τ_2. HOL también soporta funciones infijas como `+` y algunas construcciones básicas de la programación funcional, como expresiones condicionales:

- `if` b `then` t_1 `else` t_2, donde b es del tipo `bool` y t_1 y t_2 son del mismo tipo.
- `let` $x = t$ `in` u es equivalente a u donde todas las apariciones libres de u han sido reemplazadas por t. Las asignaciones múltiples se separan por puntos y comas: `let` $x_1 = t_1$`;` `...` `;` $x_n = t_n$ `in` u.
- `case` e `of` $c_1 \Rightarrow e_1$ `|` `...` `|` $c_n \Rightarrow e_n$ se evalúa como e_i si e es de la forma c_i.

Los términos también pueden contener funciones λ. Así, λx_1 `...` x_n`.` `t` es la función que toma x_1`,` `...`,x_n como argumentos y devuelve t.

Aunque Isabelle infiere automáticamente el tipo de cada variables en un término, a veces es necesario añadir restricciones de tipo. La sintaxis es `(`t`::`τ`)`.

Las **fórmulas** son términos del tipo `bool`. Existen constantes básicas `True` y `False`; las conectivas lógicas usuales, en orden decreciente de prioridad, $\neg$, $\wedge$, $\vee$ y $\longrightarrow$, asociadas por la derecha; la igualdad, que es una función infija del tipo `'a ⇒ 'a ⇒ bool`, y funciona como un sí-y-sólo-sí si `'a` es el tipo `bool`; y los cuantificadores, que se escriben como `∀x. P` y `∃x. P`.

2.3. Variables

Isabelle distingue tres tipos de variables: libres, ligadas y desconocidas, aunque este último tipo de variable en realidad es también una variable libre. Las variables desconocidas se representan con un `?` como primer carácter, e Isabelle puede instanciarlas por otro término en cualquier momento de una prueba. Por ejemplo, el teorema $x = x$ se representa en Isabelle como `?x = ?x`, que significa que Isabelle puede instanciarlo arbitrariamente.

2.4. Construcción de tipos

Para construir tipos en Isabelle se usa el comando `datatype`. El formato general de dicho comando es el siguiente:

`datatype` (α_1`,...,`α_n) `t =` C_1 τ_{11} ... τ_{1k_1} | ... | C_m τ_{m1} ... τ_{mk_m}

donde `t` es el nombre del tipo que vamos a definir; los α_i, $i = 1, ..., n$, son variables de tipo distintas, de las que depende `t`; C_i es el nombre de cada constructor, $i = 1, ..., m$; y τ_{ij} son los tipos a los que se aplica cada constructor.

Se pueden definir tipos de datos recursivos usando el tipo de dato `t` dentro de su definición, como uno de los tipos τ_{ij}.

En el siguiente ejemplo definiremos el tipo de dato recursivo `Lista`. Una lista de elementos de tipo `a` es la lista `Vacia` o se obtiene añadiendo, con el constructor `Cons`, un elemento de tipo `a` a una lista de elementos de tipo `a`.

```
datatype 'a Lista = Vacia | Cons 'a "'a Lista"
```

Cuando se define un tipo de dato, Isabelle genera automáticamente teoremas sobre dicho tipo. En particular se genera automáticamente un principio de inducción sobre cualquier tipo de dato recursivo. En el caso del ejemplo, el principio generado es el siguiente:

```
thm Lista.induct

?P Vacia ⟹
(⋀a Lista. ?P Lista ⟹ ?P (Lista.Cons a Lista)) ⟹ ?P ?Lista
```

2.5. Definición de funciones

En esta sección introduciremos las definiciones no recursivas y las recursivas.

Las definiciones no recursivas pueden hacerse con el comando `definition`, cuyo formato general es

`definition f ::` τ_1 `⇒ ... ⇒` τ_n `where "`$definicion$`"`

donde `f` es el nombre de la función y τ_1 ..., τ_n es el tipo de sus argumentos. La definición debe ser escrita entre comillas y puede incluir expresiones condicionales como las introducidas en la sección 2.2.

Veamos a modo de ejemplo la definición de la disyunción exclusiva.

```
definition xor :: "bool ⇒ bool ⇒ bool" where
  "xor A B ≡ (A ∧ ¬B) ∨ (¬A ∧ B)"
```

Cuando usamos el comando `definition` Isabelle genera automáticamente un teorema `f_def`, donde `f` es el nombre de la función definida, con dicha definición. En el ejemplo, Isabelle genera

```
thm xor_def
```

```
xor ?A ?B ≡ ?A ∧ ¬ ?B ∨ ¬ ?A ∧ ?B
```

Para definir funciones recursivas totales podemos usar el comando `fun` o el comando `function` [6]. Hay que probar que dichas funciones terminan, probando que existe una medida, que depende de los argumentos de la función, que decrece en cada llamada recursiva. En muchos casos no es necesario proporcionar una función de medida, pues Isabelle puede probar que terminan de forma automática. Isabelle trata de hacer esto mediante el uso del comando `fun`. Su formato general es el siguiente:

`fun f ::` τ_1 `⇒ ... ⇒` τ_n `where`
"$definicion\ 1$" |
⋮ |
"$definicion\ m$"

donde `f` es el nombre de la función y τ_1 ..., τ_n es el tipo de sus argumentos.

Podemos definir, por ejemplo, la suma de los n primeros números naturales, mediante la siguiente función `suma`.

```
fun suma :: "nat ⇒ nat" where
  "suma 0       = 0"
| "suma (Suc m) = (Suc m) + suma m"
```

Al definir funciones recursivas Isabelle genera de forma automática un principio de inducción. En el caso de la función `suma`, es el siguiente.

```
thm suma.induct

?P 0 ⟹ (⋀m. ?P m ⟹ ?P (Suc m)) ⟹ ?P ?a0.0
```

En este caso, el principio de inducción proporcionado por Isabelle para la función suma es, de hecho, el principio de inducción en los números naturales, pero, en general, el principio de inducción generado por Isabelle es específico para cada función.

Definamos a continuación como ejemplo la función de Ackermann.

```
fun ack :: "nat ⇒ nat ⇒ nat" where
  "ack 0       n       = n+1"
| "ack (Suc m) 0       = ack m 1"
| "ack (Suc m) (Suc n) = ack m (ack (Suc m) n)"
```

Isabelle genera el siguiente principio de inducción:

```
thm ack.induct

(⋀n. ?P 0 n) ⟹
(⋀m. ?P m 1 ⟹ ?P (Suc m) 0) ⟹
(⋀m n. ?P (Suc m) n ⟹
        ?P m (ack (Suc m) n) ⟹ ?P (Suc m) (Suc n)) ⟹
?P ?a0.0 ?a1.0
```

Para probar que una función recursiva está bien definida o que termina, se usa el comando `function`, que hace visibles dichos objetivos a probar. Veamos un ejemplo:

```
function mezclar :: "'a list ⇒ 'a list ⇒ 'a list" where
  "mezclar [] ys = ys"
| "mezclar xs [] = xs"
| "mezclar(x#xs) (y#ys) = x # (mezclar (y#ys) xs)"
  by pat_completeness auto

termination
  by (relation "measure (λ(xs, ys). length xs + length ys)") auto
```

Esta función combina dos listas, añadiendo sucesivamente en cada llamada recursiva un elemento de cada lista. La combinación de los métodos `pat_completeness` y `auto` permite probar la completitud y la compatibilidad de los patrones. El comando `termination` indica el comienzo de la prueba de terminación de la función.

2.6. Definiciones inductivas

Se pueden hacer definiciones inductivas en Isabelle, especificando las reglas de introducción correspondientes. El comando `inductive_set` genera el menor conjunto cerrado bajo las reglas de introducción definidas, mientras que el comando `inductive` genera la menor relación cerrada bajo las reglas de introducción.

Podemos definir inductivamente, por ejemplo, el conjunto de los números pares como el menor conjunto que contiene al 0 y es cerrado por la operación (+2).

```
inductive_set par :: "nat set" where
  cero: "0 ∈ par"
| paso: "n ∈ par ⟹ (Suc (Suc n)) ∈ par"
```

También se puede definir la relación "ser par":

```
inductive par :: "nat ⇒ bool" where
  cero: "par 0"
| paso: "par n ⟹ par (Suc (Suc n))"
```

Las definiciones inductivas generan varios teoremas, entre otros, las reglas de introducción, las de simplificación y un principio de inducción. Veámoslas para la primera defínción de `par`:.

- `par.cero` `0 ∈ par`.
- `par.paso` `?n ∈ par ⟹ Suc (Suc ?n) ∈ par`.
- `par.simps`
  `(?a ∈ par) = (?a = 0 ∨ (∃n. ?a = Suc (Suc n) ∧ n ∈ par))`
- `par.induct`
  ```
  ?x ∈ par ⟹
  ?P 0 ⟹
  (⋀n. n ∈ par ⟹ ?P n ⟹ ?P (Suc (Suc n))) ⟹
  ?P ?x
  ```

2.7. Métodos de prueba

En esta sección introduciremos las demostraciones interactivas usando `apply` y las demostraciones automáticas.

Para enunciar un teorema es posible usar dos comandos: `theorem` y `lemma`. Ambos comandos funcionan igual; la única diferencia es la importancia que le damos a cada uno.

Tras enunciar un teorema se puede usar el comando `by` seguido de las reglas o métodos usados en la demostración, o el comando `apply`, para indicar a Isabelle qué regla o método debe usar y acceder a los nuevos objetivos a probar que genera cada una, acabando la demostración con `done`.

2.7.1. Simplificación

La simplificación es un método para la demostración de teoremas, que consiste en la reescritura de términos o fórmulas, no necesariamente simplificándolos. Para aplicar simplificación en el primer objetivo de una prueba se escribe `apply simp`. Si queremos aplicar simplificación en todos los objetivos escribiremos `apply simp_all`.

Como vimos en secciones anteriores, al definir tipos de datos o ciertas funciones, Isabelle genera automáticamente reglas de simplificación, aunque también se pueden declarar teoremas como reglas de simplificación manualmente. Para ello se usa el atributo `[simp]`. También se puede usar

`declare` $nombre - teorema$`[simp]`
`declare` $nombre - teorema$`[simp del]`

que declara un teorema como regla de simplificación o lo elimina, respectivamente.

Por otra parte, es posible usar sólo algunas reglas de simplificación o simplificar usando teoremas que no estén contenidos en dichas reglas:

- `apply (simp add:` $lista - teoremas$`)` aplica simplificación añadiendo los teoremas $lista - teoremas$;
- `apply (simp del:` $lista - teoremas$`)` aplica simplificación eliminando de dichas reglas los teoremas $lista - teoremas$;
- `apply (simp only:` $lista - teoremas$`)` aplica simplificación sólo con los teoremas $lista - teoremas$.

También las hipótesis son parte del proceso de simplificación. Podemos modificar esto, de la siguiente forma:

- `apply (simp (no_asm))` aplica simplificación ignorando completamente las hipótesis;
- `apply (simp(no_asm_simp))` aplica simplificación usando las hipótesis sin simplificarlas;
- `apply (simp(no_asm_use))` aplica simplificación tanto en las hipótesis como en los objetivos sin usar las hipótesis para simplificar estos.

Es posible asimismo aplicar las hipótesis de un teorema sin usar reglas de simplificación, de la siguiente forma:

```
apply assumption
```

2.7.2. Aritmética

Se entienden por fórmulas aritméticas lineales aquellas en las que intervienen variables, números, $+$, $-$, $=$, $<$, $\leq$, mín, máx, las conectivas lógicas usuales, pero no la multiplicación (aunque sí está permitida la multiplicación de números). Dichas fórmulas pueden ser probadas usando el método `arith`.

```
lemma "m ≠ (n::nat) ⟹ m < n ∨ n < m"
  by arith
```

2.7.3. Razonamiento automático

Una de las características más importantes de Isabelle es su capacidad para demostrar teoremas automáticamente. Existen muchos métodos para aplicar el razonamiento automático de Isabelle, aunque sólo veremos algunos de ellos brevemente:

- `auto`. Esencialmente `auto` intenta simplificar los objetivos, y es capaz de probar objetivos lógicos o de teoría de conjuntos simples, como los siguientes:

  ```
  lemma "∀x. ∃y. x = y"
  by auto

  lemma "A ⊆ B ∩ C ⟹ A ⊆ B ∪ C"
  by auto
  ```

- `fastforce`. Logra probar objetivos más complejos que `auto`, como algunos objetivos lógicos con cuantificadores. El siguiente es un ejemplo de lo que `fastforce` puede hacer:

  ```
  lemma "∀xs ∈ A. ∃ys. xs = ys @ ys ⟹
         us ∈ A ⟹ ∃n. length us = n+n"
  by fastforce
  ```

  Este lema está fuera del alcance de `auto` debido a los cuantificadores.

- `blast`. Este método puede resolver estructuras lógicas más complejas que los anteriores, así como objetivos sobre teoría de conjuntos y relaciones. Sin embargo, es muy débil en el razonamiento de la igualdad. En el siguiente ejemplo ilustramos en uso de `blast`.

  ```
  lemma "∀x y. T x y ∨ T y x ⟹
         ∀x y. A x y ∧ A y x ⟶ x = y ⟹
         ∀x y. T x y ⟶ A x y ⟹
         ∀x y. A x y ⟶ T x y"
  by blast
  ```

Sledgehammer

El comando Sledgehammer llama a una serie de demostradores de teoremas automáticos externos (ATPs) que se ejecutan durante treinta segundos buscando una prueba. Algunos de estos ATPs forman parte de la instalación de Isabelle, mientras que otros son consultados a través de Internet. Si la prueba tiene éxito, se genera un comando que puede ser insertado en la prueba. Por ejemplo,

```
lemma "xs @ ys = ys @ xs ⟹ length xs = length ys ⟹ xs = ys"
```

no puede ser probado por ninguno de los métodos de prueba anteriores, pero Sledgehammer encuentra la siguiente solución:

```
by (metis append_eq_conv_conj)
```

Sin embargo, las demostraciones usando el comando Sledgehammer no siempre tienen éxito. Sledgehammer trata de encontrar una prueba usando los lemas que hayan sido probados previamente. En el trabajo de formalización se va construyendo la teoría y estableciendo los resultados necesarios para que, en determinados momentos, el propio sistema pueda encontrar la prueba.

2.7.4. Los métodos de reglas

Denotemos por `R` a una regla arbitraria de la siguiente forma

$$\frac{P_1 \dots P_n}{Q}$$

Introducimos a continuación los métodos básicos de reglas.

- El método `rule R` unifica Q con el subobjetivo actual, reemplazándolo por n nuevos subobjetivos: instancias de $P_1, \dots, P_n$. Este es el razonamiento hacia atrás y es apropiado para las reglas de introducción.

- El método `erule R` unifica Q con el subobjetivo actual y simultáneamente unficia P_1 con alguna hipótesis. El subobjetivo es reemplazado por los $n-1$ nuevos subobjetivos: instancias de $P_2, \dots, P_n$, con dicha hipótesis eliminada. Es apropiado para las reglas de eliminación.

- El método `drule R` unifica P_1 con alguna hipótesis, que luego es eliminada. El subobjetivo es reemplazado por los $n-1$ nuevos subobjetivos: instancias de $P_2, \dots, P_n$; un n-ésimo subobjetivo se genera como el original pero con una instancia de Q como hipótesis adicional. Es apropiado para las reglas de destrucción.

- El método `frule R` es como `drule R` excepto que la hipótesis que se unifica con P_1 no es eliminada.

Otros métodos aplican una regla mientras instancian algunas de sus variables. La forma típica es

`rule_tac` $v_1 = t_1$ `and ... and` $v_k = t_k$ `in R`

Este método es análogo para `erule_tac`, `drule_tac` y `frule_tac`.

2.7.5. Inducción estructural

En Isabelle se puede hacer recursión estructural sobre cualquier tipo recursivo. De esta forma, podemos utilizar el principio de inducción estructural para demostrar propiedades de funciones definidas por recursión (sobre un tipo de dato recursivo).

Por ejemplo, el principio de inducción sobre listas está formalizado en Isabelle mediante el teorema `list.induct`.

```
thm list.induct
```

```
?P [] ⟹ (⋀x1 x2. ?P x2 ⟹ ?P (x1 # x2)) ⟹ ?P ?list
```

En la teoría `List.thy` está definida la concatenación de listas, que se representa por `@`, a partir de las siguientes reglas:

```
append_Nil:  [] @ ?ys = ?ys
append_Cons: (?x # ?xs) @ ?ys = ?x # ?xs @ ?ys
```

Probemos usando inducción sobre listas que la concatenación de listas es asociativa.

```
lemma conc_asociativa: "xs @ (ys @ zs) = (xs @ ys) @ zs"
  by (induct xs) auto
```

Como podemos observar, basta indicar sobre qué elemento queremos realizar la inducción para que Isabelle infiera qué principio de inducción aplicar.

Al aplicar inducción se generan dos objetivos: el caso base y el paso de inducción del principio de inducción sobre listas, que anunciamos anteriormente, `list.induct`.

```
1. [] @ ys @ zs = ([] @ ys) @ zs
2. ⋀a xs.
       xs @ ys @ zs = (xs @ ys) @ zs ⟹
       (a # xs) @ ys @ zs = ((a # xs) @ ys) @ zs
```

Ambos objetivos se prueban de forma automática aplicando `auto`.

En otros casos, no basta con indicar sobre qué elemento realizar la inducción, pues como mostramos en la sección 2.5 para la función de Ackermann, el principio de inducción generado es específico para dicha función. Veámoslo en un ejemplo. Probaremos que para todo m y n, la función de Akermann A verifica $A(m, n) > n$.

```
lemma "ack m n > n"
  by (induct m n rule: ack.induct) auto
```

Realizamos la demostración de este lema por inducción en `m` y `n`, siguiendo el esquema de inducción `ack.induct`.

2.7.6. Especificación de teoremas

Veremos en esta sección algunos comando para modificar teoremas, en particular, `of`, `THEN` y `OF`.

Para instanciar un teorema se puede usar `of`, que identifica las variables en su orden de aparición de izquierda a derecha. La expresión `[of` v_1 `...` v_k`]` reemplaza las k primeras variables por v_1,..., v_k, respectivamente.

`THEN r` aplica al teorema actual la regla `r` y devuelve la conclusión resultante. Se suele usar `THEN` con reglas de destrucción. Es útil usar `THEN spec`, que elimina el cuantificador de una teorema de la forma $\forall x.P$, o `THEN mp`, que convierte la implicación $P \to Q$ en la regla $\dfrac{P}{Q}$.

`OF` genera una instancia de una regla especificando hechos para sus premisas.

2.7.7. Insertar teoremas y fórmulas como hipótesis

En esta sección veremos como insertar teoremas y fórmulas como hipótesis en la prueba de un teorema.

El método `insert` inserta un teorema como una nueva hipótesis de todos los subobjetivos, mientras que el método `subgoal_tac`, en lugar de insertar un teorema, inserta una fórmula como una nueva hipótesis. Esta fórmula debe ser probada más tarde como un nuevo subobjetivo.

2.8. Pruebas estructuradas en Isar

Entre las carácterísticas más importantes del lenguaje Isar están las siguientes:

- Es un lenguaje estructurado.
- Es más legible y fácilmente entendible por los humanos que el lenguaje interactivo usando el comando `apply`.

Una prueba típica en Isar tiene la siguiente sintaxis:

```
proof
  assume "formula_0"
  have "formula_1" by ...
  ⋮
  have "formula_n" by ...
  ⋮
  show "formula_{n+1}" by ...
qed
```

Esto prueba $formula_0 \Longrightarrow formula_{n+1}$.

Una prueba se comienza con el comando `proof`. Es posible comenzar con un método de prueba, que indicamos entre paréntesis, como por ejemplo `(induct n)`; si no se indica nada, se comienza con un método por defecto; para no comenzar con el método por defecto se añade un guión a `proof`.

Un paso puede suponer una fórmula, con el comando `assume`, o afirmar una fórmula, con el comando `have`, entre otros, junto con su prueba. Las reglas o métodos de dicha prueba se indican tras el comando `by`. Para establecer la conclusión de la prueba se usa el comando `show`.

Las pruebas acaban con `qed`.

Veamos un ejemplo de una prueba sencilla usando el lenguaje Isar.

```
fun intercambia :: "'a × 'b ⇒ 'b × 'a" where
  "intercambia (x,y) = (y,x)"

lemma "intercambia (intercambia (x,y)) = (x,y)"
  proof -
    have "intercambia (intercambia (x,y)) = intercambia (y,x)"
      by simp
    have "... = (x,y)" by simp
    show "intercambia (intercambia (x,y)) = (x,y)" by simp
  qed
```

En esta prueba, usamos los puntos suspensivos para indicar que vamos a usar la igualdad anterior. Como estamos encadenando razonamientos, también se podría haber escrito `also have` en lugar de `have` en esta línea.

Por otra parte, `from` i `have` "$formula$" `by ...` indica que la prueba de la fórmula se haga por el método indicado tras `by` usando el hecho i, que debemos etiquetar previamente. Esto también puede hacerse mediante `have` "$formula$" `using` i `by ...`. Veamos un ejemplo.

```
fun repite :: "nat ⇒ 'a ⇒ 'a list" where
  "repite 0 x       = []"
| "repite (Suc n) x = x # (repite n x)"

lemma "length (repite n x) = n"
proof (induct n)
  show "length (repite 0 x) = 0" by simp
next
  fix n
  assume HI: "length (repite n x) = n"
  have "length (repite (Suc n) x) = length (x # (repite n x))"
    by simp
  also have "... = 1 + length (repite n x)" by simp
  also have "... = 1 + n" using HI by simp
  finally show "length (repite (Suc n) x) = Suc n" by simp
qed
```

Se ha definido en primer lugar una función `repite n x` que genera una lista que contiene `n` veces el elemento `x`. Se prueba a continuación que, efectivamente, la longitud de dicha lista es `n`.

En esta demostración comenzamos especificando el método que se va a usar: inducción sobre `n`. Se generan dos objetivos:

```
1. length (repite 0 x) = 0
2. ⋀n. length (repite n x) = n ⟹
     length (repite (Suc n) x) = Suc n
```

El primero se prueba de forma automática por simplificación. Tras usar el comando `next` comienza la prueba del segundo objetivo. Podemos destacar el uso de `also have` para encadenar razonamientos, y de `finally show` en lugar de `show` para indicar el último paso del razonamiento.

Por último, veamos un ejemplo en el que se introduzcan nuevas variables locales, lo cual se hace con el comando `fix`. Probaremos nuevamente que la concatenación de listas posee la propiedad asociativa, como hicimos en la Sección 2.7.5, aunque esta vez usando el lenguaje Isar.

```
lemma "xs @ (ys @ zs) = (xs @ ys) @ zs"
proof (induct xs)
  show "[] @ (ys @ zs) = ([] @ ys) @ zs" by simp
next
  fix x xs
  assume HI: "xs @ (ys @ zs) = (xs @ ys) @ zs"
  have "(x # xs) @ (ys @ zs) = x # (xs @ (ys @ zs))" by simp
  also have "... = x # ((xs @ ys) @ zs)" using HI by simp
  also have "... = ((x # xs) @ ys) @ zs" by simp
```

```
  finally show "(x # xs) @ (ys @ zs) = ((x # xs) @ ys) @ zs" by simp
qed
```

Para probar el segundo objetivo

```
⋀a xs. xs @ ys @ zs = (xs @ ys) @ zs ⟹
     (a # xs) @ ys @ zs = ((a # xs) @ ys) @ zs
```

introducimos variables locales `x` y `xs`, mediante `fix x xs`.

2.9. Ejemplos

Presentamos en esta sección algunos ejemplos para ilustrar los conceptos vistos durante este capítulo.

Como primer ejemplo, definamos un tipo de dato recursivo. Definiremos un árbol binario `arbolB` como una hoja o un nodo aplicado a dos árboles `arbolB`.

```
datatype 'a arbolB = Hoja "'a"
                   | Nodo "'a" "'a arbolB" "'a arbolB"
```

Al ser un tipo de dato recursivo, Isabelle genera el siguiente principio de inducción.

```
thm arbolB.induct

(⋀a. ?P (Hoja a)) ⟹
(⋀a arbolB1 arbolB2.
    ?P arbolB1 ⟹ ?P arbolB2 ⟹ ?P (Nodo a arbolB1 arbolB2)) ⟹
?P ?arbolB
```

Definiremos ahora una función sobre árboles binarios. La siguiente función `espejo` aplicada a un árbol binario devuelve su imagen especular.

```
fun espejo :: "'a arbolB ⇒ 'a arbolB" where
  "espejo (Hoja x) = (Hoja x)"
| "espejo (Nodo x i d) = (Nodo x (espejo d) (espejo i))"
```

Nuevamente, al ser una función recursiva, se genera un principio de inducción. Es el mismo que para el tipo de dato `arbolB`.

```
thm espejo.induct

(⋀x. ?P (Hoja x)) ⟹
(⋀x i d. ?P d ⟹ ?P i ⟹ ?P (Nodo x i d)) ⟹ ?P ?a0.0
```

Demostremos que la función `espejo` es involutiva, es decir, para cualquier árbol `a`, `espejo (espejo (a)) = a`.

Lo haremos en primer lugar de forma estructurada, usando el lenguaje Isar.

```
lemma espejo_involutiva:
  fixes a :: "'b arbolB"
  shows "espejo (espejo a) = a" (is "?P a")
proof (induct a)
  fix x
  show "?P (Hoja x)" by simp
next
  fix x
  fix i assume h1: "?P i"
  fix d assume h2: "?P d"
  show "?P (Nodo x i d)"
  proof -
    have "espejo(espejo(Nodo x i d)) =
          espejo(Nodo x (espejo d) (espejo i))"
      by simp
    also have "... =
               Nodo x (espejo (espejo i)) (espejo (espejo d))"
      by simp
    also have "... = Nodo x i d" using h1 h2 by simp
    finally show ?thesis .
 qed
qed
```

En esta prueba:

- `fixes a :: "'b arbolB"` es una abreviatura de "sea `a` un árbol binario cuyos elementos son de tipo `b`".
- `(induct a)` indica que el método de demostración es por inducción en el árbol `a`. Se generan dos casos:

  1. `⋀a. espejo (espejo (Hoja a)) = Hoja a`
  2. `⋀a1 a2 a3.`
     `espejo (espejo a2) = a2 ⟹`
     `espejo (espejo a3) = a3 ⟹`
     `espejo (espejo (Nodo a1 a2 a3)) = Nodo a1 a2 a3`

- `fix x` es una abreviatura de "sea `x` un elemento cualquiera".

Probemos ahora este mismo lema de forma interactiva, usando `apply`.

```
lemma espejo_involutiva2:
  "espejo (espejo (a)) = a"
  apply (induct a)
  apply auto
  done
```

Por último, lo probamos de manera automática, usando `by`:

```
lemma espejo_involutiva3:
  "espejo (espejo (a)) = a"
  by (induct a) auto
```

Veamos ahora un ejemplo distinto. Se define la clausura reflexiva y transitiva de una relación `r` como la menor relación reflexiva y transitiva que contiene a `r`. Como las relaciones binarias son conjuntos de pares, se pueden definir inductivamente.

Vamos a definir la clausura reflexiva y transitiva inductivamente, como conjunto, y la representaremos por `r*`.

```
inductive_set
  crt :: "('a × 'a) set ⇒ ('a × 'a) set"   ("_*" [1000] 999)
  for r :: "('a × 'a) set"
where
  crt_refl [iff]: "(x,x) ∈ r*"
| crt_paso:       "(x,y) ∈ r ⟹ (y,z) ∈ r* ⟹ (x,z) ∈ r*"
```

La definición consta de dos reglas.

- A la regla de reflexividad se le añade el atributo `iff` para aumentar la automatización.
- A la regla del paso no se le añade ningún atributo.

El esquema de inducción generado por la definición inductiva de `r*` es el siguiente:

```
(?x1.0, ?x2.0) ∈ ?r* ⟹
(⋀x. ?P x x) ⟹
(⋀x y z. (x, y) ∈ ?r ⟹ (y, z) ∈ ?r* ⟹ ?P y z ⟹ ?P x z) ⟹
?P ?x1.0 ?x2.0
```

Vamos a probar que la relación `r*` es reflexiva. Se puede probar de forma automática usando la regla `crt_refl`.

```
lemma "(x,x) ∈ r*"
  by (rule crt_refl)
```

Probemos ahora que la relación `r*` contiene a `r`, nuevamente de forma automática.

```
lemma [intro]: "(x,y) ∈ r ⟹ (x,y) ∈ r*"
by (blast intro: crt_paso)
```

La ventaja de este lema es que puede ser declarado como una regla de introducción porque `r*` sólo ocurre en la conclusión y no en la premisa. Con esta declaración, algunas demostraciones que usan `crt_paso` se hacen de manera automática.

Vamos a probar que la relación `r*` es transitiva. Para ello usaremos el esquema de inducción `crt.induct`.

En primer lugar lo probaremos de forma interactiva, usando `apply`, y de forma automática.

```
lemma crt_trans_1: "(x,y) ∈ r* ⟹ (y,z) ∈ r* ⟹ (x,z) ∈ r*"
apply (induct rule: crt.induct)
apply (auto simp add: crt_paso)
done

lemma crt_trans_2: "(x,y) ∈ r* ⟹ (y,z) ∈ r* ⟹ (x,z) ∈ r*"
by (induct rule: crt.induct)
   (auto simp add: crt_paso)
```

Lo probaremos ahora de forma estructurada usando el lenguaje Isar, aunque reformularemos el lema para que la variable `y` aparezca también en las conclusiones.

```
lemma crt_trans [rule_format]:
  "(x,y) ∈ r* ⟹ (y,z) ∈ r* ⟶ (x,z) ∈ r*"
proof (induction rule: crt.induct)
  show "⋀x. (x,z) ∈ r* ⟶ (x,z) ∈ r*" by simp
next
  fix x y u
  assume H1: "(x,y) ∈ r" and
         H2: "(y,u) ∈ r*" and
         H3: "(u,z) ∈ r* ⟶ (y,z) ∈ r*"
  show "(u,z) ∈ r* ⟶ (x,z) ∈ r*"
  proof
    assume "(u,z) ∈ r*"
    with H3 have "(y,z) ∈ r*" ..
    with H1 show "(x,z) ∈ r*" by (rule crt_paso)
  qed
qed
```

La reformulación anterior es un caso particular de la siguiente heurística:

> "Para probar una fórmula por inducción sobre $(x_1, \ldots, x_n) \in R$, poner todas las premisas conteniendo cualquiera de las x_i en la conclusión usando $\longrightarrow$."

El atributo `rule_format` transforma $\longrightarrow$ en $\Longrightarrow$.

Probemos a continuación que `r*` es realmente la clausura reflexiva y transitiva de `r`, esto es, la menor relación reflexiva y transitiva que contiene a `r`. Ésta última es formalizada en Isabelle como sigue:

```
inductive_set
  crt2 :: "('a × 'a)set ⇒ ('a × 'a)set"
  for r :: "('a × 'a)set"
where
  "(x,y) ∈ r ⟹ (x,y) ∈ crt2 r"
| "(x,x) ∈ crt2 r"
| "(x,y) ∈ crt2 r ⟹ (y,z) ∈ crt2 r ⟹ (x,z) ∈ crt2 r"
```

Para probar que ambas definiciones de la clausura reflexiva y transitiva de `r` son equivalentes vamos a probar que `crt2` está contenido en `r*` y que `r*` está contenido en `crt2`.

Para probar lo primero vamos a usar el esquema de inducción generado automáticamente por la definición inductiva de `crt2`. Dicho esquema es el siguiente:

```
(?x1.0, ?x2.0) ∈ crt2 ?r ⟹
(⋀x y. (x, y) ∈ ?r ⟹ ?P x y) ⟹
(⋀x. ?P x x) ⟹
(⋀x y z.
    (x, y) ∈ crt2 ?r ⟹
    ?P x y ⟹ (y, z) ∈ crt2 ?r ⟹ ?P y z ⟹ ?P x z) ⟹
?P ?x1.0 ?x2.0
```

Demostremos ahora el lema.

```
lemma "(x,y) ∈ crt2 r ⟹ (x,y) ∈ r*"
proof (induction rule: crt2.induct)
  fix x y
  assume "(x,y) ∈ r"
  thus "(x,y) ∈ r*" by blast
next
  fix x
  show "(x,x) ∈ r*" by blast
next
  fix x y z
  assume H1: "(x,y) ∈ crt2 r" and
         H2: "(x,y) ∈ r*"     and
         H3: "(y,z) ∈ crt2 r" and
         H4: "(y,z) ∈ r*"
  show "(x,z) ∈ r*" using H2 H4 by (rule crt_trans_1)
qed
```

Podemos probar también este lema de forma interactiva.

```
lemma "(x,y) ∈ crt2 r ⟹ (x,y) ∈ r*"
  apply (induct rule: crt2.induct)
  apply blast
```

```
  apply blast
  apply (rule_tac y = y in crt_trans_1)
  apply assumption+
  done
```

En esta prueba, el comando `rule_tac y = y in crt_trans_1` instacia en lema `crt_trans_1` y lo aplica. La última línea `assumption+` aplica las hipótesis tantas veces como sea posible.

Probamos ahora la otra contención, es decir, que `r*` está contenido en `crt2`.

```
lemma "(x,y) ∈ r* ⟹ (x,y) ∈ crt2 r"
proof (induction rule:crt.induct)
  fix x
  show "(x,x) ∈ crt2 r" by (rule crt2.intros(2))
next
  fix x y z
  assume H1: "(x,y) ∈ r" and
         H2: "(y,z) ∈ r*" and
         H3: "(y,z) ∈ crt2 r"
  have "(x,y) ∈ crt2 r" using H1 by (rule crt2.intros(1))
  thus "(x,z) ∈ crt2 r" using H3 by (rule crt2.intros(3))
qed
```

Nuevamente, lo probamos ahora de forma interactiva:

```
lemma "(x,y) ∈ r* ⟹ (x,y) ∈ crt2 r"
  apply (induct rule: crt.induct)
  apply (rule crt2.intros(2))
  apply (rule_tac y = y in crt2.intros(3))
  apply (rule crt2.intros(1), assumption)
  apply assumption
  done
```

En esta demostración,

```
apply (rule crt2.intros(1), assumption)
```

aplica la regla `crt2.intros(1)` y, consecutivamente, las hipótesis.

Capítulo 3

Lógica Proposicional

3.1. Sintaxis de la lógica proposicional

Los elementos básicos del lenguaje de la lógica proposicional son:

- $\top$ o 'Verdadero' y $\bot$ o 'Falso';
- un conjunto infinito numerable de variable atómicas o proposicionales, denotadas comúnmente 'p', 'q', 'r', ...;
- las conectivas lógicas, que pueden tener un argumento, es decir, ser monarias, o tener dos argumentos, es decir, ser binarias. La única conectiva monaria de interés es la negación ($\neg$), mientras que se pueden considerar distintas conectivas binarias. Nosotros consideraremos la conjunción ($\wedge$), la disyunción ($\vee$) y la implicación ($\Rightarrow$).

En esta sección daremos definiciones y resultados independientes de las conectivas lógicas elegidas para representar la lógica proposicional.

Definición 3.1.1. Una *fórmula atómica o átomo* es una variable proposicional, $\top$ o $\bot$.

Definición 3.1.2. El conjunto de *fórmulas proposicionales* es el menor conjunto $\mathbf{P}$ tal que

1. si F es una fórmula atómica, entonces $F \in \mathbf{P}$,
2. si $F \in \mathbf{P}$, entonces $\neg F \in \mathbf{P}$,
3. si $\circ$ es un símbolo de conectiva binaria y $F_1, F_2 \in \mathbf{P}$, entonces $(F_1 \circ F_2) \in \mathbf{P}$.

Enunciamos a continuación un resultado que nos permitirá probar propiedades sobre fórmulas proposicionales.

Teorema 3.1.1. ***(Principio de Inducción Estructural)*** *Toda fórmula proposicional tiene una propiedad* $\mathbf{Q}$ *si:*

***Paso base**: toda fórmula atómica tiene la propiedad* **Q**.

***Pasos de inducción**:*

si F tiene la propiedad **Q** *también la tiene $\neg F$;*

si F_1 y F_2 tienen la propiedad **Q**, *entonces también la tiene $(F_1 \circ F_2)$, donde $\circ$ es un símbolo que representa una conectiva binaria.*

Demostración. Sea $\mathbf{Q}^*$ el conjunto de fórmulas proposicionales que tienen la propiedad **Q**. El paso base y los pasos de inducción aseguran que $\mathbf{Q}^*$ verifica las condiciones de la definición 3.1.2 de fórmula proposicional. Como el conjunto **P** de fórmulas proposicionales es el conjunto más pequeño que satisface dichas condiciones, **P** debe ser un subconjunto de $\mathbf{Q}^*$, y entonces todas las fórmulas proposicionales tienen la propiedad **Q**. □

El siguiente principio básico que enunciamos nos permitirá definir funciones sobre el conjunto de las fórmulas proposicionales.

Teorema 3.1.2. ***(Principio de Recursión Estructural)** Hay una, y sólo una, función f definida sobre el conjunto* **P** *de fórmulas proposicionales tal que:*

***Paso base**: el valor de f se especifica explícitamente en las fórmulas atómicas.*

***Pasos de inducción**:*

el valor de f en $\neg F$ se especifica en términos del valor de f en F;

el valor de f en $(F_1 \circ F_2)$ se especifica en términos del valor de f en F_1 y F_2, donde $\circ$ es un símbolo que representa una conectiva binaria.

Omitimos la prueba de este teorema, que es similar a la anterior, pero que requiere de principios de existencia para funciones.

Definimos ahora la noción de subfórmula. Informalmente, una subfórmula de una fórmula es una subcadena de esta que, en sí misma, es una fórmula. La siguiente es una caracterización más rigurosa.

Definición 3.1.3. Las *subfórmulas inmediatas* se definen como sigue:

1. Una fórmula atómica no tiene subfórmulas inmediatas.
2. La única subfórmula inmediata de $\neg F$ es F.
3. Para un símbolo binario $\circ$, las subfórmulas inmediatas de $(F_1 \circ F_2)$ son F_1 y F_2.

Definición 3.1.4. Sea F una fórmula. El conjunto de *subfórmulas* de F es el conjunto más pequeño **S** que contiene a F y a las subfórmulas inmediatas de cada miembro del conjunto. Se dice que F es una subfórmula impropia de sí misma.

3.2. Semántica de la lógica proposicional

En la lógica clásica, el significado de una fórmula proposicional puede tomar dos valores. Tomamos como el espacio de los *valores de verdad* el conjunto $\mathbf{Tr} = \{\mathbf{t}, \mathbf{f}\}$. Aquí $\mathbf{t}$ y $\mathbf{f}$ representan dos objetos distintos: $\mathbf{t}$ representa la verdad y $\mathbf{f}$ la falsedad.

Es evidente que se pueden definir $2^2 = 4$ conectivas monarias y $2^4 = 16$ conectivas binarias. Dejando a un lado las constantes y funciones identidad, es bien conocido que usando un subconjunto de las demás conectivas es posible definirlas todas. Por eso sólo consideraremos las conectivas lógicas $\neg$, $\wedge$, $\vee$ y $\Rightarrow$.

Ahora vamos ver cómo interpretar cada una de éstas. Definimos funciones de verdad sobre $\mathbf{Tr}$ como sigue:

$$\begin{array}{rccc} H_\neg : & \mathbf{Tr} & \to & \mathbf{Tr} \\ & \mathbf{t} & \mapsto & \mathbf{f} \\ & \mathbf{f} & \mapsto & \mathbf{t} \end{array}$$

$$\begin{array}{rc} H_\wedge : & \mathbf{Tr} \times \mathbf{Tr} \to \mathbf{Tr} \\ & (\mathbf{t}, \mathbf{t}) \mapsto \mathbf{t} \\ & (i, j) \mapsto \mathbf{f} \end{array} \qquad \begin{array}{rc} H_\vee : & \mathbf{Tr} \times \mathbf{Tr} \to \mathbf{Tr} \\ & (\mathbf{f}, \mathbf{f}) \mapsto \mathbf{f} \\ & (i, j) \mapsto \mathbf{t} \end{array} \qquad \begin{array}{rc} H_\Rightarrow : & \mathbf{Tr} \times \mathbf{Tr} \to \mathbf{Tr} \\ & (\mathbf{t}, \mathbf{f}) \mapsto \mathbf{f} \\ & (i, j) \mapsto \mathbf{t} \end{array}$$

En las tres últimas aplicaciones, entendemos que cuando consideramos (i, j) sobre $\mathbf{Tr} \times \mathbf{Tr}$ no estamos en el caso anterior.

Estudiamos asimismo cómo interpretar una fórmula proposicional en función de las conectivas lógicas que aparecen en ella.

Definición 3.2.1. Una *interpretación* es una aplicación v del conjunto de las fórmulas proposicionales al conjunto $\mathbf{Tr}$ satisfaciendo las condiciones:

1. $v(\top) = \mathbf{t}$; $v(\bot) = \mathbf{f}$;
2. $v(\neg F) = H_\neg(v(F))$;
3. $v(F_1 \circ F_2) = H_\circ(v(F_1), v(F_2))$, para cualquier conectiva binaria $\circ$ de las consideradas anteriormente.

En la última condición de esta definición se pueden considerar todas las conectivas monarias y binarias, pero no estamos interesados en ello.

Proposición 3.2.1. *Para cada función f del conjunto de variables proposicionales al conjunto* $\mathbf{Tr}$ *hay una interpretación que coincide con f en las variables proposicionales.*

Proposición 3.2.2. *Si v_1 y v_2 son dos interpretaciones que coinciden en un conjunto S de variables proposicionales, entonces v_1 y v_2 coinciden en todas las fórmulas proposicionales que contienen sólo variables proposicionales de S.*

Estamos interesados en aquellas fórmulas que son verdaderas por su estructura lógica, independientemente del significado de las variables proposicionales que aparecen en ellas. Son las llamadas tautologías, que definimos formalmente a continuación.

Definición 3.2.2. Una fórmula proposicional F es una *tautología* si $v(F) = \mathbf{t}$ para cualquier interpretación v.

Por la Proposición 3.2.1, una interpretación viene determinada por su valor en las variables proposicionales, y por la Proposición 3.2.2, para comprobar si una fórmula es una tautología, sólo necesitamos considerar este valor en las variables proposicionales que aparecen en dicha fórmula.

Definición 3.2.3. Una fórmula proposicional F es *satisfacible* si existe alguna interpretación v tal que $v(F) = \mathbf{t}$. En ese caso se dice que v es un modelo de F. Una fórmula F es *insatisfacible* si $v(F) = \mathbf{f}$ para cualquier interpretación v.

Definición 3.2.4. Un conjunto de fórmulas proposicionales S es consistente si existe alguna interpretación v tal que su valor sobre cada elemento de S es $\mathbf{t}$, en cuyo caso se dice que v es modelo de S. Un conjunto de fórmulas proposicionales S es inconsistente si el valor de toda interpretación v sobre cada miembro de S es $\mathbf{f}$.

Hay una conexión entre tautología e insatisfacibilidad. Una fórmula F es una tautología si, y sólo si, $\neg F$ es insatisfacible.

Definimos a continuación el concepto de consecuencia lógica.

Definición 3.2.5. Decimos que una fórmula proposicional F es *consecuencia lógica* de un conjunto S de fórmulas proposicionales, y escribimos $S \models F$, si toda interpretación que es modelo de S también lo es de F.

Proposición 3.2.3. *Sean $S = \{F_1, \ldots, F_n\}$ un conjunto de fórmulas, y F una fórmula proposicional. Entonces $S \models F$ si, y sólo si, $\{F_1, \ldots, F_n, \neg F\}$ es inconsistente.*

Omitimos la prueba de esta proposición, ya que se sigue inmediatamente de las definiciones de consecuencia lógica, inconsistencia e interpretación.

3.3. Teorema de reemplazamiento

Supongamos que $F(A_1, \ldots, A_n)$ es una fórmula proposicional cuyas variables proposicionales están entre $A_1, \ldots, A_n$. Sean $X_1, \ldots, X_n$ n fórmulas proposicionales arbitrarias. Denotamos por $F(X_1, \ldots, X_n)$ el resultado de reemplazar simultáneamente todas las ocurrencias de A_i por X_i in $F(A_1, \ldots, A_n)$.

Teorema 3.3.1. ***(Teorema de Reemplazamiento)*** *Supongamos que $F(P)$, X e Y son fórmulas proposicionales, y v es una interpretación. Si $v(X) = v(Y)$ entonces $v(F(X)) = v(F(Y))$.*

Demostración. Digamos, únicamente en esta demostración, que una fórmula $F(P)$ es *buena* si tiene la propiedad

$$v(X) = v(Y) \text{ implica } v(F(X)) = v(F(Y)) \text{ para cualquier } X, Y, v. \tag{3.1}$$

Vamos a probar que todas las fórmulas proposicionales son *buenas*, según esta definición. Para ello usaremos el Principio de Inducción Estructural.

(Paso base) Supongamos que $F(P)$ es atómica. Entonces hay dos casos:

1. $F(P) = P$. Entonces $F(X) = X$ y $F(Y) = Y$, y (3.1) es trivialmente verdadera.
2. $F(P) \neq P$. Entonces $F(X) = F(Y) = F(P)$, y (3.1) es trivialmente verdadera de nuevo.

Por tanto todas las fórmulas atómicas son *buenas*.

(Paso de inducción)

Supongamos que $G(P)$ es una fórmula proposicional *buena*, y sea $F(P) = \neg G(P)$. Si $v(X) = v(Y)$, se tiene

$$\begin{array}{lll} v(F(X)) & = v(\neg G(X)) & \text{(definición de } F\text{)} \\ & = H_\neg(v(G(X)) & \text{(definición de interpretación)} \\ & = H_\neg(v(G(Y)) & \text{(hipótesis de inducción)} \\ & = v(\neg G(Y)) & \text{(definición de interpretación)} \\ & = v(F(Y)) & \text{(definición de } F\text{)} \end{array}$$

Entonces $F(P)$ es *buena*.

Supongamos ahora que F_1 y F_2 son dos fórmulas proposicionales *buenas*, y sea $F(P) = F_1(P) \circ F_2(P)$, donde $\circ$ es algún símbolo binario. Si $v(X) = v(Y)$, se tiene

$$\begin{array}{lll} v(F(X)) & = v(F_1(X) \circ F_2(X)) & \text{(definición de } F\text{)} \\ & = H_\circ(v(F_1(X), v(F_2(X))) & \text{(definición de interpretación)} \\ & = H_\circ(v(F_1(Y), v(F_2(Y))) & \text{(hipótesis de inducción)} \\ & = v(F_1(Y) \circ F_2(Y)) & \text{(definición de interpretación)} \\ & = v(F(Y)) & \text{(definición de } F\text{)} \end{array}$$

Entonces $F(P)$ es *buena*.

Por tanto, todas las fórmulas proposicionales son *buenas*. □

3.4. Notación uniforme

La notación uniforme nos permite definir cualquier conectiva lógica en función de otras, consiguiendo así reducir el número de casos a considerar al probar un teorema.

Agruparemos todas las fórmulas proposicionales de la forma $(F_1 \circ F_2)$ y $\neg(F_1 \circ F_2)$, donde $\circ$ es una conectiva lógica, en dos categorías: aquellas de carácter *conjuntivo*, que denotaremos fórmulas α, y aquellas de carácter *disyuntivo*, que denotaremos fórmulas β. Para cada una de estas fórmulas definimos dos componentes, que denotaremos α_1 y α_2, o β_1 y β_2, respectivamente. Representamos esto en la siguiente tabla (Figura 3.1).

Conjunción			Disyunción		
α	α_1	α_2	β	β_1	β_2
$F_1 \wedge F_2$	F_1	F_2	$\neg(F_1 \wedge F_2)$	$\neg F_1$	$\neg F_2$
$\neg(F_1 \vee F_2)$	$\neg F_1$	$\neg F_2$	$F_1 \vee F_2$	F_1	F_2
$\neg(F_1 \Rightarrow F_2)$	F_1	$\neg F_2$	$F_1 \Rightarrow F_2$	$\neg F_1$	F_2

Figura 3.1: Fórmulas α y β y sus componentes

Formalizamos a continuación lo que hemos explicado anteriormente.

Proposición 3.4.1. *Para toda interpretación v, y para toda fórmula α y β:*

$$\begin{aligned} v(\alpha) &= H_\wedge(v(\alpha_1), v(\alpha_2)) \\ v(\beta) &= H_\vee(v(\beta_1), v(\beta_2)) \end{aligned}$$

Establecemos a continuación una versión alternativa del Principio de Inducción Estructural, Teorema 3.1.1. A partir de ahora será esta versión la que utilizaremos.

Teorema 3.4.1. ***(Principio de Inducción Estructural)*** *Toda fórmula proposicional tiene una propiedad* **Q** *si:*

> ***Paso base**: toda fórmula atómica y su negación tienen la propiedad* **Q**.
>
> ***Pasos de inducción**:*
>
> > *si F tiene la propiedad* **Q** *también la tiene $\neg\neg F$;*
> >
> > *si α_1 y α_2 tienen la propiedad* **Q**, *también la tiene α;*
> >
> > *si β_1 y β_2 tienen la propiedad* **Q**, *también la tiene β.*

Demostración. Denominemos temporalmente en esta demostración a una fórmula F *buena* si tanto F como $\neg F$ tienen la propiedad **Q**. Si probamos que toda fórmula proposicional es *buena*, entonces tendremos que toda fórmula tiene la propiedad **Q**. Lo haremos haciendo uso del Principio de Inducción Estructural, Teorema 3.1.1.

- Si F es atómica, entonces F es *buena* por el paso base.
- Si F es *buena*, vamos a probar que también lo es $\neg F$. Si F es buena, tanto F como $\neg F$ tienen la propiedad **Q**. Como F tiene la propiedad **Q**, por el primero de los pasos de inducción también la tiene $\neg\neg F$. Hemos probado que $\neg F$ y $\neg\neg F$ tienen la propiedad **Q**, por lo que $\neg F$ es *buena*.
- Supongamos que F_1 y F_2 son *buenas*, y que $\circ$ es una conectiva binaria. Vamos a probar que $(F_1 \circ F_2)$ es *buena*. Si $(F_1 \circ F_2)$ es una fórmula α, $\neg(F_1 \circ F_2)$ es una fórmula β; si $(F_1 \circ F_2)$ es una fórmula β, $\neg(F_1 \circ F_2)$ es una fórmula α. En ambos casos, $\alpha_1, \beta_1 \in \{F_1, \neg F_1\}$. F_1 es *buena* por hipótesis, y así también lo es $\neg F_1$ por el párrafo precedente. Por tanto α_1 y β_1 tienen la propiedad **Q**. $\alpha_2, \beta_2 \in \{F_2, \neg F_2\}$, y usando un razonamiento análogo ambas tienen la propiedad **Q**. De esto se sigue que $(F_1 \circ F_2)$ es *buena*.

Usando el Principio de Inducción Estructural original, toda fórmula proposicional es *buena*, por lo que toda fórmula proposicional tiene la propiedad **Q**. □

Hay también una versión del Principio de Recursión Estructural, Teorema 3.1.2, que usa la notación uniforme. Lo enunciamos a continuación, aunque omitimos la prueba.

Teorema 3.4.2. ***(Principio de Recursión Estructural)*** *Hay una, y sólo una, función f definida sobre el conjunto* **P** *de fórmulas proposicionales tal que:*

> ***Paso base**: el valor de f se especifica explícitamente en las fórmulas atómicas y en su negación.*
>
> ***Pasos de inducción**:*
>
> > *el valor de f en $\neg\neg F$ se especifica en términos del valor de f en F;*
> >
> > *el valor de f en α se especifica en términos del valor de f en α_1 y α_2.*
> >
> > *el valor de f en β se especifica en términos del valor de f en β_1 y β_2.*

3.5. Lema de Hintikka

Nuestro objetivo es probar la completitud de la deducción natural. Existen muchas formas de hacer esto, aunque nosotros usaremos un método general que se puede extender a la lógica de primer orden. Empezaremos probando la versión proposicional de un lema debido a J. Hintikka.

Definición 3.5.1. Un conjunto **H** de fórmulas proposicionales es llamado un conjunto de Hintikka si:

1. Para cualquier variable proposicional p, no pueden pertenecer a la vez a **H** p y $\neg p$;

2. $\bot \notin \mathbf{H}$; $\neg\top \notin \mathbf{H}$;

3. Si $\neg\neg F \in \mathbf{H}$, entonces $F \in \mathbf{H}$;

4. Si $\alpha \in \mathbf{H}$, entonces $\alpha_1 \in \mathbf{H}$ y $\alpha_2 \in \mathbf{H}$;

5. Si $\beta \in \mathbf{H}$, entonces $\beta_1 \in \mathbf{H}$ o $\beta_2 \in \mathbf{H}$;

donde las fórmulas α, α_1, α_2, así como β, β_1 y β_2, son las consideradas en la Figura 3.1.

Proposición 3.5.1. ***(Lema de Hintikka)*** *Todo conjunto de Hintikka proposicional es consistente.*

Demostración. Sea $\mathbf{H}$ un conjunto de Hintikka. Vamos a probar que existe una interpretación tal que su valor en cada elemento de $\mathbf{H}$ es $\mathbf{t}$. Como vimos en las Proposiciones 3.2.1 y 3.2.2, para definir una interpretación es suficiente hacerlo sobre las variables proposicionales. Entonces, sea f definida como sigue: para cualquier variable proposicional p,

$$f(p) = \begin{cases} \mathbf{t} & \text{si } p \in \mathbf{H} \\ \mathbf{f} & \text{si } \neg p \in \mathbf{H} \\ \mathbf{f} & \text{si } p, \neg p \notin \mathbf{H} \end{cases}$$

El último caso de la definición podría haber sido definido de otra forma arbitrariamente, aunque lo hacemos así por fijar la definición.

Notemos que la condición 1 de la definición de conjunto de Hintikka asegura que f está bien definida. Sea ahora v la interpretación que extiende f. Entonces el valor de v en cada elemento de $\mathbf{H}$ es $\mathbf{t}$. Veámoslo:

Definamos una propiedad Q como sigue. Diremos que una fórmula F tiene la propiedad Q si $F \in \mathbf{H} \Rightarrow v(F) = \mathbf{t}$. En otras palabras, F tiene la propiedad Q si F no está en $\mathbf{H}$ o si $v(F) = \mathbf{t}$. Usaremos el Principio de Inducción Estructural, Teorema 3.4.1, para probar que todas las fórmulas proposicionales tienen la propiedad Q.

Paso base: veamos que toda fórmula atómica y su negación tienen la propiedad $\mathbf{Q}$.

- $v(\top) = \mathbf{t}$ por definición de interpretación, por lo que también se tiene $v(\neg\bot) = \mathbf{t}$.
- $\bot, \neg\top \notin \mathbf{H}$ por la condición 2 de la definición de conjunto de Hintikka.
- El valor de v sobre cualquier variable proposicional perteneciente a $\mathbf{H}$ es $\mathbf{t}$. Si $\neg p \in \mathbf{H}$, siendo p una variable proposicional, entonces $v(p) = \mathbf{f}$, por lo que $v(\neg p) = \mathbf{t}$.

Pasos de inducción:

- Si F tiene la propiedad $\mathbf{Q}$ también la tiene $\neg\neg F$.

  Supongamos que $\neg\neg F \in \mathbf{H}$. Entonces $F \in \mathbf{H}$ por la condición 3 de la definición de conjunto de Hintikka. Como F tiene la propiedad $\mathbf{Q}$ y pertenece a $\mathbf{H}$, debe ser $v(F) = \mathbf{t}$, lo que implica que $v(\neg\neg F) = \mathbf{t}$.

- Si α_1 y α_2 tienen la propiedad $\mathbf{Q}$, también la tiene α.

  Supongamos que $\alpha \in \mathbf{H}$. Entonces $\alpha_1, \alpha_2 \in \mathbf{H}$ por la condición 4 de la definición de conjunto de Hintikka, y como tienen la propiedad $\mathbf{Q}$, $v(\alpha_1) = v(\alpha_2) = \mathbf{t}$. Entonces $v(\alpha) = \mathbf{t}$ por la Proposición 3.4.1.

- Si β_1 y β_2 tienen la propiedad $\mathbf{Q}$, también la tiene β.

  Supongamos que $\beta \in \mathbf{H}$. Entonces, por la condición 5 de la definición de conjunto de Hintikka, $\beta_1 \in \mathbf{H}$ o $\beta_2 \in \mathbf{H}$. Como ambos tienen la propiedad $\mathbf{Q}$, el valor de v debe ser $\mathbf{t}$ al menos en uno de los dos, por lo que se tiene que $v(\beta) = \mathbf{t}$.

Usando el Principio de Inducción Estructural, concluimos que todas las fórmulas proposicionales tienen la propiedad $\mathbf{Q}$, es decir, que v es modelo de $\mathbf{H}$, por lo que todo conjunto de Hintikka es consistente. □

3.6. Teorema de existencia de modelos

El Lema de Hintikka conecta la sintaxis y la semántica de la lógica proposicional. En esta sección probaremos un teorema más importante que también relaciona la sintaxis y la semántica. La prueba contiene la esencia de un argumento de completitud "estándar". Con este argumento dado de forma abstracta una vez, la completitud de muchos razonamientos lógicos será una consecuencia fácil, en concreto de la deducción natural.

Definiremos para ello la noción de *propiedad de consistencia*, que denominaremos $\mathcal{C}$. Identificaremos asimismo $\mathcal{C}$ con la colección de todos los conjuntos de fórmulas que tienen dicha propiedad, y la denominaremos *clase de consistencia*. Si un conjunto S está en la colección $\mathcal{C}$, nos referiremos a S como $\mathcal{C}$-consistente.

Definición 3.6.1. Sea $\mathcal{C}$ una colección de conjuntos de fórmulas proposicionales. Diremos que $\mathcal{C}$ es una colección de conjuntos con una *propiedad de consistencia proposicional* o una *clase de consistencia proposicional* si se verifican las siguientes condiciones para cada $S \in \mathcal{C}$:

1. Para cualquier variable proposicional p, no pueden pertenecer a la vez p y $\neg p$ a S;

2. $\bot \notin S$; $\neg\top \notin S$;

3. Si $\neg\neg F \in S$, entonces $S \cup \{F\} \in \mathcal{C}$;

4. Si $\alpha \in S$, entonces $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$;

5. Si $\beta \in S$, entonces $S \cup \{\beta_1\} \in \mathcal{C}$ o $S \cup \{\beta_2\} \in \mathcal{C}$.

Antes de enunciar el Teorema de existencia de modelos, daremos varios resultados preliminares.

Definición 3.6.2. Se dice que una clase de consistencia proposicional $\mathcal{C}$ es *cerrada por subconjuntos* si contiene, por cada elemento, todos los subconjuntos de ese elemento.

Proposición 3.6.1. *Toda clase de consistencia proposicional puede ser extendida a una que sea cerrada por subconjuntos.*

Demostración. Sea $\mathcal{C}$ una clase de consistencia proposicional. Vamos a construir una colección de conjuntos $\mathcal{C}'$ tal que:

1. $\mathcal{C} \subseteq \mathcal{C}'$,

2. $\mathcal{C}'$ es cerrada por subconjuntos,

3. $\mathcal{C}'$ es una clase de consistencia proposicional.

Definamos $\mathcal{C}'$ como la colección de todos los conjuntos S que son subconjunto de algún conjunto de $\mathcal{C}$. Probemos las propiedades anteriores.

1. $\mathcal{C} \subseteq \mathcal{C}'$. Es trivial, pues todo conjunto de $\mathcal{C}$ es subconjunto de sí mismo, por lo que pertenece a $\mathcal{C}'$.

2. $\mathcal{C}'$ es cerrada por subconjuntos. Sea $S \in \mathcal{C}'$. Entonces existe un conjunto $S' \in \mathcal{C}$ tal que $S \subseteq S'$. Como todo subconjunto de S también es subconjunto de S', pertenece a $\mathcal{C}'$.

3. $\mathcal{C}'$ es una clase de consistencia proposicional. Probemos cada una de las propiedades que definen a una clase de consistencia proposicional. Para ello sea $S \in \mathcal{C}'$. Por la definición de la colección $\mathcal{C}'$, debe existir un conjunto $S' \in \mathcal{C}$ tal que $S \subseteq S'$.

   a) Para cualquier variable proposicional p, no pueden pertenecer a la vez p y $\neg p$ a S.

   Supongamos por reducción al absurdo que $p, \neg p \in S$. Como $S \subseteq S'$ y $S' \in \mathcal{C}$, entonces $p, \neg p \in S'$, pero esto es una contradicción, pues $\mathcal{C}$ es una clase de consistencia proposicional.

   b) $\bot \notin S$; $\neg\top \notin S$.

   Razonando de forma análoga a la anterior, si $\bot \in S$ o $\neg\top \in S$, como $S \subseteq S'$ y $S' \in \mathcal{C}$, también se tendría $\bot \in S'$ o $\neg\top \in S'$, lo que supondría una contradicción con que $\mathcal{C}$ sea una clase de consistencia proposicional.

c) Si $\neg\neg F \in S$, entonces $S \cup \{F\} \in \mathcal{C}'$.

Si $\neg\neg F \in S$, como $S \subseteq S'$ y $S' \in \mathcal{C}$, entonces $\neg\neg F \in S'$ y, por tanto, $S' \cup \{F\} \in \mathcal{C}$, por ser $\mathcal{C}$ una clase de consistencia proposicional. Como $S \cup \{F\} \subseteq S' \cup \{F\}$, por definición de la clase $\mathcal{C}'$ se tiene el resultado.

d) Si $\alpha \in S$, entonces $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}'$.

Si $\alpha \in S$, como $S \subseteq S'$ y $S' \in \mathcal{C}$, entonces $\alpha \in S'$ y, por tanto, $S' \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$, por ser $\mathcal{C}$ una clase de consistencia proposicional. Como $S \cup \{\alpha_1, \alpha_2\} \subseteq S' \cup \{\alpha_1, \alpha_2\}$, por definición de la clase $\mathcal{C}'$ se tiene el resultado.

e) Si $\beta \in S$, entonces $S \cup \{\beta_1\} \in \mathcal{C}'$ o $S \cup \{\beta_2\} \in \mathcal{C}'$.

Si $\beta \in S$, como $S \subseteq S'$ y $S' \in \mathcal{C}$, entonces $\beta \in S'$ y, por tanto, $S' \cup \{\beta_1\} \in \mathcal{C}$ o $S' \cup \{\beta_2\} \in \mathcal{C}$, por ser $\mathcal{C}$ una clase de consistencia proposicional. Como $S \cup \{\beta_1\} \subseteq S' \cup \{\beta_1\}$ y $S \cup \{\beta_2\} \subseteq S' \cup \{\beta_2\}$, por definición de la clase $\mathcal{C}'$ se tiene el resultado. □

Definición 3.6.3. Se dice que una clase de consistencia proposicional $\mathcal{C}$ es *de carácter finito* si $S \in \mathcal{C}$ si, y sólo si, todo subconjunto finito de S pertenece a $\mathcal{C}$.

Proposición 3.6.2. *Toda clase de consistencia proposicional de carácter finito es cerrada por subconjuntos.*

Demostración. Sea $\mathcal{C}$ una clase de consistencia proposicional de carácter finito, y sea $S \in \mathcal{C}$. Probaremos que $\mathcal{C}$ es cerrada por subconjuntos por reducción al absurdo, suponiendo que existe un subconjunto de S, al que denotaremos por S', que no pertenece a $\mathcal{C}$.

Por ser $\mathcal{C}$ de carácter finito, $S' \notin \mathcal{C}$ si, y sólo si, existe un subconjunto finito de S' que no pertenece a $\mathcal{C}$. Pero dicho subconjunto finito de S' también lo es de S, pues $S' \subseteq S$, lo cual contradice que $S \in \mathcal{C}$ y $\mathcal{C}$ sea de carácter finito. □

Proposición 3.6.3. *Una clase de consistencia proposicional que es cerrada por subconjuntos puede ser extendida a otra de carácter finito.*

Demostración. Sea $\mathcal{C}$ una clase de consistencia proposicional cerrada por subconjuntos. Vamos a construir una colección de conjuntos $\mathcal{C}'$ tal que:

1. $\mathcal{C} \subseteq \mathcal{C}'$,
2. $\mathcal{C}'$ es de carácter finito,
3. $\mathcal{C}'$ es una clase de consistencia proposicional.

Definamos $\mathcal{C}'$ como la colección de todos los conjuntos S tal que todo subconjunto finito de S está en $\mathcal{C}$. Probemos las propiedades anteriores.

1. $\mathcal{C} \subseteq \mathcal{C}'$. Sea $S \in \mathcal{C}$. Como $\mathcal{C}$ es cerrada por subconjuntos, contiene a todos los subconjuntos de S. Entonces $S \in \mathcal{C}'$.

2. $\mathcal{C}'$ es de carácter finito. Hay que probar que $S \in \mathcal{C}'$ si, y sólo si, todo subconjunto finito de S pertenece a $\mathcal{C}'$.

   Supongamos que $S \in \mathcal{C}'$. Entonces todo subconjunto finito de S pertenece a $\mathcal{C}$. Como $C \subseteq \mathcal{C}'$, todo subconjunto finito de S pertenece a $\mathcal{C}'$.

   Supongamos que todo subconjunto finito de S pertenece a $\mathcal{C}'$. Entonces todo subconjunto finito de todo subconjunto finito de S pertenece a $\mathcal{C}$. Es decir, todo subconjunto finito de S pertenece a $\mathcal{C}$. Entonces $S \in \mathcal{C}'$.

3. $\mathcal{C}'$ es una clase de consistencia proposicional. Probemos cada una de las propiedades que definen a una clase de consistencia proposicional. Para ello sea $S \in \mathcal{C}'$.

   a) Para cualquier variable proposicional p, no pueden pertenecer a la vez p y $\neg p$ a S.

   Sea p una variable proposicional, y supongamos $p, \neg p \in S$. Entonces $\{p, \neg p\}$ es un subconjunto finito de S, que pertenece a $\mathcal{C}$ porque $S \in \mathcal{C}'$. Pero esto es una contradicción, pues $\mathcal{C}$ es una clase de consistencia proposicional.

   b) $\bot \notin S$; $\neg\top \notin S$.

   Se prueba haciendo un razonamiento análogo al anterior. Si alguno de estos elementos perteneciera a S, el conjunto formado por este único elemento sería un subconjunto finito de S, por lo que pertenecería a $\mathcal{C}$. Pero nuevamente esto no es posible por ser $\mathcal{C}$ una clase de consistencia proposicional.

   c) Si $\neg\neg F \in S$, entonces $S \cup \{F\} \in \mathcal{C}'$.

   Para probar que $S \cup \{F\} \in \mathcal{C}'$ hay que probar que todo subconjunto finito de $S \cup \{F\} \in \mathcal{C}$.

   Sea S' un subconjunto finito cualquiera de S. Como $\neg\neg F \in S$ y $S \in \mathcal{C}'$, entonces $S' \cup \{\neg\neg F\} \in \mathcal{C}$. Por ser $\mathcal{C}$ una clase de consistencia proposicional, $S' \cup \{\neg\neg F\} \cup \{F\} \in \mathcal{C}$. Entonces todo subconjunto de $S' \cup \{\neg\neg F\} \cup \{F\}$ pertenece a $\mathcal{C}$ porque es cerrada por subconjuntos.

   Finalmente, como S' es un subconjunto finito cualquiera de S, tenemos el resultado.

   d) Si $\alpha \in S$, entonces $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}'$.

   Siguiendo el razonamiento anterior, para probar que $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}'$ hay que probar que todo subconjunto finito de $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.

Sea S' un subconjunto finito cualquiera de S. Como $\alpha \in S$ y $S \in \mathcal{C}'$, entonces $S' \cup \{\alpha\} \in \mathcal{C}$. Por ser $\mathcal{C}$ una clase de consistencia proposicional, $S' \cup \{\alpha\} \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$. Entonces todo subconjunto de $S' \cup \{\alpha\} \cup \{\alpha_1, \alpha_2\}$ pertenece a $\mathcal{C}$ porque es cerrada por subconjuntos.

Finalmente, como S' es un subconjunto finito cualquiera de S, tenemos el resultado.

e) Si $\beta \in S$, entonces $S \cup \{\beta_1\} \in \mathcal{C}'$ o $S \cup \{\beta_2\} \in \mathcal{C}'$.

Siguiendo nuevamente el razonamiento anterior, para probar que $S \cup \{\beta_1\} \in \mathcal{C}'$ o $S \cup \{\beta_2\} \in \mathcal{C}'$ hay que probar que todo subconjunto finito de $S \cup \{\beta_1\} \in \mathcal{C}$ o todo subconjunto finito de $S \cup \{\beta_2\} \in \mathcal{C}$.

Sea S' un subconjunto finito cualquiera de S. Como $\beta \in S$ y $S \in \mathcal{C}'$, entonces $S' \cup \{\beta\} \in \mathcal{C}$. Por ser $\mathcal{C}$ una clase de consistencia proposicional, $S' \cup \{\beta\} \cup \{\beta_1\} \in \mathcal{C}$ o $S' \cup \{\beta\} \cup \{\beta_2\} \in \mathcal{C}$. Entonces todo subconjunto de alguno de estos conjuntos pertenece a $\mathcal{C}$ porque es cerrada por subconjuntos.

Finalmente, como S' es un subconjunto finito cualquiera de S, tenemos el resultado. □

Probemos a continuación que una clase de consistencia proposicional de carácter finito es cerrada bajo límites.

Teorema 3.6.1. *Sea $\mathcal{C}$ una clase de consistencia proposicional de carácter finito, y $S_1, S_2, S_3, \ldots$ una sucesión de elementos de $\mathcal{C}$ tal que $S_1 \subseteq S_2 \subseteq S_3 \subseteq \ldots$. Entonces $\cup_i S_i$ es un elemento de $\mathcal{C}$.*

Demostración. Como $\mathcal{C}$ es de carácter finito, para probar que $\cup_i S_i \in \mathcal{C}$, basta probar que cada subconjunto finito de $\cup_i S_i$ está en $\mathcal{C}$. Es decir, vamos a probar que si $\{A_1, \ldots, A_k\} \subseteq \cup_i S_i$, entonces $\{A_1, \ldots, A_k\} \in \mathcal{C}$. Sabemos que para cada $i = 1, \ldots, k$, $A_i \in S_{n_i}$, donde n_i es el entero más pequeño para el que se verifica esto. Sea $N = \text{máx}\{n_1, \ldots, n_k\}$. Es fácil ver que cada $A_i \in S_N$. Pero $S_N \in \mathcal{C}$, y $\mathcal{C}$ es cerrada por subconjuntos, ya que es de carácter finito, por lo que $\{A_1, \ldots, A_k\} \in \mathcal{C}$. □

Teorema 3.6.2. ***(Existencia de Modelos Proposicional)*** *Si $\mathcal{C}$ es una clase de consistencia proposicional y $S \in \mathcal{C}$, entonces S es consistente.*

Demostración. La idea básica de la prueba es mostrar que cualquier elemento S de una colección de conjuntos $\mathcal{C}$ que posee una propiedad de consistencia proposicional se puede extender a otro elemento que es un conjunto de Hintikka, por lo que será consistente por el Lema de Hintikka.

Si S es finito, esto es fácil, pues podemos extender S siguiendo las condiciones 3, 4 y 5 de la definición anterior en un número finito de pasos. Es simple ver los detalles de este proceso, por lo que los obviaremos.

Si S es infinito las cosas no son tan fáciles. Podemos añadir las fórmulas que necesitamos para obtener un conjunto de Hintikka una por una, pero este proceso no terminará. Generaremos una sucesión infinita de elementos cada vez más grandes de la colección $\mathcal{C}$, y lo que queremos es el límite de esta sucesión (unión de cadenas). Hemos probado en el Teorema 3.6.1 que las clases de consistencia proposicionales de carácter finito son cerradas bajo límites. Como también hemos probado en las proposiciones 3.6.1 y 3.6.3 que toda clase de consistencia proposicional puede ser extendida a otra de carácter finito, supongamos que $\mathcal{C}$ es una clase de consistencia proposicional de carácter finito, y supongamos que $S \in \mathcal{C}$.

Como la lista de variables proposicionales es numerable, el conjunto de fórmulas proposicionales es numerable también, por un resultado de teoría de conjuntos que no probaremos. Sea $F_1, F_2, F_3, \ldots$ una enumeración de todas las fórmulas proposicionales en algún orden fijado. Definimos una sucesión $S_0, S_1, S_2, \ldots$ de elementos de $\mathcal{C}$ como sigue:

$$\begin{aligned} S_0 &= S \\ S_{n+1} &= \begin{cases} S_n \cup \{F_n\} & \text{si } S_n \cup \{F_n\} \in \mathcal{C} \\ S_n & \text{en otro caso} \end{cases} \end{aligned}$$

Entonces todo $S_n \in \mathcal{C}$, y también S_n es un subconjunto de S_{n+1}. Finalmente, sea $\mathbf{H} = S_0 \cup S_1 \cup S_2 \cup \ldots$. Trivialmente $\mathbf{H}$ extiende a S. Como $\mathcal{C}$ es de carácter finito, hemos probado que es cerrada bajo unión de cadenas, por lo que $\mathbf{H} \in \mathcal{C}$.

Probemos ahora que $\mathbf{H}$ es maximal en $\mathcal{C}$; esto es, si $\mathbf{H} \subseteq K$ para algún $K \in \mathcal{C}$, entonces $\mathbf{H} = K$. Hagámoslo por reducción al absurdo. Supongamos que $\mathbf{H} \subsetneq K$, donde $K \in \mathcal{C}$. Entonces se tiene que existe alguna fórmula proposicional F_n que pertenece a K pero no pertenece a $\mathbf{H}$. Como $F_n \notin \mathbf{H}$, $F_n \notin S_{n+1}$, lo que implica que $S_n \cup \{F_n\} \notin \mathcal{C}$. Pero $S_n \cup \{F_n\} \subseteq K$, ya que $S_n \subseteq \mathbf{H}$, $\mathbf{H} \subseteq K$ y $F_n \in K$. Como $\mathcal{C}$ es cerrado por subconjuntos, $S_n \cup \{F_n\} \in \mathcal{C}$, y tenemos una contradicción.

Probemos por último que $\mathbf{H}$ es un conjunto de Hintikka. Debemos probar todas las condiciones de la definición de conjunto de Hintikka. Las dos primeras se tienen por ser $\mathcal{C}$ una clase de consistencia proposicional. Las demás se prueban de manera similar, así que sólo lo haremos para las fórmulas α. Supongamos entonces que $\alpha \in \mathbf{H}$. Probaremos que $\alpha_1, \alpha_2 \in \mathbf{H}$. Como $\alpha \in \mathbf{H}$ y $\mathbf{H} \in \mathcal{C}$, $\mathbf{H} \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$. Pero este conjunto extiende a $\mathbf{H}$, que es maximal, por lo que debe ser idénticamente igual a $\mathbf{H}$, lo que significa que $\alpha_1, \alpha_2 \in \mathbf{H}$.

Por tanto, por el Lema de Hintikka, $\mathbf{H}$ es consistente, de lo que se concluye que también lo es S, ya que $S \subseteq \mathbf{H}$. □

3.7. Deducción natural

Los sistemas de deducción natural constituyen una familia de mecanismos de prueba, destinados a formalizar el tipo de razonamiento que las personas hacen en

argumentos informales. Se basan en la idea de pruebas subordinadas, en las que las conclusiones se derivan de las premisas, usando las conocidas como reglas de inferencia, pues nos permiten inferir fórmulas a partir de otras.

Representaremos cada prueba subordinada mediante una caja, donde la primera línea dentro de la caja sea la hipótesis particular hecha en dicha prueba subordinada, y la primera línea bajo la caja, la conclusión de la prueba.

Consideremos las siguientes reglas de inferencia:

- Constantes.

  Introducción de la verdad $\dfrac{}{\top}\ \top i$

  Eliminación de lo falso $\dfrac{\bot}{F}\ \bot e$

- Negación.

  Introducción de la negación $\dfrac{\boxed{\begin{array}{c}F\\ \vdots\\ \bot\end{array}}}{\neg F}\ \neg i$

  Eliminación de la negación $\dfrac{F \quad \neg F}{\bot}\ \neg e$

- Doble negación.

  Introducción de la doble negación $\dfrac{F}{\neg\neg F}\ \neg\neg i$

  Eliminación de la negación $\dfrac{\neg\neg F}{F}\ \neg\neg e$

- Conjunción.

  Introducción de la conjunción $\dfrac{F \quad G}{F \wedge G}\ \wedge i$

  Eliminación de la conjunción 1 $\dfrac{F \wedge G}{F}\ \wedge e_1$

  Eliminación de la conjunción 2 $\dfrac{F \wedge G}{G}\ \wedge e_2$

- Disyunción.

  Introducción de la disyunción 1 $\dfrac{F}{F \vee G}\ \vee i_1$

  Introducción de la disyunción 2 $\dfrac{G}{F \vee G}\ \vee i_2$

Eliminación de la disyunción $$\dfrac{F \vee G \quad \boxed{\begin{array}{c} F \\ \vdots \\ H \end{array}} \quad \boxed{\begin{array}{c} G \\ \vdots \\ H \end{array}}}{H} \vee e$$

- Implicación.

Introducción de la implicación $$\dfrac{\boxed{\begin{array}{c} F \\ \vdots \\ G \end{array}}}{F \Rightarrow G} \Rightarrow i$$

Eliminación de la implicación $$\dfrac{F \quad F \Rightarrow G}{G} \Rightarrow e$$

Definición 3.7.1. Una prueba por deducción natural es una sucesión finita $F_1, F_2, \ldots, F_n$ de fórmulas tal que cada término es un axioma o se sigue de términos anteriores por una de las reglas de inferencia.

De esta forma, podemos decir que cada prueba se construye por *etapas*, lo que nos permitiría ver qué fórmulas están involucradas en cada etapa.

Definición 3.7.2. Las fórmulas *activas* en una etapa son aquellas que ocurren en cajas que aún no se han cerrado en dicha etapa.

Definición 3.7.3. Diremos que la fórmula F deriva o es consecuencia del conjunto de fórmulas S por deducción natural si existe una prueba por este método en la que se puede usar cada término de S en cualquier etapa y el término final de la secuencia es F. Lo notaremos por $S \vdash F$.

Teorema 3.7.1. ***(Adecuación de la Deducción Natural)*** *Si $S \vdash F$, entonces $S \models F$.*

Demostración. Si $S \vdash F$, existe una prueba por deducción natural de F a partir del conjunto de fórmulas $S = \{G_1, \ldots, G_n\}$. Razonaremos por inducción completa en la longitud de la prueba. Es decir, queremos probar $M(k)$: 'Si existe una secuencia finita de k fórmulas $F_1, \ldots, F_k$ tal que cada término pertenece a S o se sigue de términos anteriores por una de las reglas de inferencia, acabando en F, entonces $S \models F$'

Caso base: $k = 1$. Entonces la prueba debe ser de la forma

1. G_1 hipótesis

ya que cualquier otra regla usaría más de una línea. Por tanto debe ser G_1 igual a F, es decir, $F \vdash F$. Entonces es trivial $F \models F$.

Probemos $M(k)$. Supongamos como hipótesis de inducción que se tiene $M(k')$, para cada $k' < k$. La prueba tiene la siguiente estructura:

1.	G_1	hipótesis
2.	G_2	hipótesis
	$\vdots$	
n.	G_n	hipótesis
	$\vdots$	
k.	F	conclusión

Debemos considerar todas las reglas que pudieron ser aplicadas en último lugar. Como se sigue el mismo argumento para cada una, consideraremos sólo las dos siguientes.

- Introducción de la conjunción. Entonces sabemos que F es de la forma $G \wedge H$, donde G y H son dos fórmulas de la sucesión $F_1, \ldots, F_{k-1}$. Supongamos que $G = F_{k_1}$ y $H = F_{k_2}$. Como k_1 y k_2 son enteros menores que k, existen pruebas por deducción natural $S \vdash G$ y $S \vdash H$, lo que implica, usando la hipótesis de inducción, que $S \models G$ y $S \models H$. Entonces toda interpretación cuyo valor sobre cada miembro de S es $\mathbf{t}$, también verifica que su valor sobre G y H es $\mathbf{t}$, por lo que se tiene $S \models F$.
- Eliminación de la disyunción. Entonces alguna de las fórmulas F_r, con $r < k$, es de la forma $G \vee H$. Por tanto tenemos $S \vdash G \vee H$, y por la hipótesis de inducción, $S \models G \vee H$. Además existen pruebas $S \cup \{G\} \vdash F$ y $S \cup \{H\} \vdash F$, con lo que se tiene $S \cup \{G\} \models F$ y $S \cup \{H\} \models F$. Si toda interpretación cuyo valor sobre S es $\mathbf{t}$ verifica también que su valor en $G \vee H$ es $\mathbf{t}$; y toda interpretación cuyo valor sobre $S \cup \{G\}$ o $S \cup \{H\}$ es $\mathbf{t}$, cumple también que su valor sobre F es $\mathbf{t}$, esto implica $S \models F$.

Finalmente hemos probado $S \models F$. □

Vamos a probar ahora la completitud de la deducción natural, es decir, si $S \models F$, entonces $S \vdash F$, para cualquier conjunto de fórmulas S y cualquier fórmula F.

Definición 3.7.4. Sea F una fórmula proposicional. Decimos que un conjunto S es *F-inconsistente por deducción natural* si $S \vdash F$; en otro caso diremos que S es *F-consistente por deducción natural.*

Lema 3.7.1. *Para cada fórmula F, la colección de todos los conjuntos F- consistentes por deducción natural es una clase de consistencia proposicional.*

Demostración. Debemos probar que se satisfacen las condiciones de la definición de clase de consistencia proposicional, para cada $S \in \mathcal{C}$, denotando por $\mathcal{C}$ la colección de todos los conjuntos F-consistentes por deducción natural. Por ser S F-consistente, no existe una prueba $S \vdash F$.

1. Para cualquier variable proposicional p, no pueden pertenecer a la vez p y $\neg p$ a S.

   Supongamos por reducción al absurdo que $p, \neg p \in S$. Entonces tendríamos la siguiente prueba:

1.	p	hipótesis
2.	$\neg p$	hipótesis
3.	$\bot$	$\neg e\ (1, 2)$
4.	F	$\bot\ e\ (3)$

   Por tanto tendríamos $S \vdash F$, que es una contradicción.

2. $\bot \notin S$; $\neg\top \notin S$.

   El primer caso se deduce de forma análoga a la prueba anterior. Para el caso de $\neg\top$, supongamos que pertenece a S. Tendríamos la siguiente prueba:

1.	$\neg\top$	hipótesis
2.	$\top$	hipótesis
3.	$\neg\top$	hipótesis
4.	$\bot$	$\neg e\ (3, 2)$
5.	$\neg\neg\top$	$\neg i\ (2 - 4)$
6.	$\bot$	$\neg e\ (1, 5)$
7.	F	$\bot\ e\ (6)$

   Por tanto tendríamos $S \vdash F$, que es una contradicción.

3. Si $\neg\neg G \in S$, entonces $S \cup \{G\} \in \mathcal{C}$.

   Supongamos por reducción al absurdo que $S \cup \{G\} \notin \mathcal{C}$, es decir, $S \cup \{G\}$ no es F-consistente por deducción natural. Entonces existe una prueba $S \cup \{G\} \vdash F$.

   Suponemos además que $\neg\neg G \in S$, y S es F-consistente, por lo que $S \nvdash F$, donde $S = \{F_1, \ldots, F_n, \neg\neg G\}$. Sin embargo, si a estos axiomas le aplicamos la regla de eliminación de la doble negación, que podemos hacerlo en cualquier etapa de una prueba, podemos considerar como axiomas $S \cup \{G\}$, pero hemos supuesto que $S \cup \{G\} \vdash F$, que es una contradicción. Por tanto, $S \cup \{G\} \in \mathcal{C}$.

4. Si $\alpha \in S$, entonces $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.

   Supongamos por reducción al absurdo que $S \cup \{\alpha_1, \alpha_2\} \notin \mathcal{C}$, es decir, $S \cup \{\alpha_1, \alpha_2\}$ no es F-consistente por deducción natural. Entonces existe una prueba $S \cup \{\alpha_1, \alpha_2\} \vdash F$.

Suponemos además que $\alpha \in S$, y S es F-consistente, por lo que $S \nvdash F$, donde $S = \{F_1, \ldots, F_n, \alpha\}$. Sin embargo, si a estos axiomas le aplicamos las dos reglas de eliminación de la conjunción, que podemos hacerlo en cualquier etapa de una prueba, podemos considerar como axiomas $S \cup \{\alpha_1, \alpha_2\}$, pero hemos supuesto que $S \cup \{\alpha_1, \alpha_2\} \vdash F$, que es una contradicción. Por tanto, $S \cup \{\alpha_1, \alpha_2\} \in \mathcal{C}$.

5. Si $\beta \in S$, entonces $S \cup \{\beta_1\} \in \mathcal{C}$ o $S \cup \{\beta_2\} \in \mathcal{C}$.

   Supongamos por reducción al absurdo que $S \cup \{\beta_1\}, S \cup \{\beta_2\} \notin \mathcal{C}$, es decir, ni $S \cup \{\beta_1\}$ ni $S \cup \{\beta_2\}$ son F-consistente por deducción natural. Entonces existen dos pruebas $S \cup \{\beta_1\} \vdash F$, $S \cup \{\beta_2\} \vdash F$.

   Suponemos además que $\beta \in S$, y S es F-consistente, por lo que $S \nvdash F$, donde $S = \{F_1, \ldots, F_n, \beta\}$. Sin embargo, si a estos axiomas le aplicamos la regla de eliminación de la disyunción, que podemos hacerlo en cualquier etapa de una prueba, obtenemos $S \vdash F$, que es una contradicción. Por tanto, $S \cup \{\beta_1\} \in \mathcal{C}$ o $S \cup \{\beta_2\} \in \mathcal{C}$.

Por tanto, la colección de todos los conjuntos F-consistentes por deducción natural es una clase de consistencia proposicional. □

Lema 3.7.2. *Si $S \cup \{\neg F\} \vdash F$, entonces $S \vdash F$.*

Demostración. Si $S \cup \{\neg F\} \vdash F$, entonces $S \vdash (\neg F \Rightarrow F)$. Pero $(\neg F \Rightarrow F)$ es lógicamente equivalente a F, por lo que tendríamos $S \vdash F$.

Teorema 3.7.2. ***(Completitud de la Deducción Natural)*** *Si $S \models F$, entonces $S \vdash F$.*

Demostración. Supongamos que $S \nvdash F$. Entonces S es F-consistente por deducción natural.

Por el Lema 3.7.2, como $S \nvdash F$, también se tiene $S \cup \{\neg F\} \nvdash F$. Así, $S \cup \{\neg F\}$ es F-consistente por deducción natural. Por el Lema 3.7.1 y el Teorema de existencia de modelos, $S \cup \{\neg F\}$ es consistente, de lo que se deduce $S \not\models F$ por la Proposición 3.2.3, que contradice las hipótesis del teorema.

Por tanto, si $S \models F$, entonces $S \vdash F$. □

Capítulo 4

Formalización en Isabelle

Hemos probado en el capítulo anterior los teoremas de Adecuación y Completitud de la Lógica Proposicional, estableciendo para ello la teoría sobre la lógica proposicional necesaria. El objetivo de este capítulo es la verificación formal en Isabelle de la adecuación y completitud de la deducción natural en lógica proposicional.

La formalización en Isabelle sigue parte del libro *First-Order Logic and Automated Theorem Proving* [4] de Melvin Fitting, que hemos descrito en el capítulo 3. No obstante, hay algunas diferencias que comentaremos en el desarrollo de este capítulo. Por ejemplo, no especificaremos en Isabelle la noción de notación uniforme y el esquema de inducción asociado, sino que usaremos para las pruebas el esquema de inducción sobre el tipo de datos "formula" generado automáticamente por Isabelle.

Comenzamos definiendo una teoría `TFM`, en la que sólo importamos `Main`.

```
theory TFM
imports Main
begin
```

4.1. Resultados preliminares

Probamos, en primer lugar, algunos resultados preliminares que serán necesarios en las pruebas siguientes.

```
theorem conjE': "P ∧ Q ⟹ (P ⟹ P') ⟹ (Q ⟹ Q') ⟹ P' ∧ Q'"
  by auto

theorem conjE'': "(∀x. P x ⟶ Q x ∧ R x) ⟹
  ((∀x. P x ⟶ Q x) ⟹ Q') ⟹ ((∀x. P x ⟶ R x) ⟹ R') ⟹
   Q' ∧ R'"
  by auto
```

4.2. Sintaxis de la lógica proposicional

Representamos el conjunto de las fórmulas proposicionales, mediante el tipo de dato "`formula`", definido de forma recursiva como sigue.

```
datatype formula =
    TT
  | FF
  | Var nat
  | Neg "formula"
  | And "formula" "formula"
  | Or "formula" "formula"
  | Impl "formula" "formula"
```

En esta definición:

- proporcionamos dos constantes para representar una tautología (`TT`) y falso (`FF`),
- las variables proposicionales se representan mediante (`Var n`), siendo `n` un número natural,
- proporcionamos constructores `Neg`, `And`, `Or` e `Impl` para representar las conectivas lógicas, y las aplicamos a un argumento de tipo "`formula`" en el caso de la negación, o a dos argumentos en los demás casos.

No es necesario que definamos un Principio de Inducción Estructural, como hacíamos en el Teorema 3.1.1, pues a partir de la definición Isabelle genera de forma automática dicho teorema.

Teorema (Principio de Inducción Estructural): Toda fórmula atómica tiene una propiedad **P** si:

- Paso base: $\top$, $\bot$ y las variables proposicionales poseen la propiedad **P**.
- Paso de inducción:
  - Si F tiene la propiedad **P** también la tiene $\neg F$.
  - Si F_1 y F_2 tienen la propiedad **P** también la tienen $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$ y $(F_1 \to F_2)$.

El siguiente es el teorema que genera Isabelle:

```
thm formula.induct

?P TT ⟹
?P FF ⟹
(⋀nat. ?P (Var nat)) ⟹
```

```
(⋀formula. ?P formula ⟹ ?P (Neg formula)) ⟹
(⋀formula1 formula2.
    ?P formula1 ⟹
    ?P formula2 ⟹ ?P (And formula1 formula2)) ⟹
(⋀formula1 formula2.
    ?P formula1 ⟹
    ?P formula2 ⟹ ?P (Or formula1 formula2)) ⟹
(⋀formula1 formula2.
    ?P formula1 ⟹
    ?P formula2 ⟹ ?P (Impl formula1 formula2)) ⟹
?P ?formula
```

4.3. Semántica de la lógica proposicional

Definíamos en la definición 3.1.2 una interpretación como una aplicación del conjunto de las fórmulas proposicionales al conjunto de los valores de verdad (verdadero y falso). Sin embargo, también probamos en las Proposiciones 3.2.1 y 3.2.2 que se puede considerar como una función del conjunto de las variables proposicionales en los valores de verdad.

Por ello representaremos las interpretaciones en Isabelle como funciones del tipo "`nat ⇒ bool`".

Mediante la función `interp` extendemos, de forma recursiva, el valor de una interpretación a todas las fórmulas proposicionales.

```
fun interp :: "(nat ⇒ bool) ⇒ formula ⇒ bool"
where
  "interp e TT           = True"
| "interp e FF           = False"
| "interp e (Var p)      = e p"
| "interp e (Neg F)      = (¬ (interp e F))"
| "interp e (And F1 F2)  = ((interp e F1) ∧ (interp e F2))"
| "interp e (Or F1 F2)   = ((interp e F1) ∨ (interp e F2))"
| "interp e (Impl F1 F2) = ((interp e F1) ⟶ (interp e F2))"
```

Definición: una fórmula proposicional F es satisfacible si existe alguna interpretación `e` tal que el valor de `interp e` sobre F es verdadero, en cuyo caso se dice que `e` es un modelo de F.

En Isabelle, la siguiente función `satisfacible` aplicada a una fórmula `F` verifica si dicha fórmula es satisfacible.

```
definition
  satisfacible :: "formula ⇒ bool" where
  "satisfacible F = (∃e. interp e F)"
```

Definición: un conjunto de fórmulas S es consistente si existe alguna interpretación e que es modelo de todas sus fórmulas.

En Isabelle, la siguiente función `consistente` aplicada a un conjunto `S` verifica si dicho conjunto es consistente.

```
definition
  consistente :: "formula list ⇒ bool" where
  "consistente S = (∃e. (list_all (interp e) S))"
```

Dada una interpretación `e`, un conjunto de fórmulas `S` y una fórmula `F`, la siguiente función "`model e S F`" evalúa si, en el caso de que `e` sea modelo de todas las fórmulas de `S`, también lo es de `F`.

Representamos "`model e S F`" como "`e,S ⊨ F`".

```
definition
  model :: "(nat ⇒ bool) ⇒ formula list ⇒ formula ⇒ bool"
    ("_,_ ⊨ _" [50,50] 50)  where
  "(e,S ⊨ F) = (list_all (interp e) S ⟶ interp e F)"
```

Definición: una fórmula proposicional F es consecuencia lógica de un conjunto S de fórmulas proposicionales si toda interpretación que es modelo de S también lo es de F.

A partir de la función `model` definimos el concepto de consecuencia lógica, de la siguiente forma:

```
definition
  consecuencia_logica :: "formula list ⇒ formula ⇒ bool" where
  "consecuencia_logica S F = (∀e. e,S ⊨ F)"
```

4.4. Propiedad de consistencia

En esta sección definiremos la propiedad de consistencia proposicional, que denominaremos $\mathcal{C}$. Identificaremos asimismo $\mathcal{C}$ con la colección de todos los conjuntos de fórmulas que tienen dicha propiedad, y la denominaremos clase de consistencia.

En el capítulo anterior, antes de definir las clases de consistencia, introducíamos la notación uniforme en la sección 3.4. De esta forma reducíamos el número de casos a probar en los teoremas. Sin embargo, esto no es necesario, por lo que no introduciremos la notación uniforme en Isabelle. En lo que sigue, usaremos el Principio de Inducción Estructural para las fórmulas proposicionales generado por Isabelle, en lugar de usar el Teorema 3.4.1.

Definimos a continuación las clases de consistencia.

Definición: se dice que una colección de conjuntos de fórmulas $\mathcal{C}$ es una clase de consistencia si verifica las siguientes propiedades para cada $S \in \mathcal{C}$:

1. para cada variable proposicional p, no pertenecen a la vez p y la negación de p a S;

2. $\bot$, $\neg\top \notin S$;

3. si $\neg\neg F \in S$, entonces $S \cup \{F\} \in \mathcal{C}$;

4. si $G \wedge H \in S$, entonces $S \cup \{G, H\} \in \mathcal{C}$;

5. si $\neg(G \vee H) \in S$, entonces $S \cup \{\neg G, \neg H\} \in \mathcal{C}$;

6. si $G \vee H \in S$, entonces $S \cup \{G\} \in \mathcal{C}$ o $S \cup \{H\} \in \mathcal{C}$;

7. si $\neg(G \wedge H) \in S$, entonces $S \cup \{\neg G\} \in \mathcal{C}$ o $S \cup \{\neg H\} \in \mathcal{C}$;

8. si $G \longrightarrow H \in S$, entonces $S \cup \{\neg G\} \in \mathcal{C}$ o $S \cup \{H\} \in \mathcal{C}$;

9. si $\neg(G \longrightarrow H) \in S$, entonces $S \cup \{G, \neg H\} \in \mathcal{C}$.

Para que una colección de conjuntos $\mathcal{C}$ sea de consistencia exigimos que se cumplan estas nueve propiedades. Notemos que en la definición 3.6.1 sólo introdujimos cinco propiedades, pues las formulábamos usando la notación uniforme.

La definición en Isabelle de las clases de consistencia se especifica directamente, detallando cada una de las propiedades.

```
definition
  consistencia :: "formula set set ⇒ bool" where
  "consistencia C = (∀S. S ∈ C ⟶
     (∀p. ¬ (Var p ∈ S ∧ Neg (Var p) ∈ S)) ∧
     FF ∉ S ∧ Neg TT ∉ S ∧
     (∀F. Neg (Neg F) ∈ S ⟶ S ∪ {F} ∈ C) ∧
     (∀G H. And G H ∈ S ⟶ S ∪ {G, H} ∈ C) ∧
     (∀G H. Neg (Or G H) ∈ S ⟶ S ∪ {Neg G, Neg H} ∈ C) ∧
     (∀G H. Or G H ∈ S ⟶ S ∪ {G} ∈ C ∨ S ∪ {H} ∈ C) ∧
     (∀G H. Neg (And G H) ∈ S ⟶
         S ∪ {Neg G} ∈ C ∨ S ∪ {Neg H} ∈ C) ∧
     (∀G H. Impl G H ∈ S ⟶ S ∪ {Neg G} ∈ C ∨ S ∪ {H} ∈ C) ∧
     (∀G H. Neg (Impl G H) ∈ S ⟶ S ∪ {G, Neg H} ∈ C))"
```

Notemos que en esta definición en Isabelle, `C` es un conjunto de conjuntos de fórmulas, como se especifica en el tipo del predicado `consistencia`.

4.4.1. Cerrada por subconjuntos

En esta sección definiremos las colección de conjuntos cerradas por subconjuntos, y probaremos que toda clase de consistencia se puede extender a otra cerrada por subconjuntos. Aunque en el capítulo anterior lo probamos en una única proposición, la Proposición 3.6.1, en Isabelle desglosaremos la prueba de esta proposición en varios teoremas.

Sea $\mathcal{C}$ una colección de conjuntos de fórmulas. Definimos en primer lugar la colección de todos los subconjuntos de los conjuntos de $\mathcal{C}$.

```
definition
  subconjuntos :: "formula set set ⇒ formula set set" where
  "subconjuntos C = {S. ∃S' ∈ C. S ⊆ S'}"
```

Definición: una colección de conjuntos es cerrada por subconjuntos si contiene, por cada elemento, todos los subconjuntos de ese elemento.

```
definition
  cerrado_subconj :: "'a set set ⇒ bool" where
  "cerrado_subconj C = (∀S' ∈ C. ∀S. S ⊆ S' ⟶ S ∈ C)"
```

Notemos que podemos definir los conceptos anteriores de esta forma en Isabelle porque posee una lógica de orden superior, por lo que podemos cuantificar conjuntos.

Queremos probar que toda clase de consistencia `C` puede extenderse a la colección (`subconjuntos C`). Probaremos en primer lugar el siguiente teorema.

Teorema: si $\mathcal{C}$ es una clase de consistencia, entonces la colección de los subconjuntos de todos los conjuntos de $\mathcal{C}$ es de consistencia también.

```
theorem subconj_consistencia:
   "consistencia C ⟹ consistencia (subconjuntos C)"
```

Demostración: la prueba consiste en comprobar las propiedades que enunciamos anteriormente en la definición de clase de consistencia para la colección de los subconjuntos de todos los conjuntos de `C`.

Las cinco primeras propiedades se prueban de forma automática. En cuanto a las demás, explicaremos el caso de la sexta propiedad:

> "si `G ∨ H ∈ S`, entonces `S ∪ {G} ∈ C` o `S ∪ {H} ∈ C`".

Debemos probar lo siguiente:

```
 "⋀S x G H.
       x ∈ C ⟹
       S ⊆ x ⟹
       Or G H ∈ S ⟹
       insert G x ∈ C ∨ insert H x ∈ C ⟹
       (∃x∈C. G ∈ x ∧ S ⊆ x) ∨ (∃x∈C. H ∈ x ∧ S ⊆ x)"
```

Aplicamos la regla de eliminación de la disyunción:

```
apply (erule disjE)
```

Obtenemos dos nuevos objetivos

```
 "1. ⋀S x G H.
       x ∈ C ⟹
       S ⊆ x ⟹
       Or G H ∈ S ⟹ insert G x ∈ C ⟹
       (∃x∈C. G ∈ x ∧ S ⊆ x) ∨ (∃x∈C. H ∈ x ∧ S ⊆ x)
  2. ⋀S x G H.
       x ∈ C ⟹
       S ⊆ x ⟹
       Or G H ∈ S ⟹ insert H x ∈ C ⟹
       (∃x∈C. G ∈ x ∧ S ⊆ x) ∨ (∃x∈C. H ∈ x ∧ S ⊆ x)"
```

Como ambos se prueban de forma análoga, centrémonos en el primero:

```
apply (rule disjI1)
apply (rule_tac x="insert G x" in bexI)
apply blast
apply assumption
```

Para probar la disyunción demostramos el primer miembro, eliminando el cuantificador existencial. El resto de la prueba de este objetivo es automática. □

Incluimos la prueba completa en Isabelle.

```
theorem subconj_consistencia: "consistencia C ⟹
                       consistencia (subconjuntos C)"
  apply (simp add: subconjuntos_def consistencia_def)
  apply (rule allI)
  apply (rule impI)
  apply (erule bexE)
  apply (erule allE impE)+
  apply assumption
  apply (erule conjE', blast)
  apply (erule conjE', blast)
  apply (erule conjE', blast)
  apply (erule conjE', blast)
  apply (erule conjE', blast)
  apply (erule conjE', blast)
  apply (erule conjE')+
  apply (rule allI impI)+
  apply (erule allE impE)+
  apply (erule subsetD, assumption)
```

```
apply (erule disjE)
apply (rule disjI1)
apply (rule_tac x="insert G x" in bexI)
apply blast
apply assumption
apply (rule disjI2)
apply (rule_tac x="insert H x" in bexI)
apply blast
apply assumption
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (erule subsetD)
apply assumption
apply (erule disjE)
apply (rule disjI1)
apply (rule_tac x="insert (Neg G) x" in bexI)
apply blast
apply assumption
apply (rule disjI2)
apply (rule_tac x="insert (Neg H) x" in bexI)
apply blast
apply assumption
apply (erule conjE')
apply (rule allI impI)+
apply (erule allE impE)+
apply (erule subsetD)
apply assumption
apply (erule disjE)
apply (rule disjI1)
apply (rule_tac x="insert (Neg G) x" in bexI)
apply blast
apply assumption
apply (rule disjI2)
apply (rule_tac x="insert H x" in bexI)
apply blast
apply assumption
apply (rule allI impI)+
apply (erule allE impE)+
apply (erule subsetD)
apply assumption
apply blast
done
```

□

Se prueba de manera automática que la colección de los subconjuntos de todos los conjuntos de una colección de conjuntos $\mathcal{C}$ es cerrada por subconjuntos.

```
lemma subconjuntos1: "cerrado_subconj (subconjuntos C)"
  by (unfold subconjuntos_def cerrado_subconj_def) blast
```
□

Se prueba asimismo de manera automática que la colección de conjuntos $\mathcal{C}$ está contenida en la colección de todos los subconjuntos de los conjuntos de $\mathcal{C}$.

```
lemma subconjuntos2: "C ⊆ subconjuntos C"
  by (unfold subconjuntos_def) blast
```
□

De esta forma hemos probado que toda clase de consistencia `C` puede extenderse a otra cerrada por subconjuntos, que hemos definido en Isabelle como `subconjuntos C`.

4.4.2. Carácter finito

En esta sección definiremos las colecciones de conjuntos de carácter finito, y probaremos que toda clase de consistencia cerrada por subconjuntos se puede extender a otra de carácter finito. Desglosaremos en Isabelle la prueba de la Proposición 3.6.3 en varios teoremas.

Definición: una colección de conjuntos $\mathcal{C}$ se dice de carácter finito si S pertenece a $\mathcal{C}$ si, y sólo si, todo subconjunto finito de S pertenece a $\mathcal{C}$.

```
definition
  caracter_finito :: "'a set set ⇒ bool" where
  "caracter_finito C =
        (∀S. S ∈ C = (∀S'. finite S' ⟶ S' ⊆ S ⟶ S' ∈ C))"
```

Probaremos en primer lugar el siguiente teorema.

Teorema: si una colección de conjuntos $\mathcal{C}$ es de carácter finito, entonces $\mathcal{C}$ es cerrada por subconjuntos.

```
theorem carfin_cerrsub: "caracter_finito C ⟹ cerrado_subconj C"
  apply (unfold caracter_finito_def cerrado_subconj_def)
  apply (rule ballI)
  apply (rule allI)
  apply (rule impI)
  apply (frule spec)
  apply (erule iffD2)
  apply blast
  done
```
□

En la prueba anterior hemos detallado cada una de las reglas que se aplican. Ahora bien, también se puede probar de forma automática mediante

```
by (metis caracter_finito_def cerrado_subconj_def order.trans)
```
□

Si `C` es una colección de conjuntos, definimos a continuación una extensión de dicha colección a otra que será de carácter finito, de la forma siguiente. La extensión (`mk_car_finito C`) está formada por los conjuntos S tales que cualquier subconjunto suyo S', finito, pertenece a `C`.

```
definition
  mk_car_finito :: "'a set set ⇒ 'a set set" where
  "mk_car_finito C = {S. ∀S'. S' ⊆ S ⟶ finite S' ⟶ S' ∈ C}"
```

Teorema: toda clase de consistencia proposicional `C` cerrada por subconjuntos puede extenderse a otra de carácter finito, más concretamente, a la extensión definida anteriormente (`mk_car_finito C`).

```
theorem consistencia_car_fin: "consistencia C ⟹
                               cerrado_subconj C ⟹
                               consistencia (mk_car_finito C)"
```

Demostración: la prueba consiste en demostrar que (`mk_car_finito C`) verifica cada una de las propiedades que definen a una clase de consistencia proposicional.

Las dos primeras propiedades de clase de consistencia se prueban por reducción al absurdo de manera casi automática, sin más que indicar, para cada `S ∈ (mk_car_finito C)`, qué subconjunto finito `S'` de `S` lleva a contradicción.

En cuanto a las demás propiedades, como se sigue el mismo esquema de demostración para cada una, tomaremos una de ellas y estudiaremos su demostración. Veamos por ejemplo la tercera propiedad,

> "para todo `S ∈ (mk_car_finito C)`, si `¬¬ F ∈ S`, entonces `S ∪ {F} ∈ (mk_car_finito C)`".

Hay que probar que todo conjunto finito de `S` pertenece a `C`:

```
"⋀S.
  ∀S'∈C. ∀S⊆S'. S ∈ C ⟹
  ∀S'⊆S. finite S' ⟶ S' ∈ C ⟹
  ∀S. S ∈ C ⟶ (∀F. Neg (Neg F) ∈ S ⟶ S ∪ {F} ∈ C) ⟹
  ∀F. Neg (Neg F) ∈ S ⟶ S ∪ {F} ∈
    {S. ∀S'⊆S. finite S' ⟶ S' ∈ C}"
```

Probemos que el conjunto `S' - {F} ∪ {¬¬ F}` pertenece a `C` usando la hipótesis

"∀ S' ⊆ S. finite S' ⟶ S' ∈ $\mathcal{C}$":

```
apply (rule allI impI CollectI)+
apply (erule_tac x = "S' - {F} ∪ {Neg (Neg F)}" in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
```

A continuación probamos que el conjunto S' - {F}∪ {Neg (Neg F)} ∪{F} pertenece a C usando la hipótesis

"∀S. S ∈ C ⟶ (∀F. Neg (Neg F) ∈ S ⟶ S ∪ {F} ∈ C)":

```
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = F in allE)
apply (erule impE)
apply blast
```

Probamos por último que el conjunto S' pertenece a C usando la hipótesis

"∀S'∈C. ∀S⊆S'. S ∈ C":

```
apply (drule_tac x = "S' - {F} ∪ {Neg (Neg F)} ∪ {F}" in bspec)
apply assumption
apply blast
```

□

Incluimos la prueba completa en Isabelle.

```
theorem consistencia_car_fin: "consistencia C ⟹
                               cerrado_subconj C ⟹
                               consistencia (mk_car_finito C)"
  apply (unfold mk_car_finito_def cerrado_subconj_def
         consistencia_def)
  apply (rule allI impI)+
  apply (erule CollectE)
  apply (erule conjE'')
  apply (rule allI notI)+
  apply (erule_tac x = "{Var p, Neg (Var p)}" in allE)
  apply blast
  apply (erule conjE'')
  apply (erule_tac x = "{FF}" in allE)
  apply blast
  apply (erule conjE'')
```

```
apply (erule_tac x = "{Neg TT}" in allE)
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (erule_tac x = "S' - {F} ∪ {Neg (Neg F)}" in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = F in allE)
apply (erule impE)
apply blast
apply (drule_tac x = "S' - {F} ∪ {Neg (Neg F)} ∪ {F}" in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (erule_tac x = "S' - {G, H} ∪ {And G H}" in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = G in allE)
apply (erule_tac x = H in allE)
apply (erule impE)
apply blast
apply (drule_tac x = "S' - {G, H} ∪ {And G H} ∪ {G, H}" in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI CollectI)+
apply (erule_tac x = "S' - {Neg G, Neg H} ∪ {Neg (Or G H)}"
       in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = G in allE)
```

```
apply (erule_tac x = H in allE)
apply (erule impE)
apply blast
apply (drule_tac x = "S' - {Neg G, Neg H} ∪ {Neg (Or G H)}
                      ∪ {Neg G, Neg H}" in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI)+
apply (rule ccontr)
apply (simp (no_asm_use))
apply (erule conjE exE)+
apply (erule_tac x = "(S' - {G}) ∪ (S'a - {H})
                      ∪ {Or G H}" in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = G in allE)
apply (erule_tac x = H in allE)
apply (erule impE)
apply blast
apply (erule disjE)
apply (drule_tac x = "insert G (S' - {G} ∪ (S'a - {H})
                      ∪ {Or G H})" in bspec)
apply assumption
apply blast
apply (drule_tac x="insert H (S' - {G} ∪ (S'a - {H})
                    ∪ {Or G H})" in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI)+
apply (rule ccontr)
apply (simp (no_asm_use))
apply (erule conjE exE)+
apply (erule_tac x = "(S' - {Neg G}) ∪ (S'a - {Neg H})
                      ∪ {Neg (And G H)}" in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
```

```
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = G in allE)
apply (erule_tac x = H in allE)
apply (erule impE)
apply blast
apply (erule disjE)
apply (drule_tac x = "insert (Neg G)
    (S' - {Neg G} ∪ (S'a - {Neg H}) ∪ {Neg (And G H)})" in bspec)
apply assumption
apply blast
apply (drule_tac x = "insert (Neg H)
    (S' - {Neg G} ∪ (S'a - {Neg H}) ∪ {Neg (And G H)})" in bspec)
apply assumption
apply blast
apply (erule conjE'')
apply (rule allI impI)+
apply (rule ccontr)
apply (simp (no_asm_use))
apply (erule conjE exE)+
apply (erule_tac x = "(S' - {Neg G}) ∪ (S'a - {H})
                        ∪ {Impl G H}" in allE)
apply (erule impE)
apply blast
apply (erule impE)
apply blast
apply (erule allE, erule impE)
apply assumption
apply (erule_tac x = G in allE)
apply (erule_tac x = H in allE)
apply (erule impE)
apply blast
apply (erule disjE)
apply (drule_tac x = "insert (Neg G)
    (S' - {Neg G} ∪ (S'a - {H}) ∪ {Impl G H})" in bspec)
apply assumption
apply blast
apply (drule_tac x = "insert H
    (S' - {Neg G} ∪ (S'a - {H}) ∪ {Impl G H})" in bspec)
apply assumption
apply blast
apply (rule allI impI CollectI)+
apply (erule_tac x = "S' - {G, Neg H} ∪ {Neg (Impl G H)}"
       in allE)
```

```
  apply (erule impE)
  apply blast
  apply (erule impE)
  apply blast
  apply (erule allE, erule impE)
  apply assumption
  apply (erule_tac x = G in allE)
  apply (erule_tac x = H in allE)
  apply (erule impE)
  apply blast
  apply (drule_tac x = "S' - {G, Neg H} ∪ {Neg (Impl G H)}
                        ∪ {G, Neg H}" in bspec)
  apply assumption
  apply blast
  done
```
□

Se prueba de manera automática que la colección de conjuntos (`mk_car_finito C`) es de carácter finito, como adelantábamos.

```
lemma car_finito: "caracter_finito (mk_car_finito C)"
  by (unfold caracter_finito_def mk_car_finito_def) blast
```
□

Se prueba asimismo de manera automática que si `C` es una colección de conjuntos cerrada por subconjuntos, entonces está contenida en la extensión de dicha colección a (`mk_car_finito C`).

```
lemma ext_car_fin: "cerrado_subconj C ⟹ C ⊆ mk_car_finito C"
  by (unfold mk_car_finito_def cerrado_subconj_def) blast
```
□

Concluimos que toda clase de consistencia proposicional cerrada por subconjuntos puede extenderse a otra de carácter finito. Como habíamos probado que toda clase de consistencia proposicional puede extenderse a una cerrada por subconjuntos, de hecho se tiene que toda clase de consistencia proposicional puede extenderse a otra de carácter finito. Hemos definido en Isabelle dicha colección de conjuntos como (`mk_car_finito (subconjuntos C)`), dada `C` una clase de consistencia proposicional.

4.4.3. Enumeración de fórmulas

Las variables proposicionales `Var n` se construyen a partir de los naturales, por lo que el conjunto de las variables proposicionales es un conjunto numerable. Por tanto, el conjunto de las fórmulas proposicionales es también un conjunto numerable. En esta sección construiremos una enumeración de las fórmulas proposicionales, aunque no entraremos en detalles al respecto.

Definimos a continuación la función diagonal, que recorre todos los pares de números naturales.

$$\begin{aligned} \text{diag } (0) &= (0,0) \\ \text{diag } (n+1) &= \begin{cases} (0, x+1) & \text{si } y = 0 \\ (x+1, y-1) & \text{en otro caso} \end{cases} \end{aligned}$$

donde $(x, y) = \text{diag } (n)$. Este método para enumerar pares de números naturales se debe a Cantor.

Se representa en Isabelle de la siguiente forma:

```
fun
  diag :: "nat ⇒ (nat × nat)"
where
  "diag 0 = (0, 0)"
| "diag (Suc n) =
     (let (x, y) = diag n
      in case y of
          0 ⇒ (0, Suc x)
        | Suc y ⇒ (Suc x, y))"
```

Enunciamos los siguientes teoremas sobre la función diagonal.

Teorema: para todo $n \geq 1$, si diag $(n) = (a, b)$, entonces $a < n$.

```
theorem diag_le1: "fst (diag (Suc n)) < Suc n"
  by (induct n)
     (simp_all add: Let_def split_def split add: nat.split)
```

□

Teorema: para todo $n \geq 2$, si diag $(n) = (a, b)$, entonces $b < n$.

```
theorem diag_le2: "snd (diag (Suc (Suc n))) < Suc (Suc n)"
  apply (induct n)
  apply (simp_all add: Let_def split_def split add: nat.split
                       nat.split_asm)
  apply (rule impI)
  apply (case_tac n)
  apply simp
  apply hypsubst
  apply (rule diag_le1)
  done
```

□

Teorema: sea n un natural, y diag $(n) = (a, b)$. Si $a \geq 1$, entonces $b < n$.

```
theorem diag_le3: "fst (diag n) = Suc x ⟹ snd (diag n) < n"
  apply (case_tac n)
  apply simp
  apply (case_tac nat)
  apply (simp add: Let_def)
  apply hypsubst
  apply (rule diag_le2)
  done
```

□

Teorema: sea n un natural, y diag $(n) = (a, b)$. Si $a \geq 1$, entonces $a < n + 1$.

```
theorem diag_le4: "fst (diag n) = Suc x ⟹ x < n"
  apply (case_tac n)
  apply simp
  apply (case_tac nat)
  apply (simp add: Let_def)
  apply hypsubst_thin
  apply (drule sym)
  apply (drule ord_eq_less_trans)
  apply (rule diag_le1)
  apply simp
  done
```

□

Definimos también la inversa de la función diagonal.

$$\begin{aligned} \text{undiag } (0,0) &= 0 \\ \text{undiag } (x,y) &= \begin{cases} 1 + \text{undiag } (y-1,0) & \text{si } x = 0 \\ 1 + \text{undiag } (x-1,y+1) & \text{en otro caso} \end{cases} \end{aligned}$$

Cabe destacar que al ser una función recursiva, debe terminar. Debemos definir en Isabelle una medida que decrezca en cada paso recursivo, aunque se prueba automáticamente que es así.

```
function
  undiag :: "nat × nat ⇒ nat"
where
  "undiag (0, 0) = 0"
| "undiag (0, Suc y) = Suc (undiag (y, 0))"
| "undiag (Suc x, y) = Suc (undiag (x, Suc y))"
  by pat_completeness auto

termination
  by (relation
      "measure (λ(x, y). ((x + y) * (x + y + 1)) div 2 + x)") auto
```

Teorema: las funciones `undiag` y `diag` son inversa la una de la otra.

```
theorem diag_undiag [simp]: "diag (undiag (x, y)) = (x, y)"
  by (rule undiag.induct) auto

theorem undiag_diag[simp]: "undiag (diag n) = n"
  apply (induct n)
  apply (simp_all add: Let_def split_def split add: nat.split)
  apply (metis prod.collapse)+
  done
```
□

Notemos que hemos añadido los teoremas anteriores a `simp`.

Definimos un tipo de dato recursivo `btree` para representar los árboles binarios con etiquetas en las hojas. Un árbol `btree` es una hoja"`Leaf n`" o se construye con el constructor "`Branch`" a partir de dos árboles `btree`.

```
datatype btree = Leaf nat | Branch btree btree
```

La siguiente función `diag_btree` asocia a cada natural un elemento del tipo `btree`, usando para ello la función diagonal. Cabe destacar que también hay que probar que esta función termina, lo cual se hace de manera automática usando los lemas `diag_le3` y `diag_le4`.

$$\text{diag\_btree}\ (n) = \begin{cases} \texttt{Leaf}\ n & \text{si } a = 0 \\ \texttt{Branch}\ (\text{diag\_btree}\ (a-1))\ (\text{diag\_btree}\ (b)) & \text{e.o.c.} \end{cases}$$

donde $(a, b) = \text{diag}\ (n)$.

Definimos la función en Isabelle como sigue:

```
function
  diag_btree :: "nat ⇒ btree"
where
  "diag_btree n = (case fst (diag n) of
       0 ⇒ Leaf (snd (diag n))
     | Suc x ⇒ Branch (diag_btree x)
                          (diag_btree (snd (diag n))))"
  by auto

termination
  by (relation "measure (λx. x)") (auto intro: diag_le3 diag_le4)
```

Definimos a continuación la función inversa de `diag_btree`, que asocia a cada elemento del tipo `btree` un natural, usando la inversa de la función diagonal.

$$\begin{aligned} \text{undiag\_btree}\ (\texttt{Leaf}\ n) &= \text{undiag}\ (0, n) \\ \text{undiag\_btree}\ (\texttt{Branch}\ t_1\ t_2) &= \text{undiag}\ (1 + \text{undiag\_btree}\ t_1, \text{undiag\_btree}\ t_2) \end{aligned}$$

Lo definimos en Isabelle como

```
fun
  undiag_btree :: "btree ⇒ nat"
where
  "undiag_btree (Leaf n) = undiag (0, n)"
| "undiag_btree (Branch t1 t2) =
     undiag (Suc (undiag_btree t1), undiag_btree t2)"
```

Teorema: las funciones `undiag_btree` y `diag_btree` son inversa la una de la otra.

```
theorem diag_undiag_btree [simp]: "diag_btree (undiag_btree t) = t"
  by (induct t) auto

theorem undiag_diag_btree [simp]: "undiag_btree (diag_btree n) = n"
  apply (induct rule: diag_btree.induct)
  apply (simp_all add: Let_def split_def split add: nat.split)
  apply (metis prod.collapse undiag.simps(3) undiag_diag)
  done
```

□

Mediante el atributo `[simp]` añadimos los teoremas `diag_undiag_btree` y `undiag_diag_btree` a las reglas de eliminación, mientras que con el comando siguiente eliminamos `diag_btree.simps` y `undiag_btree.simps` de las reglas de simplificación.

```
declare diag_btree.simps [simp del] undiag_btree.simps [simp del]
```

Vamos a definir a continuación una correspondencia entre árboles `btree` y fórmulas. Vamos a definir para ello dos funciones `form_of_btree` y `btree_of_form`, de forma que cada árbol `btree` se represente por una única fórmula, y viceversa, respectivamente.

- Los árboles que sólo tienen una hoja se representan por las fórmulas $\bot$ y $\top$, si las hojas se aplican al natural 0 o al 1, respectivamente.

- Los árboles `Branch (Leaf 0) (Leaf n)` se representan por la variable proposicional `Var n`.

- Los árboles `Branch (Leaf m) (Branch t1 t2)`, para `m` $= 1, 2, 3$, se representan respectivamente por las fórmulas `f1` $\wedge$ `f2`, `f1` $\vee$ `f2` y `f1` $\longrightarrow$ `f2`, donde `f1` y `f2` es el resultado de aplicar recursivamente la función `form_of_btree` a los árboles `t1` y `t2`.

- Los árboles `Branch (Leaf 4) t` se representan por la negación de la fórmula resultante de aplicar recursivamente la función `form_of_btree` al árbol `t`.

Así, en los árboles que usan el constructor `Branch`, el árbol izquierdo de `Branch` indica qué tipo de fórmula representa: es una variable proposicional si el árbol izquierdo es `Leaf 0`, una conjunción si es `Leaf 1`, una disyunción si es `Leaf 2`, una implicación si es `Leaf 3` y una negación si es `Leaf 4`. El árbol derecho representa a qué fórmulas se aplican las conectivas lógicas anteriores.

Cabe notar que no todos los árboles se representan por una fórmula, pero todas las fórmulas sí se representan por un árbol.

Definamos en Isabelle la función `form_of_btree`.

```
fun
  form_of_btree :: "btree ⇒ formula"
where
  "form_of_btree (Leaf 0) = FF"
| "form_of_btree (Leaf (Suc 0)) = TT"
| "form_of_btree (Branch (Leaf 0) (Leaf n)) = Var n"
| "form_of_btree (Branch (Leaf (Suc 0)) (Branch t1 t2)) =
     And (form_of_btree t1) (form_of_btree t2)"
| "form_of_btree (Branch (Leaf (Suc (Suc 0))) (Branch t1 t2)) =
     Or (form_of_btree t1) (form_of_btree t2)"
| "form_of_btree (Branch (Leaf (Suc (Suc (Suc 0)))) (Branch t1 t2)) =
     Impl (form_of_btree t1) (form_of_btree t2)"
| "form_of_btree (Branch (Leaf (Suc (Suc (Suc (Suc 0))))) t) =
     Neg (form_of_btree t)"
```

Definamos también la función recíproca `btree_of_form`.

```
fun
  btree_of_form :: "formula ⇒ btree"
where
  "btree_of_form FF = Leaf 0"
| "btree_of_form TT = Leaf (Suc 0)"
| "btree_of_form (Var n) = (Branch (Leaf 0) (Leaf n))"
| "btree_of_form (And a b) = Branch (Leaf (Suc 0))
     (Branch (btree_of_form a) (btree_of_form b))"
| "btree_of_form (Or a b) = Branch (Leaf (Suc (Suc 0)))
     (Branch (btree_of_form a) (btree_of_form b))"
| "btree_of_form (Impl a b) = Branch (Leaf (Suc (Suc (Suc 0))))
     (Branch (btree_of_form a) (btree_of_form b))"
| "btree_of_form (Neg a) = Branch (Leaf (Suc (Suc (Suc (Suc 0)))))
     (btree_of_form a)"
```

La siguiente función `diag_form` asocia a un natural `n` un árbol mediante la función `diag_btree`, y a este árbol una fórmula, con la función `form_of_btree`, siempre que dicha fórmula exista para este árbol.

```
definition
  diag_form :: "nat ⇒ formula" where
  "diag_form n = form_of_btree (diag_btree n)"
```

Recíprocamente, la función `undiag_form` asocia a cada fórmula `F` un árbol, haciendo uso de la función `btree_of_form`, y a este árbol un natural, mediante la función `undiag_btree`.

```
definition
  undiag_form :: "formula ⇒ nat" where
  "undiag_form F = undiag_btree (btree_of_form F)"
```

Mediante estas dos funciones hemos definido una enumeración de las fórmulas proposicionales. Así, `undiag_form F` es el número de orden de la fórmula `F` en la enumeración construida. Además, aunque `diag_form n` no representa una fórmula proposicional para cada `n`, pues existen árboles que no tienen asociada ninguna fórmula, como se ha comentado previamente, esta función es sobreyectiva.

Teorema: la función `undiag_form` es la inversa por la derecha de `diag_form`.

```
theorem diag_undiag_form [simp]: "diag_form (undiag_form f) = f"
  by (induct f) (simp_all add: diag_form_def undiag_form_def)      □
```

4.4.4. Extensión a conjuntos consistentes maximales

Dada una colección de conjuntos $\mathcal{C}$ de carácter finito, vamos a probar que la unión de cualquier sucesión creciente de elementos de $\mathcal{C}$ está contenida en $\mathcal{C}$.

Se dice que $\mathcal{A} = \{A_n\}$ es una cadena o sucesión creciente de conjuntos si $\forall n\ A_n \subseteq A_{n+1}$.

Representaremos la cadena $\mathcal{A}$ por $f : \mathbb{N} \to$ 'a set, donde "'a set" representa el tipo de los conjuntos con elementos en a, luego representaremos A_n por $(f\ n)$.

La siguiente función "`es_cadena`" aplicada a f comprueba si f es una cadena creciente de conjuntos.

```
definition
  es_cadena :: "(nat ⇒ 'a set) ⇒ bool" where
  "es_cadena f = (∀n. f n ⊆ f (Suc n))"
```

Lema: sea f una cadena creciente de conjuntos, m un natural y $x \in f(m)$. Entonces $x \in f(m+n)$, para cualquier natural n.

Se prueba de manera automática.

```
lemma es_cadenaD: "es_cadena f ⟹ x ∈ f m ⟹ x ∈ f (m + n)"
  by (induct n) (auto simp add: es_cadena_def)                     □
```

Corolario: sea f una cadena creciente de conjuntos, m un natural y $x \in f(m)$. Entonces para todo natural $k \geq m$ se tiene $x \in f(k)$.

```
lemma es_cadenaD': "es_cadena f ⟹ x ∈ f m ⟹ m ≤ k ⟹ x ∈ f k"
  apply (subgoal_tac "∃n. k = m + n")
  apply (erule exE)
  apply simp
  apply (erule_tac n = "n" in es_cadenaD)
  apply assumption
  apply arith
  done
```
□

En la prueba anterior hemos detallado cada una de las reglas que se aplican. Ahora bien, también se puede probar de forma automática mediante

```
by (metis es_cadenaD le_Suc_ex)
```
□

Teorema: sea f una cadena creciente de conjuntos y F un conjunto finito. Si $F \subseteq \bigcup_n f(n)$, entonces existe un natural n tal que $F \subseteq f(n)$.

```
theorem cadena_index:
  assumes ch: "es_cadena f" and fin: "finite F"
  shows "F ⊆ (⋃n. f n) ⟹ ∃n. F ⊆ f n" using fin
```

Demostración: realizamos la prueba usando el esquema de inducción sobre conjuntos finitos. Si `P` es una propiedad sobre conjuntos, el esquema de inducción es el siguiente:

- Paso base. El conjunto vacío tiene la propiedad `P`.
- Paso de inducción. Si `F` es un conjunto finito con la propiedad `P`, entonces el conjunto resultante de añadir un elemento a `F` también tiene la propiedad `P`.

Entonces todo conjunto finito tiene la propiedad `P`.

El esquema inductivo para conjuntos finitos en Isabelle se enuncia como sigue:

```
thm finite_induct

finite ?F  ⟹
?P {} ⟹
(⋀x F. finite F ⟹ x ∉ F  ⟹ ?P F ⟹ ?P (insert x F)) ⟹ ?P ?F
```

Para comenzar la prueba aplicamos dicho esquema de inducción.

```
  apply (induct rule: finite_induct)
```

El caso base se prueba de forma automática.

```
apply blast
```

Sea un nuevo conjunto resultado de añadir un elemento `x` al conjunto `F`. Hay que probar que si `(insert x F)` $\subseteq$ `⋃n. f(n)`, entonces existe un natural `n` tal que `(insert x F)` $\subseteq$ `f(n)`. Por la hipótesis de inducción deben existir dos naturales `n` y `xa` tal que `x` $\in$ `f(xa)` y `F` $\subseteq$ `f(n)`.

```
apply (insert ch)
apply simp
apply (erule exE)
apply (erule conjE)
apply (erule exE)
```

Tomando el máximo entre `xa` y `n`, y aplicando los lemas anteriores, tenemos el resultado.

```
apply (rule_tac k = "max n xa" and m = "xa" in es_cadenaD',
       simp_all)
apply (rule subsetI)
apply (drule_tac B = "f n" and c = "xb" in subsetD)
apply assumption
apply (erule_tac k = "max n xa" and m = "n" in es_cadenaD')
apply (simp_all add: max_def)
done
```

□

Incluimos la prueba completa en Isabelle.

```
theorem cadena_index:
  assumes ch: "es_cadena f" and fin: "finite F"
  shows "F ⊆ (⋃n. f n) ⟹ ∃n. F ⊆ f n" using fin
  apply (induct rule: finite_induct)
  apply blast
  apply (insert ch)
  apply simp
  apply (erule exE)
  apply (erule conjE)
  apply (erule exE)
  apply (rule_tac x = "max n xa" in exI)
  apply (rule conjI)
  apply (rule_tac k = "max n xa" and m = "xa" in es_cadenaD',
         simp_all)
  apply (rule subsetI)
  apply (drule_tac B = "f n" and c = "xb" in subsetD)
```

```
  apply assumption
  apply (erule_tac k = "max n xa" and m = "n" in es_cadenaD')
  apply (simp_all add: max_def)
  done
```

□

Haciendo uso de estos teoremas vamos a probar ahora el Teorema 3.6.1.

Teorema: sea $\mathcal{C}$ una colección de conjuntos de carácter finito, y sea f una cadena creciente de elementos de $\mathcal{C}$. Entonces la unión de todos los elementos de la sucesión también está en $\mathcal{C}$.

```
theorem union_cadenas:
  "caracter_finito C ⟹ es_cadena f ⟹ ∀n. f n ∈ C
  ⟹ (⋃n. f n) ∈ C"
```

Demostración: por ser `C` de carácter finito, para probar que `⋃n. f(n) ∈ C`, hay que probar que todo subconjunto finito de dicha unión pertenece a `C`. Por el teorema anterior, un subconjunto de dicha unión debe ser subconjunto de un miembro de la sucesión. Por ser `C` de carácter finito también se tiene que es cerrada por subconjuntos, por lo que todo subconjunto de un miembro de la sucesión debe pertenecer a `C`, con lo que se concluye el teorema. □

A continuación enunciamos y probamos el teorema en Isabelle.

```
theorem union_cadenas:
  "caracter_finito C ⟹ es_cadena f ⟹ ∀n. f n ∈ C
  ⟹ (⋃n. f n) ∈ C"
  apply (frule carfin_cerrsub)
  apply (unfold caracter_finito_def cerrado_subconj_def)
  apply (drule_tac x = "⋃n. f n" in spec)
  apply (erule iffD2)
  apply (rule allI impI)+
  apply (drule_tac F = "S'" in cadena_index)
  apply blast+
  done
```

□

Sea S un conjunto de fórmulas, $\mathcal{C}$ una colección de conjuntos de fórmulas y f una sucesión o enumeración de fórmulas. En la prueba del Teorema 3.6.2, el Teorema de existencia de modelos, definimos una sucesión de conjuntos de fórmulas, a la que denominamos S_n. Recordémosla.

$$\begin{aligned} S_0 &= S \\ S_{n+1} &= \begin{cases} S_n \cup \{f(n)\} & \text{si } S_n \cup \{f(n)\} \in \mathcal{C} \\ S_n & \text{en otro caso} \end{cases} \end{aligned}$$

Definimos esta sucesión en Isabelle, como la siguiente función `(extend S C f)`, donde `f: nat ⇒ formula` representa una enumeración cualquiera de las fórmulas proposicionales. En la sección anterior hemos construido en efecto una enumeración. Esta función `(extend S C f)` aplicada a un natural `n` es el elemento n-ésimo de la suceción.

```
fun extend :: "formula set ⇒ formula set set ⇒
    (nat ⇒ formula) ⇒ nat ⇒ formula set"
where
  "extend S C f 0 = S"
| "extend S C f (Suc n) = (if extend S C f n ∪ f n ∈ C
     then extend S C f n ∪ f n
     else extend S C f n)"
```

Definimos ahora una función "`Extend`" tal que `(Extend S C f)` es la unión de todos los elementos de la sucesión S_n.

```
definition
  Extend :: "formula set ⇒ formula set set ⇒
    (nat ⇒ formula) ⇒ formula set" where
  "Extend S C f = (⋃n. extend S C f n)"
```

Se prueba de manera automática que la sucesión S_n que hemos definido es una cadena creciente.

```
theorem es_cadena_extend: "es_cadena (extend S C f)"
  by (simp add: es_cadena_def) blast
```
□

Se prueba por inducción en n que si $\mathcal{C}$ es una clase de consistencia, y $S \in \mathcal{C}$, entonces los elementos de la sucesión S_n pertenecen a $\mathcal{C}$.

```
theorem extend_en_C: "consistencia C ⟹ S ∈ C  ⟹
                      extend S C f n ∈ C"
  by (induct n) auto
```
□

Teorema: si $\mathcal{C}$ es una clase de consistencia de carácter finito, y $S \in \mathcal{C}$, entonces $\bigcup_n S_n$ también pertenece a $\mathcal{C}$, donde S_n es la sucesión que hemos definido anteriormente.

```
theorem Extend_en_C: "consistencia C ⟹ caracter_finito C ⟹
                      S ∈ C ⟹ Extend S C f ∈ C"
  apply (unfold Extend_def)
  apply (erule union_cadenas)
  apply (rule es_cadena_extend)
  apply (rule allI)
  apply (simp_all add: extend_en_C)
  done
```
□

En la prueba anterior hemos detallado cada una de las reglas que se aplican. Ahora bien, también se puede probar de forma automática mediante

```
by (metis Extend_def es_cadena_extend extend_en_C union_cadenas)
```
□

Teorema: el conjunto de fórmulas S está contenido en $\bigcup_n S_n$, donde S es el primer elemento de la sucesión S_n.

```
theorem Extend_subconjuntos: "S ⊆ Extend S C f"
  apply (rule subsetI)
  apply (simp add: Extend_def)
  apply (rule_tac x=0 in exI)
  apply simp
  done
```
□

Podemos automatizar nuevamente la demostración de este teorema como sigue

```
by (metis Extend_def UNIV_I UN_iff extend.simps(1) subsetI)
```
□

Definimos a continuación el concepto de conjunto maximal.

Definición: sea $\mathcal{C}$ una colección de conjuntos y S un conjunto de $\mathcal{C}$. Se dice que S es maximal en $\mathcal{C}$ si para todo conjunto S' tal que $S \subseteq S'$ se tiene que $S = S'$.

```
definition
  maximal :: "'a set ⇒ 'a set set ⇒ bool" where
  "maximal S C = (∀S'∈C. S ⊆ S' ⟶ S = S')"
```

Vamos a probar que, bajo ciertas hipótesis, si S_n es la sucesión definida anteriormente, entonces $\bigcup_n S_n$ es maximal en la colección de conjuntos $\mathcal{C}$ de la definición de S_n.

Teorema: sea $\mathcal{C}$ una colección de conjuntos de fórmulas de carácter finito, S un conjunto de fórmulas y f una enumeración de las fórmulas proposicionales. Entonces el conjunto $\bigcup_n S_n$ es maximal en $\mathcal{C}$.

```
theorem extend_maximal: "∀y. ∃n. y = f n ⟹
  caracter_finito C ⟹ maximal (Extend S C f) C"
```

Demostración: hay que probar que para todo conjunto `S'` tal que $\bigcup_n S_n \subseteq$ `S'`, se tiene que $\bigcup_n S_n =$ `S'`. Esto se prueba por doble contención, aunque una de las direcciones es trivial.

```
  apply (simp add: maximal_def Extend_def)
  apply (rule ballI)
  apply (rule impI)
  apply (rule subset_antisym)
  apply assumption
```

La otra contención se prueba por reducción al absurdo, suponiendo que

"∃ z. z ∈ S' ∧ z ∉ $\bigcup_x S_x$":

```
apply (rule ccontr)
apply (subgoal_tac "∃z. z ∈ S' ∧ z ∉ (⋃x. extend S C f x)")
prefer 2
apply blast
```

Debemos llegar a una contradicción, probando que

"z ∈ $\bigcup_x S_x$".

Eliminando los cuantificadores de las hipótesis, obtenemos como objetivo:

```
 "⋀S' z n.
    caracter_finito C ⟹
    S' ∈ C ⟹
    (⋃x. extend S C f x) ⊆ S' ⟹
    ¬ S' ⊆ (⋃x. extend S C f x) ⟹
    z ∈ S' ⟹ z ∉ (⋃x. extend S C f x) ⟹ z = f n ⟹ False"
```

Se puede probar que $S_n \cup \{f\ n\} \subseteq$ S', pues f(n) ∈ S' y $\bigcup_x S_x \subseteq$ S':

```
apply (subgoal_tac "extend S C f n ∪ f n ⊆ S'")
prefer 2
apply simp
apply (rule_tac B = "⋃x. extend S C f x" in subset_trans)
apply blast
apply assumption
```

Como C es de carácter finito, aplicando el teorema carfin_cerrsub, tenemos que también es cerrado por subconjuntos. Por último, usando esta hipótesis el resto de la prueba es inmediata. □

Incluimos la demostracióm completa del teorema en Isabelle.

```
theorem extend_maximal: "∀y. ∃n. y = f n ⟹
  caracter_finito C ⟹ maximal (Extend S C f) C"
  apply (simp add: maximal_def Extend_def)
  apply (rule ballI)
  apply (rule impI)
  apply (rule subset_antisym)
  apply assumption
  apply (rule ccontr)
  apply (subgoal_tac "∃z. z ∈ S' ∧ z ∉ (⋃x. extend S C f x)")
  prefer 2
  apply blast
```

```
apply (erule exE)
apply (erule conjE)
apply (erule_tac x=z in allE)
apply (erule exE)
apply (subgoal_tac "extend S C f n ∪ f n ⊆ S'")
prefer 2
apply simp
apply (rule_tac B = "⋃x. extend S C f x" in subset_trans)
apply blast
apply assumption
apply (drule carfin_cerrsub)
apply (unfold cerrado_subconj_def)
apply (drule_tac x = "S'" in bspec)
apply assumption
apply (erule_tac x = "extend S C f n ∪ f n" in allE)
apply (erule impE)
apply assumption
apply (erule_tac P = "z ∈ (⋃x. extend S C f x)" in notE)
apply (rule_tac a = "Suc n" in UN_I [OF UNIV_I])
apply simp
done
```

□

4.4.5. Conjuntos de Hintikka

En esta sección definiremos los conjuntos de Hintikka y probaremos el Lema de Hintikka, que afirma que todo conjunto de Hintikka es consistente.

Definición: un conjunto H de fórmulas proposicionales es llamado un conjunto de Hintikka si:

1. para cualquier variable proposicional p, no pueden pertenecer a la vez a H p y su negación;
2. $\bot$, $\neg\top \notin H$;
3. si $\neg\neg Z \in H$, entonces $Z \in H$;
4. si $A \wedge B \in H$, entonces $A \in H$ y $B \in H$;
5. si $\neg(A \vee B) \in H$, entonces $\neg A \in H$ y $\neg B \in H$;
6. si $A \vee B \in H$, entonces $A \in H$ o $B \in H$;
7. si $\neg(A \wedge B) \in H$, entonces $\neg A \in H$ o $\neg B \in H$;
8. si $A \longrightarrow B \in H$, entonces $\neg A \in H$ o $B \in H$;
9. si $\neg(A \longrightarrow B) \in H$, entonces $A \in H$ y $\neg B \in H$.

Para que un conjunto H sea de Hintikka exigimos que se cumplan estas nueve propiedades. Notemos que en la definición 3.5.1 sólo introdujimos cinco propiedades, pues las formulábamos usando la notación uniforme.

Definimos los conjuntos de Hintikkka a continuación en Isabelle, especificando las propiedades anteriores.

```
definition
  hintikka :: "formula  set ⇒ bool" where
  "hintikka H =
     ((∀p. ¬ (Var p ∈ H ∧ Neg (Var p) ∈ H)) ∧
     FF ∉ H ∧ Neg TT ∉ H ∧
     (∀Z. Neg (Neg Z) ∈ H ⟶ Z ∈ H) ∧
     (∀A B. And A B ∈ H ⟶ A ∈ H ∧ B ∈ H) ∧
     (∀A B. Neg (Or A B) ∈ H ⟶ Neg A ∈ H ∧ Neg B ∈ H) ∧
     (∀A B. Or A B ∈ H ⟶ A ∈ H ∨ B ∈ H) ∧
     (∀A B. Neg (And A B) ∈ H ⟶ Neg A ∈ H ∨ Neg B ∈ H) ∧
     (∀A B. Impl A B ∈ H ⟶ Neg A ∈ H ∨ B ∈ H) ∧
     (∀A B. Neg (Impl A B) ∈ H ⟶ A ∈ H ∧ Neg B ∈ H))"
```

Para probar el lema de Hintikka, definiremos antes una interpretación que es modelo de toda fórmula de un conjunto de Hintikka. Probaremos esto en el siguiente teorema.

Teorema: sea F una fórmula proposicional y H un conjunto de Hintikka. La interpretación i definida sobre las variables proposicionales como sigue

$$i(p) = \begin{cases} \text{True} & \text{si } p \in H \\ \text{False} & \text{en caso contrario} \end{cases}$$

es modelo de F.

```
theorem hintikka_cons : "hintikka H ⟹
                 (F ∈ H ⟶ interp (λp. Var p ∈ H) F) ∧
                 (Neg F ∈ H ⟶ interp (λp. Var p ∈ H) (Neg F))"
```

Demostración: hacemos la prueba por inducción en las fórmulas proposicionales. Accedemos a cada objetivo a probar usando las reglas `conjunct1` y `conjunct2`, eliminamos los cuantificadores y automatizamos la prueba. □

A continuación incluimos la prueba completa de este teorema en Isabelle.

```
theorem hintikka_cons : "hintikka H ⟹
                 (F ∈ H ⟶ interp (λp. Var p ∈ H) F) ∧
                 (Neg F ∈ H ⟶ interp (λp. Var p ∈ H) (Neg F))"
  apply (unfold hintikka_def)
  apply (rule_tac formula = "F" in formula.induct)
```

```
apply (rule conjI)
apply (simp (no_asm))
apply (drule conjunct2, drule conjunct2, drule conjunct1)
apply simp
apply (rule conjI)
apply (drule conjunct2, drule conjunct1)
apply simp
apply (simp (no_asm))
apply (rule conjI)
apply (simp (no_asm))
apply (drule conjunct1)
apply (erule_tac x = "nat" in allE)
apply fastforce
apply (rule conjI)
apply fastforce
apply (drule conjunct2, drule conjunct1, drule conjunct2,
       drule conjunct2, drule conjunct1)
apply fastforce
apply (rule conjI)
apply (drule conjunct2, drule conjunct1, drule conjunct1,
       drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct1)
apply (erule_tac x = "formula1" in allE)
apply (erule_tac x = "formula2" in allE)
apply fastforce
apply (drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct1)
apply (erule_tac x = "formula1" in allE)
apply (erule_tac x = "formula2" in allE)
apply fastforce
apply (rule conjI)
apply (drule conjunct2, drule conjunct1, drule conjunct1,
       drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct2, drule conjunct2, drule conjunct1)
apply (erule_tac x = "formula1" in allE)
apply (erule_tac x = "formula2" in allE)
apply fastforce
apply (drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct2, drule conjunct2, drule conjunct2,
       drule conjunct2, drule conjunct2, drule conjunct1)
apply (erule_tac x = "formula1" in allE)
apply (erule_tac x = "formula2" in allE)
```

```
  apply fastforce
  apply (rule conjI)
  apply (drule conjunct2, drule conjunct2, drule conjunct1,
         drule conjunct2, drule conjunct2, drule conjunct2,
         drule conjunct2, drule conjunct2, drule conjunct2,
         drule conjunct2, drule conjunct1)
  apply (erule_tac x = "formula1" in allE)
  apply (erule_tac x = "formula2" in allE)
  apply fastforce
  apply (drule conjunct2, drule conjunct1, drule conjunct2,
         drule conjunct2, drule conjunct2, drule conjunct2,
         drule conjunct2, drule conjunct2, drule conjunct2,
         drule conjunct2, drule conjunct2)
  apply (erule_tac x = "formula1" in allE)
  apply (erule_tac x = "formula2" in allE)
  apply fastforce
  done
```
□

Probamos el Lema de Hintikka como corolario del teorema anterior.

Teorema (Lema de Hintikka): todo conjunto de hintikka es consistente.

```
theorem hintikka : "hintikka (set H) ⟹ consistente H"
```

Demostración: usando el teorema anterior, la demostración de este teorema es automática. □

```
theorem hintikka : "hintikka (set H) ⟹ consistente H"
  by (metis Ball_set_list_all consistente_def hintikka_cons)
```
□

Probamos a continuación que $\bigcup_n S_n$ es un conjunto de Hintikka, donde S_n es la sucesión definida en la sección anterior. Recordemos que dicha definición depende de una clase de consistencia $\mathcal{C}$, un conjunto de fórmulas $S \in \mathcal{C}$ y una enumeración de las fórmulas proposicionales f.

Teorema: sea $\mathcal{C}$ una clase de consistencia de carácter finito, $S \in \mathcal{C}$ y f una enumeración de las fórmulas proposicionales. Entonces el conjunto de fórmulas $\bigcup_n S_n$ es un conjunto de Hintikka.

```
theorem extend_hintikka:
  assumes car_fin: "caracter_finito C"
  and surj: "∀y. ∃n. y = f n"
  shows "consistencia C ⟹ S ∈ C ⟹ hintikka (Extend S C f)"
```

Demostración: hay que demostrar cada una de las propiedades que definen un conjunto de Hintikka. Haciendo uso de los teoremas `extend_maximal` y `Extend_en_C`, la demostración es automática. □

```
theorem extend_hintikka:
  assumes car_fin: "caracter_finito C"
  and surj: "∀y. ∃n. y = f n"
  shows "consistencia C ⟹ S ∈ C ⟹ hintikka (Extend S C f)"
  apply (insert extend_maximal [OF surj car_fin])
  apply (frule_tac f = "f" in Extend_en_C)
  apply (rule car_fin)
  apply assumption
  apply (unfold consistencia_def hintikka_def maximal_def)
  apply (erule_tac x = "Extend S C f" in allE, erule impE,
         assumption)
  apply (erule conjE', fastforce)+
  apply (rule allI, rule allI, rule impI, fastforce)
  done
```

□

4.4.6. Teorema de existencia de modelos

En esta sección vamos a probar el Teorema de existencia de modelos.

Teorema de existencia de modelos: si $\mathcal{C}$ es una clase de consistencia proposicional y $S \in \mathcal{C}$, entonces S es consistente.

```
theorem existencia_modelos:
  "consistencia C ⟹ set S ∈ C ⟹ consistente S"
```

Demostración: debemos probar que existe una interpretación que es modelo de S. Definimos dicha interpretación para cada variable proposicional p como

$$i(p) = \begin{cases} \text{True} & \text{si } p \in \bigcup_n S_n \\ \text{False} & \text{en otro caso} \end{cases}$$

donde la sucesión S_n es la siguiente:

$$\begin{aligned} S_0 &= S \\ S_{n+1} &= \begin{cases} S_n \cup \{f(n)\} & \text{si } S_n \cup \{f(n)\} \in \mathcal{C}' \\ S_n & \text{en otro caso} \end{cases} \end{aligned}$$

definida para la colección de conjuntos de fórmulas $\mathcal{C}'$ y la enumeración de las fórmulas proposicionales f que siguen:

- $\mathcal{C}'$ es la extensión de la clase de consistencia `C` a otra de carácter finito, que hemos definido en Isabelle como `(mk_car_finito (subconjuntos C))`,
- f es la enumeración de las fórmulas proposicionales `diag_form`.

Para probar que esta interpretación es modelo de S aplicamos el teorema `hintikka_cons`, pero para ello debemos probar que el conjunto $\bigcup_n S_n$, para los $\mathcal{C}'$ y f anteriores, es de Hintikka. Aplicamos el teorema `extend_hintikka`, probando sus hipótesis.

- Para probar que $\mathcal{C}'$ es de carácter finito aplicamos el teorema `car_finito`.
- Para probar que f es una enumeración de las fórmulas proposicionales aplicamos de forma implícita el teorema `diag_undiag_form`, pues lo hemos añadido a `simp`.
- Para probar que $\mathcal{C}'$ es una clase de consistencia aplicamos los teoremas `consistencia_car_fin`, `subconj_consistencia` y `subconjuntos1`.
- La prueba de que $S \in \mathcal{C}'$ es automática. □

Incluimos la demostración completa del teorema en Isabelle.

```
theorem existencia_modelos:
  "consistencia C ⟹ set S ∈ C ⟹ consistente S"
  apply (unfold consistente_def)
  apply (rule_tac x = "(λp. Var p ∈ Extend (set S)
        (mk_car_finito (subconjuntos C)) diag_form)" in exI)
  apply (simp only: list_all_iff)
  apply (rule ballI)
  apply (rule hintikka_cons [THEN conjunct1, THEN mp])
  apply (rule extend_hintikka)
  apply (rule car_finito)
  apply (rule allI)
  apply (rule_tac x="undiag_form y" in exI)
  apply simp
  apply (rule consistencia_car_fin)
  apply (rule subconj_consistencia, simp)
  apply (rule subconjuntos1)
  apply (unfold mk_car_finito_def)
  apply (simp add: subconjuntos_def)
  apply blast
  apply (erule Extend_subconjuntos [THEN subsetD])
  done
```

□

4.5. Deducción Natural

En esta sección estableceremos las reglas de la deducción natural para la lógica proposicional.

Definimos la noción de demostrabilidad por deducción natural $S \vdash F$ de manera inductiva. Para ello usamos en Isabelle el comando `inductive`, que genera la menor relación cerrada bajo el conjunto de reglas siguiente:

- `Hipot`: toda fórmula del conjunto de hipótesis S es demostrable por deducción natural a partir de S;
- `TTI`: $S \vdash \top$;
- `FFE`: si $S \vdash \bot$, entonces, para cualquier F, $S \vdash F$;
- `NegI`: si $S \cup \{F\} \vdash \bot$, entonces $S \vdash \neg F$;
- `NegE`: si $S \vdash \neg F$ y $S \vdash F$, entonces $S \vdash \bot$;
- `DNegI`: si $S \vdash F$, entonces $S \vdash \neg\neg F$;
- `DNegE`: si $S \vdash \neg\neg F$, entonces $S \vdash F$;
- `AndI`: si $S \vdash G$ y $S \vdash H$, entonces $S \vdash G \wedge H$;
- `AndE1`: si $S \vdash G \wedge H$, entonces $S \vdash G$;
- `AndE2`: si $S \vdash G \wedge H$, entonces $S \vdash H$;
- `OrI1`: si $S \vdash G$, entonces $S \vdash G \vee H$;
- `OrI2`: si $S \vdash H$, entonces $S \vdash G \vee H$;
- `OrE`: si $S \vdash G \vee H$, $S \cup \{G\} \vdash F$ y $S \cup \{H\} \vdash F$, entonces $S \vdash F$;
- `ImplI`: si $S \cup \{F\} \vdash G$, entonces $S \vdash (F \Rightarrow G)$;
- `ImplE`: si $S \vdash (F \Rightarrow G)$ y $S \vdash F$, entonces $S \vdash G$.

Definimos a continuación la noción de demostrabilidad por deducción natural a partir de las reglas anteriores.

```
inductive
  deducnat :: "formula list ⇒ formula ⇒ bool"
  ("_ ⊢ _" [50,50] 50)
where
  Hipot: "F ∈ set S ⟹ S ⊢ F"
| TTI: "S ⊢ TT"
| FFE: "S ⊢ FF ⟹ S ⊢ F"
| NegI: "F # S ⊢ FF ⟹ S ⊢ Neg F"
| NegE: "S ⊢ Neg F ⟹ S ⊢ F ⟹ S ⊢ FF"
| DNegI: "S ⊢ F ⟹ S ⊢ Neg (Neg F)"
| DNegE: "S ⊢ Neg (Neg F) ⟹ S ⊢ F"
```

```
| AndI: "S ⊢ G ⟹ S ⊢ H ⟹ S ⊢ And G H"
| AndE1: "S ⊢ And G H ⟹ S ⊢ G"
| AndE2: "S ⊢ And G H ⟹ S ⊢ H"
| OrI1: "S ⊢ G ⟹ S ⊢ Or G H"
| OrI2: "S ⊢ H ⟹ S ⊢ Or G H"
| OrE: "S ⊢ Or G H ⟹ G # S ⊢ F ⟹ H # S ⊢ F ⟹ S ⊢ F"
| ImplI: "F # S ⊢ G ⟹ S ⊢ Impl F G"
| ImplE: "S ⊢ Impl F G ⟹ S ⊢ F ⟹ S ⊢ G"
```

Como hemos definido las reglas de la deducción natural de forma inductiva, Isabelle genera de forma automática un principio de inducción sobre las reglas de deducción natural. Accedemos a él de la siguiente forma:

```
thm deducnat.induct
```

El principio de inducción es el que sigue:

```
"?x1.0 ⊢ ?x2.0 ⟹
(⋀F S. F ∈ set S ⟹ ?P S F) ⟹
(⋀S. ?P S TT) ⟹
(⋀S F. S ⊢ FF ⟹ ?P S FF ⟹ ?P S F) ⟹
(⋀F S. F # S ⊢ FF ⟹ ?P (F # S) FF ⟹ ?P S (Neg F)) ⟹
(⋀S F.
    S ⊢ Neg F ⟹
    ?P S (Neg F) ⟹ S ⊢ F ⟹ ?P S F ⟹ ?P S FF) ⟹
(⋀S F. S ⊢ F ⟹ ?P S F ⟹ ?P S (Neg (Neg F))) ⟹
(⋀S F. S ⊢ Neg (Neg F) ⟹ ?P S (Neg (Neg F)) ⟹ ?P S F) ⟹
(⋀S G H.
    S ⊢ G ⟹
    ?P S G ⟹ S ⊢ H ⟹ ?P S H ⟹ ?P S (And G H)) ⟹
(⋀S G H. S ⊢ And G H ⟹ ?P S (And G H) ⟹ ?P S G) ⟹
(⋀S G H. S ⊢ And G H ⟹ ?P S (And G H) ⟹ ?P S H) ⟹
(⋀S G H. S ⊢ G ⟹ ?P S G ⟹ ?P S (Or G H)) ⟹
(⋀S H G. S ⊢ H ⟹ ?P S H ⟹ ?P S (Or G H)) ⟹
(⋀S G H F.
    S ⊢ Or G H ⟹
    ?P S (Or G H) ⟹
    G # S ⊢ F ⟹
    ?P (G # S) F ⟹ H # S ⊢ F ⟹ ?P (H # S) F ⟹ ?P S F) ⟹
(⋀F S G. F # S ⊢ G ⟹ ?P (F # S) G ⟹ ?P S (Impl F G)) ⟹
(⋀S F G.
    S ⊢ Impl F G
    ⟹ ?P S (Impl F G) ⟹
    S ⊢ F ⟹ ?P S F ⟹ ?P S G) ⟹ ?P ?x1.0 ?x2.0"
```

Como podemos observar se generan 15 casos, tantos como reglas de deducción natural hemos introducido.

Isabelle también genera de forma automática los siguientes teoremas:

- `deducnat.simps` (reglas de simplificación)
- `deducnat.intros` (reglas de introducción)
- `deducnat.inducts` (un caso particular de `deducnat.induct`)

4.5.1. Adecuación de la Deducción Natural

En esta sección vamos a probar el Teorema de Adecuación de la Deducción Natural para la lógica proposicional.

Teorema de Adecuación: si $S \vdash F$, entonces $S \models F$.

```
theorem Adecuacion: "S ⊢ F ⟹ consecuencia_logica S F"
```

Demostración: la prueba se realiza por inducción en las reglas de deducción natural; después cada caso se prueba de forma automática. □

El teorema en Isabelle es el siguiente:

```
theorem Adecuacion: "S ⊢ F ⟹ consecuencia_logica S F"
  apply (unfold consecuencia_logica_def model_def)
  apply (induct rule: deducnat.induct)
  apply (rule allI)
  apply (simp_all add: list_all_iff)
  apply blast+
  done
```

□

4.5.2. Completitud de la Deducción Natural

En esta sección vamos a probar la completitud de la deducción natural, es decir, si $S \models F$, entonces $S \vdash F$. Antes de hacerlo probaremos dos reglas derivadas de forma automática.

```
lemma derivada1: "A # G ⊢ B ⟹ G ⊢ A ⟹ G ⊢ B"
  by (rule ImplE) (rule ImplI)

lemma derivada2: "Neg F # S ⊢ FF ⟹ S ⊢ F"
  by (rule DNegE) (rule NegI)
```

En la definición 3.7.4 hemos definido los conjuntos de fórmulas F-consistentes por deducción natural, es decir, los conjuntos S tales que $S \nvdash F$.

En el Lema 3.7.1 hemos probado que la colección de todos los conjuntos F-consistentes por deducción natural es una clase de consistencia proposicional. Si aplicamos el lema `derivada2`, si tenemos $S \nvdash F$, también tenemos $\{\neg F\} \cup S \nvdash \bot$. Entonces, se puede probar que la colección de todos los conjuntos $\{\neg F\} \cup S$ tal que $\{\neg F\} \cup S \nvdash \bot$ es una clase de consistencia proposicional, para cada fórmula F.

Podemos generalizar este lema probándolo para todo conjunto S tal que $S \nvdash \bot$ en lugar de probarlo para todo conjunto $\{\neg F\} \cup S$, para cada fórmula F. Enunciamos el siguiente lema:

Lema: la colección de los conjuntos de fórmulas S tal que $S \nvdash \bot$ es una clase de consistencia.

```
lemma deriv_consistencia:
  "consistencia {S::formula set. ∃G. S = set G ∧ ¬ (G ⊢ FF)}"
```

Demostración: se deben probar todas las reglas que definen a las clases de consistencia. Para acceder a cada uno de los objetivos a probar usamos la regla `conjI`, y después la probamos por reducción al absurdo.

Como se sigue el mismo esquema en toda la prueba, tomemos un fragmento de ella para estudiarla. Veamos cómo se prueba la regla 4 de clase de consistencia: la referente a la conjunción.

Tras haber usado la regla `conjI`, obtenemos el objetivo:

```
 "⋀S G Ga H.
    S = set G ⟹
    ¬ G ⊢ FF ⟹
    And Ga H ∈ set G ⟹
    ∃Gb. insert Ga (insert H (set G)) = set Gb ∧ ¬ Gb ⊢ FF"
```

Eliminamos el cuantificador existencial, y la probamos por reducción al absurdo:

```
apply (rule_tac x = "Ga # (H # G)" in exI, simp)
apply (rule notI)
apply (rule notE, simp)
```

Tras aplicar la regla `derivada1`, tenemos dos objetivos por probar:

```
 "1. ⋀S G Ga H.
    S = set G ⟹
    ¬ G ⊢ FF ⟹
    And Ga H ∈ set G ⟹ Ga # H # G ⊢ FF ⟹ H # G ⊢ Ga
  2. ⋀S G Ga H.
```

```
S = set G ⟹
¬ G ⊢ FF ⟹
And Ga H ∈ set G ⟹ Ga # H # G ⊢ FF ⟹ G ⊢ H"
```

Para probar el primer objetivo aplicamos la regla de eliminación de la conjunción 1, por lo que sólo debemos probar {H} ∪ G ⊢ Ga ∧ H. Esto se tiene de las hipótesis, pues (Ga ∧ H) ∈ G.

```
apply (rule_tac H = "H" in AndE1, rule Hipot, simp)
```

El segundo objetivo se prueba de forma análoga, usando la regla de eliminación de la conjunción 2.

```
apply (rule_tac H = "H" and G = "Ga" in AndE2, rule Hipot, simp)
```
□

Incluimos la prueba completa de este lema en Isabelle.

```
lemma deriv_consistencia:
  "consistencia S::formula set. ∃G. S = set G ∧ ¬ (G ⊢ FF)"
  apply (unfold consistencia_def)
  apply (rule allI)
  apply (rule impI)
  apply simp
  apply (erule exE)
  apply (erule conjE)
  apply simp
  apply (rule conjI allI impI notI)+
  apply (erule notE)
  apply (rule_tac F = "Var p" in NegE)
  apply (rule Hipot, simp)+
  apply (rule conjI notI)+
  apply (erule notE)
  apply (rule Hipot, simp)
  apply (rule conjI notI)+
  apply (erule notE)
  apply (rule_tac F = "TT" in NegE)
  apply (rule Hipot, simp)
  apply (rule TTI)
  apply (rule conjI impI allI)+
  apply (rule_tac x = "F # G" in exI, simp)
  apply (rule notI)
  apply (rule notE, simp)
  apply (rule_tac F = "F" in NegE)
  apply (rule NegI, simp)
  apply (rule DNegE, rule Hipot, simp)
```

```
apply (rule conjI impI allI)+
apply (rule_tac x = "Ga # (H # G)" in exI, simp)
apply (rule notI)
apply (rule notE, simp)
apply (rule derivada1, rule derivada1, simp)
apply (rule_tac H = "H" in AndE1, rule Hipot, simp)
apply (rule_tac H = "H" and G = "Ga" in AndE2, rule Hipot, simp)
apply (rule conjI impI allI)+
apply (rule_tac x = "(Neg Ga) # ((Neg H) # G)" in exI, simp)
apply (rule notI, rule notE, simp)
apply (rule derivada1, rule derivada1, simp)
apply (rule NegI)
apply (rule_tac F = "Or Ga H" in NegE)
apply (rule Hipot, simp)
apply (rule OrI1, rule Hipot, simp)
apply (rule NegI)
apply (rule_tac F = "Or Ga H" in NegE)
apply (rule Hipot, simp)
apply (rule OrI2, rule Hipot, simp)
apply (rule conjI impI allI)+
apply (rule ccontr, simp)
apply (erule conjE, erule notE)
apply (rule OrE, rule Hipot, simp)
apply (erule_tac x = "Ga # G" in allE, erule impE, simp, simp)
apply (erule_tac x = "H # G" in allE, erule impE, simp, simp)
apply (rule conjI impI allI)+
apply (rule ccontr, simp)
apply (erule conjE, erule notE)
apply (rule_tac F = "And Ga H" in NegE, rule Hipot, simp)
apply (rule AndI)
apply (rule DNegE, rule NegI, simp)
apply (rule DNegE, rule NegI, simp)
apply (rule conjI impI allI)+
apply (rule ccontr, simp)
apply (erule notE)
apply (rule_tac F = "H" in NegE)
apply (drule conjunct2)
apply (erule_tac x = "H # G" in allE, erule impE, simp)
apply (rule NegI, simp)
apply (erule conjE)
apply (rule ImplE, rule Hipot, simp)
apply (erule_tac x = "Neg Ga # G" in allE, erule impE, simp)
apply (rule DNegE, rule NegI, simp)
apply (rule conjI impI allI)+
```

```
  apply (rule_tac x = "Ga # Neg H # G" in exI)
  apply (rule conjI, simp)
  apply (rule notI, rule notE, simp)
  apply (rule derivada1, rule derivada1, simp)
  apply (rule derivada2)
  apply (rule_tac F = "Impl Ga H" in NegE, rule Hipot, simp)
  apply (rule ImplI)
  apply (rule FFE)
  apply (rule_tac F = "Ga" in NegE)
  apply (rule Hipot, simp)+
  apply (rule NegI)
  apply (rule_tac F = "Impl Ga H" in NegE)
  apply (rule Hipot, simp)
  apply (rule ImplI, rule Hipot, simp)
  done
```
□

Vamos a probar por último el Teorema de Completitud de la Deducción Natural en Lógica Proposicional.

Teorema de Completitud: si $S \models F$, entonces $S \vdash F$.

```
theorem deducnat_completa: "consecuencia_logica S F ⟹ S ⊢ F"
```

Demostración: vamos a realizar la prueba por reducción al absurdo. Es decir, suponiendo que $\{\neg F\} \cup S \nvdash \bot$, queremos probar que existe una interpretación que es modelo de $\{\neg F\} \cup S$. Es decir, vamos a probar que el conjunto $\{\neg F\} \cup S$ es consistente. Esto lo hacemos usando el Teorema de existencia de modelos.

Para probar las hipótesis de dicho teorema usamos el lema `deriv_consistencia`, y automatizamos el resto de la prueba. □

Incluimos la demostración de este teorema en Isabelle.

```
theorem deducnat_completa: "consecuencia_logica S F ⟹ S ⊢ F"
  apply (rule derivada2)
  apply (rule notI [THEN notnotD])
  apply (rule_tac P = "∃e. list_all (interp e) (Neg F # S)" in notE)
  apply (unfold consecuencia_logica_def)
  apply (simp add: model_def)
  apply blast
  apply (subgoal_tac "consistente (Neg F # S)")
  prefer 2
  apply (rule existencia_modelos)
  apply (rule deriv_consistencia)
  apply blast
  apply (unfold consistente_def, auto)
  done
```
□

Bibliografía

[1] Archive of formal proofs. https://www.isa-afp.org/.

[2] Isabelle. http://isabelle.in.tum.de/index.html. Accedido por última vez el 11-06-2017.

[3] Stefan Berghofer. First-order logic according to fitting. *Archive of Formal Proofs*, 2007. https://www.isa-afp.org/entries/FOL-Fitting.shtml.

[4] Melvin Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[5] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

[6] Alexander Krauss. Defining recursive functions in isabelle/hol. Technical report, http://isabelle.in.tum.de/website-Isabelle2016/dist/Isabelle2016/doc/functions.pdf, 2008.

[7] Lawrence C Paulson. The foundation of a generic theorem prover. Technical report, University of Cambridge.

[8] Lawrence C. Paulson. Old introduction to isabelle. Technical report, University of Cambridge, http://isabelle.in.tum.de/website-Isabelle2011-1/dist/Isabelle2011-1/doc/intro.pdf, 2011.

[9] Lawrence C Paulson and Tobias Nipkow. Isabelle tutorial and user's manual. Technical report, University of Cambridge, Computer Laboratory, 1990.

[10] Floris van Doorn. Propositional calculus in coq. 2014. https://arxiv.org/abs/1503.08744.

[11] Makarius Wenzel. The isabelle/isar reference manual. Technical report, http://isabelle.in.tum.de/dist/Isabelle2016-1/doc/isar-ref.pdf, 2016.