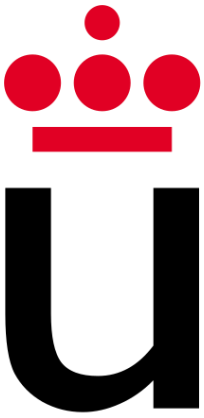

Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DE COMPUTADORES

Curso Académico 2023/2024

Trabajo de Fin de Grado

DEMOSTRACIÓN DE TEOREMAS MEDIANTE EL LENGUAJE DE PROGRAMACIÓN FUNCIONAL SCALA

Autor: Sergio Blázquez Teixeira

Tutor: Juan Manuel Serrano Hidalgo

ÍNDICE

Antes de comenzar, debo agradecer infinitamente el reiterado apoyo, material informativo y correcciones para mejorar que he recibido por parte de mi tutor de este trabajo, Juan Manuel Serrano Hidalgo. También agradecer a mis familiares y amigos la paciencia que me han transmitido, tras las incontables horas de explicaciones matemáticas que les he dado, sin terminar de comprender la mayoría de lo que comentaba, a veces incluso por mi propia parte.

1. INTRODUCCIÓN

En este escrito se va a probar la utilidad que tiene la correspondencia de Curry-Howard, la cual nos ayuda a establecer un paralelismo entre dos campos a primera vista completamente diferentes: la lógica y los lenguajes de programación funcional, en especial Scala.

Scala se diseñó en 2004 con el objetivo de mejorar algunas de las críticas que recibía Java. Pudiendo adoptar múltiples paradigmas, se puede utilizar tanto para programación funcional, como para programación orientada a objetos.

Lo que en la lógica son las proposiciones a probar, en la programación funcional son nuestros tipos de datos. Implementar un programa que utiliza esos tipos de datos es al final una forma de probar las proposiciones que tenemos.

En el apartado lógico se tienen diversas técnicas de resolución, de las cuales utilizaremos la deducción natural y la resolución, además de varios tipos de lógicas (proposicional, primer orden, segundo y superiores órdenes, etc) e incluso ejemplos recreativos, de donde sacaremos los puzles de Knights and Knaves para el último apartado.

Para los ejercicios de deducción natural, se deberá seguir este tipo de razonamiento para conseguir un código igual que demuestre su certeza. En la deducción natural, cada constante lógica puede introducirse o eliminarse por medio de unas reglas dispuestas. Partiendo de los supuestos o premisas, se deben ir aplicando estas reglas a las funciones respectivas para llegar a la conclusión.

Las siguientes dos reglas, específicas para la disyunción, son un ejemplo de las reglas utilizadas para cada conector en la deducción lógica:

- **vI, introducción de la disyunción.** $\dfrac{\Phi}{\Phi \vee \Psi}$ $\dfrac{\Phi}{\Psi \vee \Phi}$
- **vE, eliminación de la disyunción.** $\dfrac{\Phi \vee \Psi \quad \neg\Phi}{\Psi}$ $\dfrac{\Phi \vee \Psi \quad \neg\Psi}{\Phi}$

Para los ejercicios de resolución y refutación, se deberá demostrar la conclusión del problema por medio del uso de la regla de inferencia de la resolución. Esta regla dicta lo siguiente: teniendo dos proposiciones disyuntivas, en las cuales se tiene separadas un supuesto y su negación, se puede concluir en la disyunción de los otros supuestos de cada proposición.

El siguiente ejemplo resume la regla de la resolución:

- **(A OR B) AND (¬A OR C) → B OR C**

Y finalmente, los puzles de Knights and Knaves, que proponen un número de habitantes y un número de frases que ellos mismos dicen. Estos habitantes sólo pueden ser Knights o Knaves, y para resolver los puzles se requiere buscar el tipo de cada uno de ellos en base a las frases dichas. Estos puzles tienen su origen en el libro "What Is the Name of This Book?", de Raymond Smullyan. También se pueden presentar puzles en los cuales seamos nosotros los que planteemos preguntas simples, con respuesta de sí o no, aunque estos puzles no se desarrollarán en este trabajo.

Sabiendo que los Knights siempre dirán la verdad, y que los Knaves siempre mentirán, se debe sacar la conclusión de los tipos exactos de cada uno. El código a desarrollar se deberá apoyar en un razonamiento por pasos, buscando contradicciones que puedan darnos pistas sobre el tipo de cada uno. Además, siempre que sea necesario, se utilizarán funciones auxiliares basadas en tautologías.

La metodología a seguir para todos los ejercicios será la siguiente: primero se realizarán las demostraciones lógicas clásicas, con sus respectivos sistemas en cada uno de los ejercicios, después se hará el desarrollo de un programa análogo mediante Scala, para probar la certeza de la demostración realizada. Se ha de tener especial cuidado en tener un programa que muestre de forma sencilla el paralelismo entre código de Scala y demostración lógica, para poder así enseñar la correspondencia de Curry-Howard lo mejor posible.

2. OBJETIVOS

Los objetivos que se deben conseguir en este trabajo son los siguientes:

1. Explicar las bases de Scala y su uso para demostraciones lógicas. Enseñar los tipos algebraicos de datos y la programación orientada a objetos en Scala, y la correspondencia entre lógica y programación funcional.
2. Demostrar ejercicios de deducción natural por medio de reglas de inferencia y tautologías, mostrando los pasos en un diagrama conciso.
    - Desarrollar códigos manteniendo la correspondencia Curry-Howard con los diagramas creados que demuestren las conclusiones sacadas, definiendo previamente las funciones auxiliares necesarias.
3. Demostrar ejercicios de resolución y refutación utilizando las reglas de inferencia y funciones auxiliares necesarias, mostrando el diagrama que define los pasos para llegar a la conclusión.
    - Desarrollar códigos manteniendo la correspondencia Curry-Howard, utilizando una función de resolución obligatoria en todos los casos, además de definir las funciones auxiliares utilizadas.
4. Demostrar la lógica de orden superior por medio de puzles de Knights And Knaves, habiendo previamente definido el marco de trabajo, el cual definirá las reglas básicas y los tipos de estos puzles.
    - Desarrollar códigos en base a los razonamientos de cada puzle, siguiendo los mismos pasos que se hagan previamente en dichas demostraciones.

3. PREÁMBULO: SCALA Y LOS TIPOS ALGEBRAICOS DE DATOS

En la programación funcional no se tiene las clases o herencias que enriquecen los lenguajes orientados a objetos, por lo que los tipos que se tienen se basan simplemente en productos, sumas y exponenciales. Estos tipos de datos son los "Tipos Algebraicos de Datos", o TADs.

Tipos producto

Estos tipos tienen como resultado un valor X * Y, creado a partir de dos valores X e Y. La función constructora de estos tipos, siendo siempre de tamaño dos, es la siguiente:

```
1. create: (X, Y) -> X * Y
```

Ilustración 1: Constructor de los tipos producto.

Teniendo un valor de tipo producto, se puede obtener cualquiera de los dos valores que la forman.

Los tipos "case classes" se utilizan para poder automatizar cosas como la creación de nuevos objetos producto y los patrones.

```
1. case class Caja(largo: Double, ancho: Double, alto: Double)
2. case class Cajon(largo: Double, ancho: Double, alto: Double = 0.5)
3.
4. val miCaja = Caja(0.65, 0.75, 0.8)
5. val miCajon = Cajon(1.0, 0.6)
```

Ilustración 2: Creación dos variables provenientes de las case classes "Caja" y "Cajón"

Las tuplas en Scala se tienen como clases genéricas para N productos, con un máximo de 22. La definición de una tupla de dos objetos es:

```
1. object Std{
2.     case class Tuple2[A, B](_1: A, _2: B) }
```

Ilustración 3: Definición de la 2-tupla estándar de Scala.

Aprovechándose del azúcar sintáctico, la creación de variables "tuplas" se puede hacer de la siguiente forma:

```
1. val tupla3: (Int, Boolean, Caja) = (1, false, miCaja)
```
Ilustración 4: Creación de una variable 3-tupla, con un entero, un booleano y la variable de caja previa.

Para mostrar el valor "1", equivalente a un *Singleton*, se tiene el tipo "Unit", cuyo único valor sólo puede ser ( ).

```
1. val unidad: Unit = ()
```
Ilustración 5: Variable unidad del tipo "Unit".

El isomorfismo muestra cuando dos tipos representan la misma información. Partiendo de la siguiente plantilla, se puede crear un isomorfismo entre dos tipos:

```
 1. trait Isomorphic[A, B]{
 2.     def from(a: A): B
 3.     def to(b: B): A
 4.     // equality
 5.     def equalA(a1: A, a2: A): Boolean =
 6.         a1 == a2
 7.     def equalB(b1: B, b2: B): Boolean =
 8.         b1 == b2
 9.     // Bijection laws
10.     def law1(a: A): Boolean =
11.         equalA(to(from(a)), a)
12.     def law2(b: B): Boolean =
13.         equalB(from(to(b)), b) }
```
Ilustración 6: Plantilla para los tipos isomórficos.

Demostrando el isomorfismo "*Boolean * 1 ≃ Boolean*" creando el objeto:

```
1. object Iso1 extends Isomorphic[Boolean, (Boolean, Unit)]{
2.
3.     def from(b: Boolean): (Boolean, Unit) =
4.         (b, ())
5.
6.     def to(p: (Boolean, Unit)): Boolean =
7.         p._1 }
```
*Ilustración 7: Objeto que demuestra el isomorfismo de Bool*1 ≃ Bool*

Tipos suma

Los tipos suma, partiendo de X e Y, representan uno de sus dos tipos, X o Y.

Un ejemplo sería crear el tipo suma "Almacenamiento", pudiendo almacenar en una caja, o en un cajón (utilizando los tipos "Caja" y "Cajón" previos)

- *Almacenamiento := Caja + Cajon. Si tenemos un objeto de Almacenamiento, tenemos una Caja o un Cajon.*

Para crear los tipos suma en Scala, se debe utilizar la herencia con la palabra *sealed*, que impedirá la creación de subclases nuevas.

```
1. sealed abstract class Almacenamiento
2. case class Caja(largo: Double, ancho: Double, alto: Double) extends Almacenamiento
3. case class Cajon(largo: Double, ancho: Double, alto: Double = 0.5) extends
Almacenamiento
```

Ilustración 8: Tipo suma, creado como clase Almacenamiento

Entonces, podemos crear valores del tipo Almacenamiento con los constructores de sus subclases necesarios:

```
1. val miCaja: Almacenamiento = Caja(0.65, 0.75, 0.5)
2. val miCajon: Almacenamiento = Cajon(0.65, 0.75)
```

Ilustración 9: Valores del tipo Almacenamiento

Se puede verificar el tipo de cada valor por medio de *pattern matching*:

```
1. val tipo: String = miCaja match{
2.     case caja: Caja => "Caja"
3.     case cajon: Cajon => "Cajón"}

X. tipo: String = "Caja"
```

Ilustración 10: Pattern matching probando el tipo Caja de "miCaja"

La librería estándar de Scala ofrece dos tipos suma importantes: *Option* y *Either*:

```
1. object StdSumTypes{
2. sealed abstract class Option[A]
// Útil para manejar errores, pudiendo devolver:
// A: Un objeto en caso de ejecución correcta
3.     case class Some[A](a: A) extends Option[A]

// B: El tipo None en caso de no tener nada que devolver
4.     case class None[A]() extends Option[A]
5.
// Lo que utilizaremos para la lógica, devolviendo:
6.     sealed abstract class Either[A, B]
// A: El objeto en el lado izquierdo de la suma
7.     case class Left[A, B](a: A) extends Either[A, B]
// B: El objeto en el lado derecho de la suma
8.     case class Right[A, B](b: B) extends Either[A, B] }
```

Ilustración 11: Tipos suma Option y Either

Útiles para la devolución de errores, pudiendo evitar excepciones. En caso de tener la solución necesitada, se devuelve el lado respectivo, mientras que si tenemos una excepción, podemos devolver por el otro lado:

```
1. def divideWithOption(a: Double, b: Double): Option[Double] =
2.     if (b==0) None // None
3.     else Some(a/b)
```

Ilustración 12: División con tipo Suma para evitar una excepción dividiendo entre cero

El tipo 0 en Scala funciona como Unit para los productos, la identidad. Para mostrar la identidad en los tipos suma de tiene *Nothing*:

```
1. lazy val nada: Nothing =
2.     throw new Exception("no value of type Nothing")
```

Ilustración 13: Variable nada del tipo Nothing

Tipos exponente

Finalmente, los tipos exponente, los cuales no son más que los tipos función que se han estado usando hasta ahora (=>)

Para resumir estos tres tipos, se muestra a continuación una tabla con los constructores y destructores para cada tipo de datos:

Tabla 1: Constructores y destructores de los TADs

TAD	CONSTRUCTOR	DESTRUCTOR
Tipo suma X OR Y	**Left (x : X)** : Either[X, Y] **Right (y : Y)** : Either[X, Y]	**val eith**: Either[X, Y] = … **eith match{** **case Left(x : X) => …** **case Right(y : Y) => …}**
Tipo producto X AND Y	**(a : A, b : B)** : (A, B)	**val and: (A, B)** **and._1** : A **and._2** : B
Tipo exponente X => Y	**{ a : A => … : B}** : (A => B)	**val func** : A => B **val a** : A **func(a)** : B

4. PRUEBAS DE DEDUCCIÓN NATURAL

Para enlazar lo aprendido hasta ahora con los ejercicios a desarrollar, hay que explicar primero las igualdades de tipos que tienen entre sí. Es por esto por lo que para demostrar enunciados lógicos se tienen los siguientes tipos:

- True ($\top$) & False ($\bot$) = Unit & Nothing
- Sumas (OR, V) = Either (y Option)
- Multiplicaciones (AND, ^) = Tuplas de dos tipos

Y finalmente, el azúcar sintáctico => para crear funciones, que lo utilizaremos para igualar el funcionamiento de las flechas → en la lógica.

Todo esto se puede resumir en las siguientes tablas:

Scala ADTs	Logic
Unit	$\top$
Nothing	$\bot$
Either[P, Q]	$p \vee q$
(P, Q)	$p \wedge q$
P => Q	$p \to q$

Scala types	Propositions
(P, Either[Q, R])	$p \wedge (q \vee r)$
P => Q => (Q, R)	$p \to q \to q \wedge r$
(P => Nothing) => Nothing	$(p \to \bot) \to \bot$

Ilustración 14: Resumen de los tipos en Scala para la lógica

Tras esto, nos falta buscar igualdades para la negación y el bicondicional, que podemos traducir en los siguientes tipos:

```
1. type Not[P] = P => Nothing
2. type <=>[P, Q] = (P => Q, Q => P)
```

Ilustración 15: Tipos negación y bicondicional

A partir de este punto, todo enunciado y ejercicio que se muestre tendrá al final la salida computada por el compilador. En esta salida, nos es relevante verificar que el método o función que se ha definido aparezca correcto. Si aparece definido sin errores, quiere decir que es correcto.

A continuación, se muestra un ejemplo básico de cómo pasar enunciados lógicos a Scala:

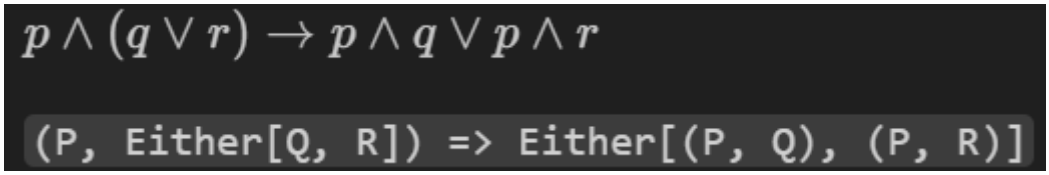

Ilustración 16: Enunciado de un ejemplo básico para traducir

```
1. def program[P, Q, R]: ((P, Either[Q, R])) => Either[(P, Q), (P, R)] = {
2.       case (p: P, Left(q: Q)) => Left((p, q): (P, Q))
3.       case (p: P, Right(r: R)) => Right((p, r): (P, R)) }

X. defined function program
```

Ilustración 17: Traducción del ejemplo básico en Scala

En el ejemplo previo, se tiene una tupla con P y con Q o R, y se necesita devolver una de dos tuplas: P y Q, o P y R. Como en la premisa se nos da P de forma inequívoca, se pasa a estudiar la segunda parte de la premisa: Either[Q, R].

Se deben estudiar los dos casos de la suma, por lo tanto, se dividen en:

- La parte izquierda, con q, con la cual devolvemos la tupla (p, q) por la parte izquierda del Either.
- La parte derecha, con r, con la cual devolvemos la tupla (p, r) por la parte derecha del Either.

Tras compilar, podemos observar que se ha definido la función “program”, por lo que todo está correcto.

Volviendo a las reglas de inferencia que vamos a utilizar, se muestra a continuación una tabla con todas las reglas que se van a ver a lo largo de las demostraciones de los ejercicios de deducción natural:

Tabla 2: Reglas de inferencia para la deducción natural y sus correspondencias en Scala

Introducción de la **conjunción**, **∧I**	$\dfrac{\Phi \quad \Psi}{\Phi \wedge \Psi}$ $\dfrac{\Psi \quad \Phi}{\Phi \wedge \Psi}$	**val a** : A **, val b** : B **(a, b)** : (A,B)
Eliminación de la **conjunción**, **∧E**	$\dfrac{\Phi \wedge \Psi}{\Phi}$ $\dfrac{\Phi \wedge \Psi}{\Psi}$	**val aANDb** : (A,B) **val a** : aANDb._1, **val b** : aANDb._2
Introducción de la **disyunción**, **∨I**	$\dfrac{\Phi}{\Phi \vee \Psi}$ $\dfrac{\Phi}{\Psi \vee \Phi}$	**val a** : A **Left(a**: A**)** : Either[A, B]
Eliminación de la **disyunción**, **∨E**	$\dfrac{\Phi \vee \Psi \quad \neg\Phi}{\Psi}$ $\dfrac{\Phi \vee \Psi \quad \neg\Psi}{\Phi}$	**val eith** : Either[A, B], **val notA** : Not[A] **eith match{** **case Left(a**: A**) => notA(a)** **case Right(b**: B**) => b}**
Introducción de la **negación**, **¬I**	$\dfrac{\Phi \ldots \bot}{\neg\Phi}$	**Type Not[A]** : A => Nothing
Eliminación de la **negación**, **¬E**	$\dfrac{\neg\neg\Phi}{\Phi}$	**val dN** : Not[Not[P]] => P
Introducción del **condicional material**, **→I**	$\dfrac{\Phi \ldots \Psi}{\Phi \rightarrow \Psi}$	**(a : A => … : B)** : (A => B)
Eliminación del **condicional material**, **→E**	$\dfrac{\Phi \ldots \Psi \quad \Phi}{\Psi}$	**val aTOb** : (A => B) **val a** : A **aTOb(a)** : B

En los desarrollos de código de los ejercicios, se encuentran las variables con los nombres descriptivos paralelos al desarrollo deductivo, además de líneas de comentario para ir guiando la solución paso por paso.

4.1. <u>Demostrar la siguiente deducción con el cálculo de deducción natural</u>

$$T[p \rightarrow q, \neg r \rightarrow \neg q, r \rightarrow \neg s] \vdash \neg s \vee \neg p$$

Ilustración 18: Enunciado del primer ejercicio de deducción natural

Cuya demostración por medio del cálculo de deducción natural puede hacerse de la siguiente forma:

Ilustración 19: Demostración por deducción natural del ejercicio 5.1

Como se puede observar, para este cálculo se necesita del principio del tercero excluido, proveniente de la lógica clásica.

En él se presenta que teniendo una proposición afirmando un supuesto, y otra contradiciéndolo, se debe tener siempre una de las dos, sin ser posible tener un tercer caso. Esto se representa de la siguiente forma: **A v ¬A**

Para su uso en Scala, se va a definir como un objeto proveniente de una interfaz, lo que nos permite utilizarlo con cualquier variable necesaria de forma sencilla.

```
1. trait LEM{
2.     def apply[P]: Either[P, Not[P]]}
3.
4. object LEMobj{
5.     type Prop[P] = Either[P, Not[P]]}

X. defined trait LEM
X. defined object LEMobj
```

Ilustración 20: Interfaz para usar el tercero excluido

Después, se debe importar como objeto, al cual vamos a llamar “lem”, antes de poner las premisas. Por lo tanto, el código que resulta es:

```
 1. def proof[P,Q,R,S](lem: LEM): (((P => Q, Not[R] => Not[Q]), R => Not[S])) =>
Either[Not[S],Not[P]] =

// 1. ((P -> Q) ^ (¬R -> ¬Q)) ^ (R -> ¬S)
 2.     (_1: ((P => Q, Not[R] => Not[Q]), R => Not[S])) => {

// 1.1. P -> Q                                          ^E(^E(1))
 3.      val _1_1: P => Q = _1._1._1

// 1.2. ¬R -> ¬Q                                        ^E(^E(1))
 4.      val _1_2: Not[R] => Not[Q] = _1._1._2

// 1.3. R -> ¬S                                         ^E(1)
 5.         val _1_3: R => Not[S] = _1._2

// 1.4. P v ¬P                                          LEM
 6.         val _1_4: LEMobj.Prop[P] = lem[P]
 7.         _1_4 match {

// 1.5. ¬P
 8.             case Right(_1_5: Not[P]) =>

// 1.5.1 ¬S v ¬P                                        vI(1.5)
 9.                 Right(_1_5: Not[P])
10.

// 1.6. P
11.             case Left(_1_6: P) =>

// 1.6.1. R v ¬R                                        LEM
12.                 val _1_6_1: LEMobj.Prop[R] = lem[R]
13.                 _1_6_1 match {

// 1.6.2. R
14.                     case Left(_1_6_2: R) =>

// 1.6.2.1. ¬S                                          →E(1.3, 1.6.2)
15.                         val _1_6_2_1: Not[S] = _1_3(_1_6_2)

// 1.6.2.2. ¬S v ¬P                                     vI(1.6.2.1)
16.                         Left(_1_6_2_1: Not[S])
17.

// 1.6.3. ¬R
18.                     case Right(_1_6_3: Not[R]) =>

// 1.6.3.1. Q                                           →E(1.1, 1.6)
19.                         val _1_6_3_1: Q = _1_1(_1_6)

// 1.6.3.2. ¬Q                                          →E(1.2, 1.6.3)
20.                         val _1_6_3_2: Not[Q] = _1_2(_1_6_3)

// 1.6.3.3. ⊥                                           →E(1.6.3.2, 1.6.3.1)
21.                         val _1_6_3_3: Nothing = _1_6_3_2(_1_6_3_1)

// 1.6.3.4. ¬S v ¬P                                     ⊥E(1.6.3.3)
22.                         _1_6_3_3

// 1.6.4. ¬S v ¬P                                       vE(1.6.2-1.6.2.2, 1.6.3-1.6.3.4, 1.6.1)
23.                 }
// 1.7. ¬S v ¬P                                         vE(1.5-1.5.1, 1.6-1.6.4, 1.4)
24.         }
// 2. ((P -> Q) ^ (¬R -> ¬Q)) ^ (R -> ¬S) -> ¬S v ¬P       →I(1, 1.7)
25.     }

X. defined function proof
```

Ilustración 21: Desarrollo del ejercicio de deducción natural 5.1

Al tener el desarrollo de código segmentado en variables, podemos copiar la deducción natural paso por paso para mejorar la claridad. Son destacables los dos usos del tercero excluido, en 1.4 y 1.6.1 (líneas 6 y 13, respectivamente), donde se deben estudiar todos los casos posibles.

4.1.2 <u>Demostrar la siguiente deducción con el cálculo de deducción natural, usando además funciones auxiliares</u>

Con el objetivo de aumentar la claridad y tener un nivel de abstracción superior, en este ejercicio se deben utilizar varias funciones auxiliares sobre el ejercicio anterior. Estas funciones se basan en las reglas de implicación material, modus tollendo tollens y la doble negación, además del necesario tercero excluido. También se usa el combinador "andThen" de Scala para concluir **A → C** desde **A → B** y **B → C.**

1	$((p \to q) \land (\neg r \to \neg q)) \land (r \to \neg s)$	$premise$
1.1	$p \to q$	$\land E(\land E(1))$
1.2	$\neg r \to \neg q$	$\land E(\land E(1))$
1.3	$r \to \neg s$	$\land E(1)$
1.4	$\neg\neg r \to \neg s$	$antDN(1.3)$
1.5	$s \to \neg r$	$tollensR(1.4)$
1.6	$\neg q \to \neg p$	$tollensL(1.1)$
1.7	$s \to \neg q$	$andThen(1.5 - 1.2)$
1.8	$s \to \neg p$	$andThen(1.7 - 1.6)$
1.9	$\neg s \lor \neg p$	$materialImplL(1.8)$
2	$((p \to q) \land (\neg r \to \neg q)) \land (r \to \neg s) \to \neg s \lor \neg p$	$\to I(1 - 1.9)$

Ilustración 22: Demostración por deducción natural del ejercicio 5.1.2

Empezando por definir las funciones auxiliares, tanto la doble negación como el tercero excluido se implementan como interfaces, para simplificar su uso:

```
 1. trait DN:
 2.     def left[P]: Not[Not[P]] => P
 3.     def right[P]: P => Not[Not[P]] =
 4.         p => np => np(p)
 5.
 6. trait LEM:
 7.     def apply[P]: Either[P, Not[P]]
 8.
 9. def materialImplL[P, Q](f: P => Q)(using LEM: LEM): Either[Not[P], Q] =
10.     LEM[P].fold(
11.         p => Right(f(p)),
12.         np => Left(np))
13.
14. def tollensL[P, Q](f: P => Q): Not[Q] => Not[P] =
15.     nq => p =>
16.         nq(f(p))
17.
18. def tollensR[P, Q](f: Not[P] => Not[Q])(using DN: DN): Q => P =
19.     q =>
20.         val nnq : Not[Not[Q]] = DN.right(q)
21.         val nnp : Not[Not[P]] = tollensL(f)(nnq)
22.         DN.left(nnp)
23.
24. def antDN[P, Q](f: P => Q)(using DN: DN): Not[Not[P]] => Q =
25.     nnp =>
26.         f(DN.left(nnp))

 X. defined trait DN
 X. defined trait LEM
 X. defined function materialImplL
 X. defined function tollensL
 X. defined function tollensR
 X. defined function antDN
```

Ilustración 23: Funciones auxiliares para el ejercicio 5.1.2

Y a continuación, la implementación de la deducción natural con el uso de dichas funciones auxiliares.

```
1. def proof[P,Q,R,S](_1_1: P => Q, _1_2: Not[R] => Not[Q], _1_3: R => Not[S])(using DN:
DN, LEM: LEM): Either[Not[S], Not[P]] =

// 1.4. ¬¬R -> ¬S                                              antDN(1.3)
2.     val _1_4 : Not[Not[R]] => Not[S] = antDN(_1_3)

// 1.5. S -> ¬R                                               tollensR(1.4)
3.     val _1_5 : S => Not[R] = tollensR(_1_4)

// 1.6. ¬Q -> ¬P                                              tollensL(1.5)
4.     val _1_6 : Not[Q] => Not[P] = tollensL(_1_1)

// 1.7. S -> ¬Q                                               andThen(1.5-1.2)
5.     val _1_7 : S => Not[Q] = _1_5 andThen _1_2

// 1.8. S -> ¬P                                               andThen(1.7-1.6)
6.     val _1_8 : S => Not[P] = _1_7 andThen _1_6

// 1.9. ¬S v ¬P                                               materialImplL(1.8)
7.     val _1_9 : Either[Not[S], Not[P]] = materialImplL(_1_8)

// 2. ((P -> Q) ^ (¬R -> ¬Q)) ^ (R -> ¬S) -> ¬S v ¬P           →I(1, 1.9)
8.     _1_9

X. defined function proof
```

Ilustración 24: Desarrollo del ejercicio de deducción natural 5.1.2

Resultando en el previo código, segmentado de nuevo en variables para tener una solución auto explicativa y concisa.

4.2. Demostrar la siguiente deducción con el cálculo de deducción natural

$$T[\neg p \to \neg s, \neg p \vee r, r \to \neg t] \vdash \neg s \vee \neg t$$

Ilustración 25: Enunciado del segundo ejercicio de deducción natural

Este ejercicio se desarrolla de una forma más clara de lo que era el desarrollo original del primer ejercicio. Es por esto por lo que no nos es tan necesario depender de funciones auxiliares. El desarrollo de la deducción natural es el siguiente:

1	$((\neg p \to \neg s) \wedge (\neg p \vee r)) \wedge (r \to \neg t)$	$premise$
1.1	$\neg p \to \neg s$	$\wedge E(\wedge E(1))$
1.2	$\neg p \vee r$	$\wedge E(\wedge E(1))$
1.3	$r \to \neg t$	$\wedge E(1)$
1.4	$\neg p$	$\vee E(1.2)$
1.4.1	$\neg s$	$\to E(1.1, 1.4)$
1.4.2	$\neg s \vee \neg t$	$\vee I(1.4.1)$
1.5	r	$\vee E(1.2)$
1.5.1	$\neg t$	$\to E(1.3, 1.5)$
1.5.2	$\neg s \vee \neg t$	$\vee I(1.5.1)$
1.6	$\neg s \vee \neg t$	$\vee E(1.4 - 1.4.2, 1.5 - 1.5.2, 1.2)$
2	$((\neg p \to \neg s) \wedge (\neg p \vee r)) \wedge (r \to \neg t) \to \neg s \vee \neg t$	$\to I(1 - 1.6)$

Ilustración 26: Demostración por deducción natural del ejercicio 5.2

En este caso, no necesitamos de ninguna regla externa, incluso podemos evitar el uso del tercero excluido.

Como se puede observar, todo depende de la segunda premisa, **¬P v R**, desde la cual podemos igualmente devolver un lado u otro de la conclusión. Siendo así, el código nos queda de la siguiente forma:

```
 1. def proof2[P,R,S,T]: (((Not[P] => Not[S], Either[Not[P],R]), R => Not[T])) =>
Either[Not[S],Not[T]] =

//  1.      ((¬P -> ¬S) ^ (¬P v R)) ^ (R -> ¬T)            PREMISE
 2. (_1 : ((Not[P] => Not[S], Either[Not[P],R]), R => Not[T])) => {

//  1.1.    ¬P -> ¬S                                      ^E(^E(1))
 3.     val _1_1 : Not[P] => Not[S] = _1._1._1

//  1.2.    ¬P v R                                        ^E(^E(1))
 4.     val _1_2 : Either[Not[P],R] = _1._1._2

//  1.3.    R -> ¬T                                       ^E(1)
 5.     val _1_3 : R => Not[T] = _1._2
 6.     _1_2 match{

//  1.4.    ¬P                                            vE(1.2)
 7.         case Left(_1_4: Not[P]) =>

//  1.4.1.  ¬S                                            →E(1.1, 1.4)
 8.             val _1_4_1 : Not[S] = _1_1(_1_4)

//  1.4.2.  ¬S v ¬T                                       vI(1.4.1)
 9.             Left(_1_4_1: Not[S])
10.

//  1.5.    R                                             vE(1.2)
11.         case Right(_1_5: R) =>

//  1.5.1.  ¬T                                            →E(1.3, 1.5)
12.             val _1_5_1: Not[T] = _1_3(_1_5)

//  1.5.2.  ¬S v ¬T                                       vI(1.5.1)
13.             Right(_1_5_1: Not[T])

//  1.6     ¬S v ¬T                                       vE(1.4-1.4.2, 1.5-1.5.2, 1.2)
14.     }

//  2       ((¬P -> ¬S) ^ (¬P v R)) ^ (R -> ¬T) -> (¬S v ¬T)    ->I(1-1.6)
15. }

 X. defined function proof2
```

Ilustración 27: Desarrollo del ejercicio de deducción natural 5.2

Como previamente explicado, partiendo de la segunda premisa, podemos ir desarrollando la conclusión.

- En el caso de tener la parte izquierda, ¬P, utilizamos la primera premisa para devolver ¬S, la parte izquierda de la conclusión.
- En el caso contrario, teniendo R, utilizamos la tercera premisa para devolver ¬T, la parte derecha de la conclusión

Habiendo finalizado el capítulo de deducción natural, pasamos a los ejercicios de resolución y refutación

5. PRUEBAS DE RESOLUCIÓN Y REFUTACIÓN

A continuación, se va a demostrar el desarrollo y funcionamiento de dos ejercicios: el primero de refutación, y el segundo de resolución. Ambos aparecen con sus respectivos diagramas para explicar la solución y su implementación paso por paso en Scala.

Para ambos ejercicios, se va a utilizar el método de resolución. En Scala, el método se desarrolla así:

```
1. def resolution[A, B, C](p1: Either[A, C], p2: Either[B, Not[C]]): Either[A, B] =
2.     (p1, p2) match
3.         case (Left(a), _) => Left(a)
4.         case (_, Left(b)) => Right(b)
5.         case (Right(c), Right(nc)) => nc(c): Nothing

X. defined function resolution
```

Ilustración 28: Método de resolución

En los dos casos en los que se tiene “A” o “B” proveniente de cualquiera de las dos sumas, se puede devolver el que se tenga por el lado correspondiente. En el caso restante en el que se no se tiene ninguno de los dos, podemos utilizar la contradicción con “C” y “¬C”, demostrando el absurdo, por lo que es una solución válida.

5.1. Resolver el siguiente ejercicio por medio del método de refutación

$$\frac{\neg(p \wedge q) \vee \neg r \qquad r \vee \neg q}{\neg p \vee \neg q}$$

Ilustración 29: Enunciado del primer ejercicio, refutación

Para resolver el ejercicio por medio de la refutación, se debe demostrar que no hay forma de contradecir el enunciado. Dicho de otra forma, se debe llegar a una contradicción contando con una conclusión falsa. Para ello, se necesitan utilizar funciones auxiliares, provenientes de varias tautologías.

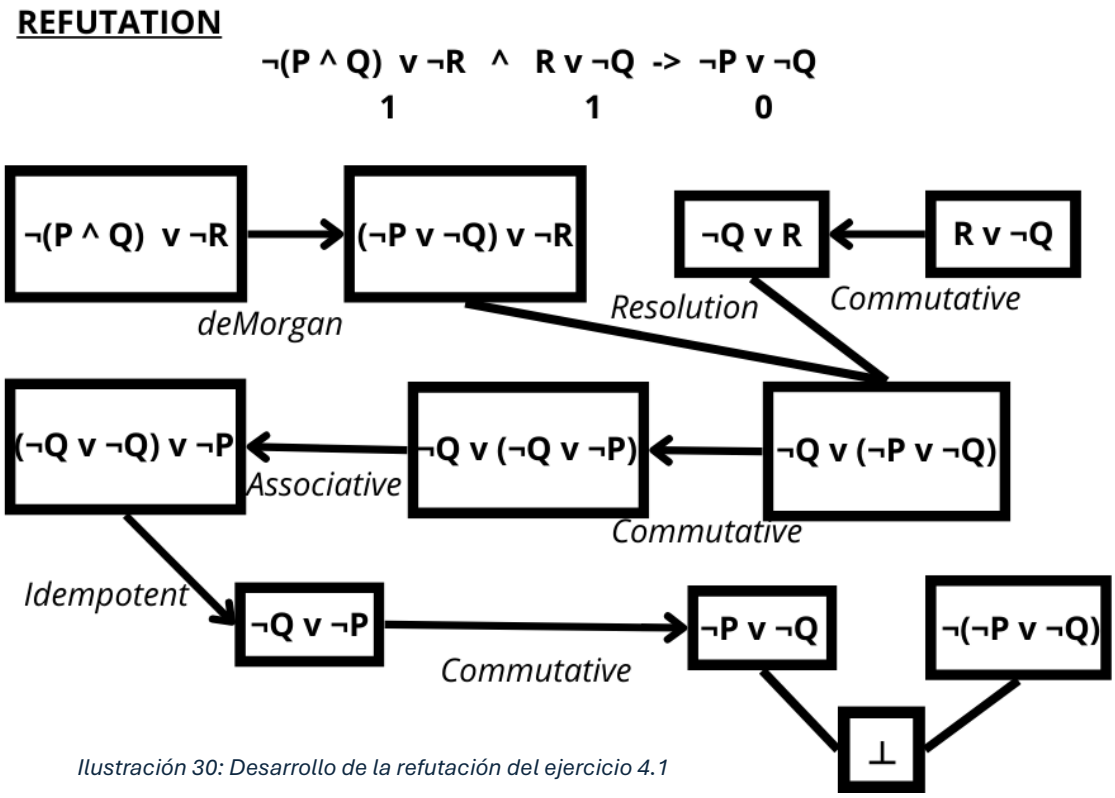

Ilustración 30: Desarrollo de la refutación del ejercicio 4.1

Para comenzar con el desarrollo en Scala de la demostración, se deben primero crear las funciones auxiliares de cada tautología. En la ilustración previa, se muestra el uso de cada regla auxiliar con una flecha, para mostrar pasos intermedios entre la resolución del ejercicio.

```
 1. def commutativeEither[A,B] : Either[A, B] => Either[B, A] =
 2.     case Left(a) => Right(a)
 3.     case Right(b) => Left(b)
 4.
 5. def deMorganAND2EITHER[A,B](thirdMiddleA: Either[A, Not[A]]): Not[(A, B)] =>
Either[Not[A], Not[B]] =
 6.     notAB =>
 7.         thirdMiddleA.fold(
 8.             a => Right((b) => notAB((a, b)))
 9.             ,
10.             na => Left((a) => na(a))
11.         )
12.
13. def idempotent[A] : Either[A, A] => A =
14.     _.fold(identity, identity)
15.
16. def associative[P,Q,R] : Either[P,Either[Q,R]] => Either[Either[P,Q],R] =
17.     case Left(p) => Left(Left(p))
18.     case Right(qORr) => qORr match
19.         case Left(q) => Left(Right(q))
20.         case Right(r) => Right(r)

 X. defined function commutativeEither
 X. defined function deMorganAND2EITHER
 X. defined function idempotent
 x. defined function associative
```

Ilustración 31: Funciones auxiliares provenientes de las tautologías necesarias

Una vez se tienen definidas correctamente las funciones auxiliares necesarias para el ejercicio, se puede pasar a desarrollar el ejercicio.

```
 1. def refutacion1[P,Q,R](p1: Either[Not[(P, Q)], Not[R]])(p2: Either[R, Not[Q]])(tmp :
Either[P, Not[P]]) : Not[Either[Not[P], Not[Q]]] => Nothing =

 2.     (negated : Not[Either[Not[P], Not[Q]]]) =>

 3.         val NPorNQ_orNR : Either[Either[Not[P], Not[Q]], Not[R]] =
          p1.fold(npq => Left(deMorganAND2EITHER(tmp)(npq)), nr => Right(nr))

 4.         val p2commutative : Either[Not[Q], R] = commutativeEither(p2)

 5.         val NQor_NPorNQ : Either[Not[Q], Either[Not[P], Not[Q]]] =
          resolution(p2commutative, NPorNQ_orNR)

 6.         val NQor_NQorNP : Either[Not[Q], Either[Not[Q], Not[P]]] =
          NQor_NPorNQ.fold(nq => Left(nq), npnq => Right(commutativeEither(npnq)))

 7.         val NQorNQ_orNP : Either[Either[Not[Q], Not[Q]], Not[P]] =
          associative(NQor_NQorNP)
 8.         val NQorNP : Either[Not[Q], Not[P]] =
          NQorNQ_orNP.fold(nqnq => Left(idempotent(nqnq)), np => Right(np))

 9.         val NPorNQ : Either[Not[P], Not[Q]] = commutativeEither(NQorNP)

10.         negated(NPorNQ) : Nothing

 X. defined function refutacion1
```

Ilustración 32: Desarrollo del ejercicio de refutación 4.1

En la línea 1. se tienen las premisas divididas en “p1”, “p2” y “p3”. Además, como es necesario el principio del tercio excluido para la función “deMorganAND2EITHER”, debemos ponerlo como premisa. Para mostrar que se llega a la contradicción, se devuelve “¬(¬P∨¬Q) → ⊥”.

Como la última cosa que se devuelve es ⊥ (Nothing), la parte de ¬(¬P∨¬Q) se coge como un parámetro, al que se llama “negated”.

A partir de la línea 3., se van poniendo las soluciones de aplicar cada función paso a paso. A cada variable se le va poniendo el nombre de su tipo para mayor claridad.

- Línea 3: uso de las leyes de De Morgan, devolviendo (¬P v ¬Q) v ¬R
- Línea 4: uso de la conmutatividad, devolviendo ¬Q v R
- Línea 5: resolución con las líneas 3 y 4, devolviendo ¬Q v (¬P v ¬Q)
- Línea 6: conmutatividad, devolviendo ¬Q v (¬Q v ¬P)
- Línea 7: asociatividad para reagrupar los tipos correctamente para su uso, devolviendo (¬Q v ¬Q) v ¬P
- Línea 8: propiedad de la idempotencia, devolviendo ¬Q v ¬P

- Línea 9: conmutatividad, devolviendo ¬P v ¬Q

  Y finalmente, teniendo esto último y su negación al principio, ya podemos contradecir el ejercicio, finalizando esta refutación.

5.2. <u>Resolver el siguiente ejercicio por medio del método de resolución</u>

$$T[\neg s \wedge (r \rightarrow t), \neg r \rightarrow (p \rightarrow q), t \rightarrow \neg r] \vdash \neg s \wedge \neg(\neg q \wedge p)$$

Ilustración 33: Enunciado del segundo ejercicio, resolución

Este ejercicio se debe resolver por medio de resolución, por lo que nos basta con llegar a tener la conclusión por medio de las premisas. Una vez se ha llegado a la conclusión desde las premisas y el uso de tautologías como funciones auxiliares, habremos demostrado que tenemos un argumento válido.

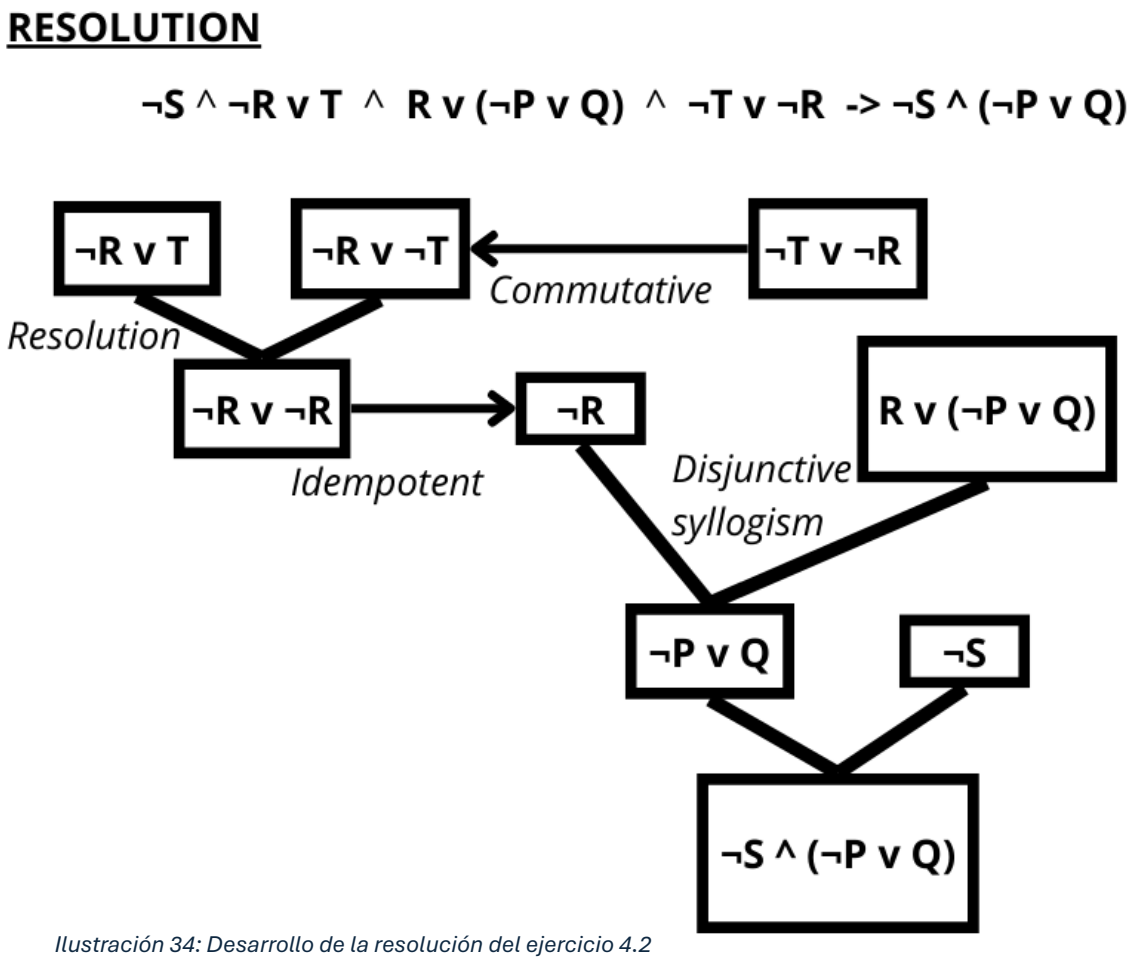

Ilustración 34: Desarrollo de la resolución del ejercicio 4.2

Además de las funciones auxiliares previamente utilizadas en el ejercicio anterior, ahora necesitamos una función para el silogismo disyuntivo, también llamado “Modus tollendo ponens”.

```
1. def disjunctiveSyllogism[A,B] : ((Either[A, B], Not[A])) => B =
2.     and =>
3.         and._1.fold(
4.             a => and._2(a),
5.             b => b)

X. defined function disjunctiveSyllogism
```

Ilustración 35: Función auxiliar modus tollendo ponens

Y a continuación, el desarrollo del ejercicio:

```
1. def resolucion2[P,Q,R,S,T](p1: (Not[S], Either[Not[R], T]))(p2: Either[R,
Either[Not[P], Q]])(p3: Either[Not[T], Not[R]]) : (Not[S], Either[Not[P], Q]) =

2.     val p3commutative : Either[Not[R], Not[T]] = commutativeEither(p3)

3.     val eitherNRs : Either[Not[R], Not[R]] = resolution(p1._2, p3commutative)

4.     val NR : Not[R] = idempotent(eitherNRs)

5.     val NPorQ : Either[Not[P], Q] = disjunctiveSyllogism(p2, NR)

6.     (p1._1, NPorQ) : (Not[S], Either[Not[P], Q])

X. defined function resolucion2
```

Ilustración 36: Desarrollo del ejercicio de resolución 4.2

Como en la resolución sólo necesitamos demostrar que se puede llegar a la conclusión con las premisas dadas, no se necesita nombrar más parámetros. A partir de la segunda línea se tiene:

- Línea 2: uso de la conmutatividad, devolviendo ¬R v ¬T
- Línea 3: aplicación de la resolución con la línea 2 y la parte derecha de la primera premisa, devolviendo ¬R v ¬R
- Línea 4: idempotencia con la variable previa, devolviendo ¬R
- Línea 5: silogismo disyuntivo con la línea 4 y la segunda premisa, devolviendo ¬P v Q

Y finalmente, juntamos la previa línea con la parte izquierda de la primera premisa, pasa resultar en ¬S y (¬P v Q) y concluir esta resolución.

Dado por finalizado el apartado de resolución, finalizaremos los ejercicios con los puzles de Knights and Knaves.

6. PUZLES DE KNIGHTS AND KNAVES

Los puzles de Knights and Knaves se basan en problemas lógicos en los que un tipo de personas sólo pueden decir la verdad, mientras que el otro sólo puede mentir. En ellos se tiene dos tipos de habitantes en una isla:

- Los caballeros (Knights), los cuales sólo pueden contestar a preguntas con verdades,
- Y los rufianes (Knaves), que de forma opuesta sólo pueden responder mintiendo.

Para estos ejercicios, serán los propios habitantes los que dirán frases, y partiendo de estas frases será como debamos sacar la conclusión del tipo exacto de cada uno de los habitantes presentes.

Entonces, las dos bases de un habitante son su tipo y lo que dice, por lo que se pueden definir en Scala de la siguiente forma:

```
1. trait Inhabitant:
2.     type Knight
3.     type Knave = Not[Knight]
4.     type Says[_]

X. defined trait Inhabitant
```

Ilustración 37: Definición de la interfaz para los habitantes

De esta forma, para cualquier habitante “X”, se va a poder definir si

- X.Knight, que mostrará que es un caballero.
- X.Knave, para los rufianes, que por dentro es simplemente un “No-caballero”.
- X.Says[_], marcado con un guión bajo al poder ser de cualquier tipo. Esto será lo que usemos para tener cualquier declaración que haga el habitante.

Para estos puzles, se han de tener en cuenta tres reglas básicas:

P1: $\forall x.\mathrm{Knight}(x) \vee \mathrm{Knave}(x)$, where $\mathrm{Knave}(x) \equiv \neg\mathrm{Knight}(x)$

P2: $\forall x.\mathrm{Knight}(x) \rightarrow \forall p.(\mathrm{Says}(x,p) \rightarrow p)$

P3: $\forall x.\mathrm{Knave}(x) \rightarrow \forall p.(\mathrm{Says}(x,p) \rightarrow \neg p)$

Ilustración 38: Tres reglas para los puzles de Knights and Knaves

1. Todo habitante debe ser un Knight o un Knave
2. Todo habitante Knight que diga algo, dirá la verdad, por lo que si Knight.Says[P], P = True
3. Todo habitante Knave que diga algo, mentirá, por lo que si Knave.Says[P], P = False (Not[P])

Y podemos definir estas reglas básicas así:

```
1. trait KnightsAndKnaves:
2.     // ∀ x. Inhabitant(x) → Knight(x) ∨ Knave(x)
3.     val P1: (x: Inhabitant) => Either[x.Knight, x.Knave]
4.     // ∀ p. ∀ x. Knight(x) → Says(x, p) → p
5.     val P2: [P] => (x: Inhabitant) => x.Knight => x.Says[P] => P
6.     // ∀ p. ∀ x. Knave(x) → Says(x, p) → Not[p]
7.     val P3: [P] => (x: Inhabitant) => x.Knave => x.Says[P] => Not[P]

X. defined trait KnightsAndKnaves
```

Ilustración 39: Definición del marco Knights and Knaves, que contiene las 3 reglas básicas

Teniendo así para los problemas que crear un objeto que provenga de KnightsAndKnaves o escribir "using KnightsAndKnaves" para poder utilizar estas reglas.

Hay una "cuarta regla" que realmente resulta implicada de las anteriores y de una posible contradicción que se debe tener en cuenta para todos los puzles de Knights and Knaves, incluso fuera de estas implementaciones en Scala. Esta regla es que un habitante nunca puede decir que él mismo es un Knave. Esta contradicción se da por hecho debido a que,

- Si estuviese diciendo la verdad, eso le convertiría en un Knight que dice que es un Knave, lo cual no puede ser,
- Y si estuviese mintiendo, sería un Knave que detrás de su mentira (teniendo doble negación, Not[Not[Knight]]), es un Knight que miente, algo imposible.

Esta cuarta regla o puzle se puede demostrar en Scala:

```
1. def puzzle1(KK: KnightsAndKnaves): (x: Inhabitant) => Not[x.Says[x.Knave]] =
2.     (x: Inhabitant) => (s: x.Says[x.Knave]) =>
3.         KK.P1(x) match
4.             case Left(xIsKnight: x.Knight) =>
5.                 KK.P2[x.Knave](x)(xIsKnight)(s)(xIsKnight) : Nothing
6.             case Right(xIsKnave: x. Knave) =>
7.                 KK.P3[x.Knave](x)(xIsKnave)(s)(xIsKnave) : Nothing

X. defined function puzzle1
```

Ilustración 40: Demostración de la contradicción de que un habitante diga que es un Knave

Teniendo en ambos casos que las contradicciones nos devuelven Nothing, que recordemos es nuestro tipo para False. Son de destacar los usos de las reglas básicas de Knights and Knaves:

- Línea 3: uso de la primera regla que dicta que un habitante sólo puede ser Knight o Knave. Aquí dividimos el problema en las dos posibles opciones para el habitante X.
- Línea 5: con la suposición de que "X es Knight", nos basamos en la segunda regla para pensar que lo que dice es cierto, que "X es Knave". Esto se contradice inmediatamente con el hecho de que partíamos de que "X es Knight".
- Línea 7: con la posibilidad restante de que "X es Knave" y la tercera regla, que nos explica que las declaraciones de un Knave deben ser falsas, tendríamos que "X no es Knave", contradicho con la suposición previa de que "X es Knave".

Demostrando así de forma exitosa que ningún habitante puede decir que él mismo es un Knave, lo que se podrá tomar de aquí en adelante como una mentira absoluta.

Para dejar el código de forma más legible, se puede añadir azúcar sintáctico a estos puzles. Este azúcar sintáctico se puede definir en un objeto con funciones que hagan lo mismo que estas reglas ya definidas:

```
 1. object Sugar:
 2.     implicit class NoOp[P](np: Not[P]):
 3.         def contradicts(p: P): Nothing = np(p)
 4.
 5.     def eitherKnightOrKnave[A](x: Inhabitant): KnightsAndKnaves ?=> (x.Knight ?=> A,
x.Knave ?=> A) => A =
 6.         P ?=> (f, g) => P.P1(x).fold(a => f(using a), a => g(using a))
 7.
 8.     def knightsAreTruthful[A](x: Inhabitant): KnightsAndKnaves ?=> x.Knight ?=>
x.Says[A] => A =
 9.         P ?=> xIsKnight ?=> xSaysA => P.P2(x)(xIsKnight)(xSaysA)
10.
11.     def knavesAreLiers[A](x: Inhabitant): KnightsAndKnaves ?=> x.Knave ?=> x.Says[A]
=> Not[A] =
12.         P ?=> xIsKnave ?=> xSaysA => P.P3(x)(xIsKnave)(xSaysA)

 X. defined object Sugar
```

Ilustración 41: Azúcar sintáctico para las reglas de Knights and Knaves

Y nos vale con importar todas las funciones del objeto Sugar para poder utilizarlas de aquí en adelante:

```
1. import Sugar._

X. import Sugar._
```

Ilustración 42: Importar todas las funciones del objeto Sugar

Con todo preparado, se puede pasar finalmente a desarrollar los tres puzles. Todo puzle aquí desarrollado procede de esta página. Esta herramienta crea puzles de Knights and Knaves con diferentes dificultades, y tiene un apartado para ver la solución con una explicación por pasos.

Se desarrollarán tres puzles, con creciente dificultad, lo que también se puede escoger en la herramienta anterior. Se busca poder pasar todo enunciado y explicación a Scala, para verificar si tienen un desarrollo correcto, lo que nos verifica que tengamos los puzles definidos y resueltos correctamente.

6.1. **Primer puzle, easy**

You have met a group of 3 islanders. Their names are Bob, Wallace, and Joan.

* Joan says: Wallace is a knight.

* Wallace says: Bob never lies.

* Joan says: Bob is my type.

Answer:

Bob, Wallace and Joan are all knights.

Reasoning:

* A knight or a knave will say they are the same type as a knight. So when Joan says they are the same type as Bob, we know that Bob is a knight.

* All islanders will call one of their same kind a knight. So when Wallace says that Bob is a knight, we know that Bob and Wallace are the same type. Since Bob is a knight, then Wallace is a knight.

* All islanders will call one of their same kind a knight. So when Joan says that Wallace is a knight, we know that Wallace and Joan are the same type. Since Wallace is a knight, then Joan is a knight.

Ilustración 43: Enunciado, solución y explicación del primer puzle

Para comenzar, con el razonamiento de la solución podemos hacer funciones auxiliares para llegar a esta. Empezando con la primera función, que viene del razonamiento "todo habitante dirá que es del mismo tipo que un caballero". Esto se muestra de la siguiente forma:

```
1. def deMorgan[P, Q](n: Not[Either[P, Q]]): (Not[P], Not[Q]) =
2.     (p => n(Left(p)), q => n(Right(q)))

X. defined function deMorgan
```

Ilustración 44: Función auxiliar de De Morgan para el puzle 6.1

```
 1. def saysSameTypeThenOtherKnight(using KnightsAndKnaves)(x: Inhabitant, y: Inhabitant):
 2.         x.Says[Either[(x.Knight, y.Knight), (x.Knave, y.Knave)]] => y.Knight =
 3.     xSays => eitherKnightOrKnave(x)(
 4.         xIsKnight ?=>
 5.             knightsAreTruthful(x)(xSays) match
 6.                 case Left(xIsKnight, yIsKnight) => yIsKnight
 7.                 case Right(xIsKnave, _) => xIsKnave contradicts xIsKnight,
 8.         xIsKnave ?=>
 9.             val l: (Not[(x.Knight, y.Knight)], Not[(x.Knave, y.Knave)]) =
                           deMorgan(knavesAreLiers(x)(xSays))
10.             eitherKnightOrKnave(y)(
11.                 yIsKnight ?=> yIsKnight ,
12.                 yIsKnave ?=> l._2 contradicts (xIsKnave, yIsKnave)))

 X. defined function saysSameTypeThenOtherKnight
```

Ilustración 45: Primera función auxiliar para el puzle 6.1

Como a mitad del desarrollo se ha necesitado una de las leyes de De Morgan, se ha creado previamente.

En esta función, como podemos observar, se debe partir de los dos habitantes. Uno de ellos dice que ambos son del mismo tipo, por lo que lo transformamos en el tipo "X.Says[_]", diciendo que "X.Knight y Y.Knight V X.Knave y Y.Knave". Tenemos que llegar a la conclusión de que el otro habitante sólo puede ser Knight, debido a que en caso contrario, X estaría diciendo que él mismo es Knave, lo que ya sabemos que es una contradicción.

Por lo tanto, utilizando el azúcar sintáctico, comenzamos con las dos posibilidades del habitante X.

- En la suposición de que es Knight, devolvemos por un lado que ambos son Knights, al decir la verdad en la parte izquierda de la frase, y por el otro la contradicción.

- En la suposición de que es Knave, buscamos las posibilidades de Y, el cual o es caballero (la conclusión que buscamos), o contradice la mentira que previamente ha dicho X.

Es por esto que la única solución que podemos devolver que no sea falsa, es que Y sea Knight, habiendo demostrado esta función y razonamiento.

El siguiente razonamiento es "todo habitante llama a uno de su mismo tipo caballero". Entonces tenemos un habitante que llama a otro caballero, lo que nos devolverá que ambos son Knights, o ambos son Knaves:

```
1. def saysOtherKnightThenSameType(using KnightsAndKnaves)(x: Inhabitant, y: Inhabitant):
2.         x.Says[y.Knight] => Either[(x.Knight, y.Knight), (x.Knave, y.Knave)] =
3.     xSays => eitherKnightOrKnave(x)(
4.         xIsKnight ?=>
5.             Left(xIsKnight, knightsAreTruthful(x)(xSays)),
6.         xIsKnave ?=>
7.             Right(xIsKnave, knavesAreLiers(x)(xSays)))

X. defined function saysOtherKnightThenSameType
```

Ilustración 46: Segunda función auxiliar para el puzle 6.1

La conclusión se devuelve de una forma muy simple:

- En caso de que el habitante (X) que está diciendo esto sea Knight, como debe decir la verdad, querrá decir que ambos son Knights.
- Mientras que, si miente, querrá decir que Y no es Knight, convirtiéndole también en Knave.

Para continuar con el uso de esta función, se debe crear otra secundaria que la apoye. Esta tercera función simplemente nos facilita la solución que, bajo el ojo humano, es clara: "si X e Y son iguales, y tenemos que Y es Knight, entonces X será Knight":

```
1. def sameTypeAndOtherKnightThenIsKnight[P, Q]: (Either[(P, Q), (Not[P], Not[Q])], Q) =>
P =
2.     case (Left(p, _),_) => p
3.     case (Right(_, nq),q) => nq(q)

X. defined function sameTypeAndOtherKnightThenIsKnight
```

Ilustración 47: Tercera función auxiliar para el puzle 6.1

Como el tercer razonamiento es igual que el segundo, pero con distintos habitantes, podemos reutilizar esas funciones, por lo que ya podemos pasar a desarrollar el puzle por completo.

Volviendo al enunciado del puzle, tendremos que definir a los tres habitantes, y las tres frases que han dicho, para probar si se puede conseguir la solución que se requiere. Entonces, la definición de este puzle deberá ser así:

```
1. def puzzle(using KnightsAndKnaves)(
2.     joan: Inhabitant, wallace: Inhabitant, bob: Inhabitant):
3.     (joan.Says[wallace.Knight],
4.      wallace.Says[bob.Knight],
5.      joan.Says[Either[(joan.Knight, bob.Knight),(joan.Knave, bob.Knave)]]) =>
(joan.Knight, wallace.Knight, bob.Knight) =
```

Ilustración 48: Definición y tipos del puzle 6.1

Teniendo los tres habitantes (Joan, Wallace & Bob) y las tres frases contenidas en las premisas, se debe llegar a nuestra conclusión de que los tres son Knights. Pasemos a la solución completa:

```
 1. def puzzle(using KnightsAndKnaves)(
 2.     joan: Inhabitant, wallace: Inhabitant, bob: Inhabitant):
 3.     (joan.Says[wallace.Knight],
 4.      wallace.Says[bob.Knight],
 5.      joan.Says[Either[(joan.Knight, bob.Knight),(joan.Knave, bob.Knave)]]) =>
(joan.Knight, wallace.Knight, bob.Knight) =

 6.     (joanSays1, wallaceSays, joanSays2) =>

 7.         // A knight or a knave will say they are the same type as a knight.
 8.         // So when Joan says they are the same type as Bob, we know that Bob is a
knight.
 9.         val bobIsKnight: bob.Knight =
                saysSameTypeThenOtherKnight(joan, bob)(joanSays2)

10.         // All islanders will call one of their same kind a knight. So when Wallace
says that Bob is a knight, we know that
11.         // Bob and Wallace are the same type.
12.         val sameTypeWB: Either[(wallace.Knight, bob.Knight), (wallace.Knave,
bob.Knave)] = saysOtherKnightThenSameType(wallace, bob)(wallaceSays)

13.         // Since Bob is a knight, then Wallace is a knight.
14.         val wallaceIsKnight: wallace.Knight =
                sameTypeAndOtherKnightThenIsKnight(sameTypeWB, bobIsKnight)

15.         // All islanders will call one of their same kind a knight. So when Joan says
that Wallace is a knight,
16.         // we know that Wallace and Joan are the same type.
17.         val sameTypeJW: Either[(joan.Knight, wallace.Knight), (joan.Knave,
wallace.Knave)] = saysOtherKnightThenSameType(joan, wallace)(joanSays1)


18.         //  Since Wallace is a knight, then Joan is a knight.
19.         val joanIsKnight: joan.Knight = sameTypeAndOtherKnightThenIsKnight(sameTypeJW,
wallaceIsKnight)
20.
21.         (joanIsKnight, wallaceIsKnight, bobIsKnight)

 X.     defined function puzzle
```

Ilustración 49: Desarrollo completo del puzle 6.1

En la línea 6 se nombran a las frases que se han dicho como “joanSays1, wallaceSays & joansays2”, para su uso con las funciones auxiliares definidas. Teniendo los razonamientos separados por los comentarios que los explican, van quedando las variables que se utilizan para la solución de una forma concisa. De esta forma, podemos acabar devolviendo las tres variables que hemos sacado de los razonamientos: “joanIsKnight, wallaceIsKnight & bobIsKnight”, lo que demuestra que el desarrollo del puzle funciona acorde con lo estudiado por medio de los razonamientos.

6.2. **Segundo puzle, tricky**

You have met a group of 4 islanders. Their names are Quentin, Beatrix, Robert, and Nancy.

* Robert says: Beatrix is a knave.

* Robert says: Nancy is lying.

* Quentin says: Robert is a knave and I am a knave.

Answer:

The knaves were Quentin, Nancy, and Beatrix, and the only knight was Robert.

Reasoning:

* Because Quentin said 'Robert is a knave and I am a knave,' we know Quentin is not making a true statement. (If it was true, the speaker would be a knight claiming to be a knave, which cannot happen.) Therefore, Quentin is a knave and Robert is a knight.

* All islanders will call a member of the opposite type a knave. So when Robert says that Beatrix is a knave, we know that Beatrix and Robert are opposite types. Since Robert is a knight, then Beatrix is a knave.

* All islanders will call a member of the opposite type a knave. So when Robert says that Nancy is a knave, we know that Nancy and Robert are opposite types. Since Robert is a knight, then Nancy is a knave.

Ilustración 50: Enunciado, solución y explicación del segundo puzle

En este puzle se tienen cuatro habitantes, pero sólo tres frases, lo que puede complicar un poco la búsqueda de la solución. Por suerte, Quentin menciona la contradicción que ya conocemos, por lo que se puede empezar a desmenuzar el puzle por ese apartado:

```
1. def saysBothKnaveThenIsKnave(using KK : KnightsAndKnaves)(x: Inhabitant, y:
Inhabitant):
2.         x.Says[(x.Knave, y.Knave)] => x.Knave =
3.         xSays => eitherKnightOrKnave(x)(
4.                 xIsKnight ?=>
5.                         knightsAreTruthful(x)(xSays)._1 contradicts xIsKnight,
6.                 xIsKnave ?=>
7.                         xIsKnave )

X. defined function saysBothKnaveThenIsKnave
```

Ilustración 51: Primera función auxiliar para el puzle 6.2

Con esta función demostramos que en caso de que alguien diga ser Knave, siempre va a llegarse a False. A continuación, hacemos la segunda parte de este razonamiento, que es sacar que el otro habitante mencionado en esta frase, por descarte, es Knight. Se han necesitado estas tres funciones secundarias para el desarrollo de la siguiente:

```
 1. def deMorgan2[P, Q](thirdMiddleP: Either[P, Not[P]])(n: Not[(P, Q)]): Either[Not[P],
Not[Q]] =
 2.     thirdMiddleP.fold(
 3.             p => Right((q) => n((p, q)))
 4.             ,
 5.             np => Left((p) => np(p)) )
 6.
 7. def commutativeEither[A,B] : Either[A, B] => Either[B, A] =
 8.     case Left(a) => Right(a)
 9.     case Right(b) => Left(b)
10.
11. trait DN:
12.     def left[P]: Not[Not[P]] => P
13.     def right[P]: P => Not[Not[P]] =
14.         p => np => np(p)

 X. defined function deMorgan2
 X. defined function commutativeEither
 X. defined trait DN
```

Ilustración 52: Funciones auxiliares de De Morgan, ley conmutativa y doble negación

```
 1. def saysBothKnaveThenIsKnaveAndOtherKnight(using KK : KnightsAndKnaves)(using dn :
DN)(x: Inhabitant, y: Inhabitant):
 2.         x.Says[(x.Knave, y.Knave)] => x.Knave => (x.Knave, y.Knight) =
 3.         xSays => xIsKnave => eitherKnightOrKnave(x)(
 4.                 xIsKnight ?=>
 5.                         xIsKnave contradicts xIsKnight,
 6.                 xIsKnave ?=>
 7.                         val lie : Not[(x.Knave, y.Knave)] = knavesAreLiers(x)(xSays)
 8.                         val knaveORnnknave : Either[x.Knave, Not[x.Knave]] =
commutativeEither(KK.P1(x)).fold(v => Left(v), g => Right(dn.right(g)))
 9.                         val eitherLie: Either[Not[x.Knave], Not[y.Knave]] =
deMorgan2(knaveORnnknave)(lie)
10.                         eitherLie.fold(
11.                                 xNotKnave => xNotKnave contradicts xIsKnave,
12.                                 yNotKnave => (xIsKnave, dn.left(yNotKnave)) ) )

 X. defined function saysBothKnaveThenIsKnaveAndOtherKnight
```

Ilustración 53: Segunda función auxiliar para el puzle 6.2

Esta función cogerá directamente la anterior como premisa, para devolver la pareja resultante de X.Knave y Y.Knight. Como ya partimos de X.Knave, estudiamos a partir de ese conocimiento.

Es importante recalcar la línea 8, en la cual se crea la variable knaveORnnknave, donde damos la vuelta a la primera regla que menciona que un habitante debe ser Knight o Knave, para tener Knave o Not[Knave], con el uso de la doble negación desde Knight. Esta variable la utilizamos en **deMorgan2** como tercero excluso, llegando así a la conclusión de que lo único en lo que podemos concluir es en que tenemos Y.Not[Knave], que es Y.Knight.

El segundo razonamiento, "todo habitante llama a uno del tipo opuesto Knave", se expresa de forma muy parecida al razonamiento del puzle anterior que mencionaba lo contrario:

```
1. def saysOtherKnaveThenDiffType(using KnightsAndKnaves)(using dn: DN)(x: Inhabitant, y:
Inhabitant):
2.     x.Says[y.Knave] => Either[(x.Knight, y.Knave), (x.Knave, y.Knight)] =
3.     xSays => eitherKnightOrKnave(x)(
4.         xIsKnight ?=>
5.             Left((xIsKnight, knightsAreTruthful(x)(xSays))),
6.         xIsKnave ?=>
7.             Right((xIsKnave, dn.left(knavesAreLiers(x)(xSays))))
8.     )

X. defined function saysOtherKnaveThenDiffType
```

Ilustración 54: Tercera función auxiliar para el puzle 6.2

La cual, como en el caso de la función opuesta, requiere de una función auxiliar para devolvernos lo que necesitamos:

```
1. def diffTypeAndIsKnightThenOtherKnave[P, Q]: (Either[(P, Not[Q]), (Not[P], Q)], P) =>
Not[Q] =
2.     case (Left(_, nq),_) => nq
3.     case (Right(np, _),p) => np(p)

X. defined function diffTypeAndIsKnightThenOtherKnave
```

Ilustración 55: Cuarta función auxiliar para el puzle 6.2

Y como el tercer razonamiento tiene las mismas bases y declaraciones que el segundo, podemos reutilizar esas funciones. Por lo tanto, el segundo puzle, por completo, nos queda de la siguiente forma:

```
 1. def puzzle2(using KnightsAndKnaves)(
 2.     quentin: Inhabitant, beatrix: Inhabitant, robert: Inhabitant, nancy:
Inhabitant)(using dn: DN):
 3.     (robert.Says[beatrix.Knave],
 4.      robert.Says[nancy.Knave],
 5.      quentin.Says[(quentin.Knave, robert.Knave)]) => (quentin.Knave, beatrix.Knave,
robert.Knight, nancy.Knave) =

 6.     (robertSays1, robertSays2, quentinSays) =>

 7.         // Because Quentin said 'Robert is a knave and I am a knave,' we know Quentin
is not making a true statement.
 8.         // (If it was true, the speaker would be a knight claiming to be a knave,
which cannot happen.)
 9.         val quentinIsKnave : quentin.Knave =
                saysBothKnaveThenIsKnave(quentin, robert)(quentinSays)

10.         // Therefore, Quentin is a knave and Robert is a knight.
11.         val quentinIsKnaverobertIsKnight: (quentin.Knave, robert.Knight) =
saysBothKnaveThenIsKnaveAndOtherKnight(quentin, robert)(quentinSays)(quentinIsKnave)
12.         val robertIsKnight : robert.Knight = quentinIsKnaverobertIsKnight._2

13.         // All islanders will call a member of the opposite type a knave.
14.         // So when Robert says that Beatrix is a knave, we know that Beatrix and
Robert are opposite types.
15.         val oppositeTypeRB: Either[(robert.Knight, beatrix.Knave), (robert.Knave,
beatrix.Knight)] = saysOtherKnaveThenDiffType(robert, beatrix)(robertSays1)

16.         // Since Robert is a knight, then Beatrix is a knave.
17.         val beatrixIsKnave: beatrix.Knave =
                diffTypeAndIsKnightThenOtherKnave(oppositeTypeRB, robertIsKnight)

18.         // All islanders will call a member of the opposite type a knave.
19.         // So when Robert says that Nancy is a knave, we know that Nancy and Robert
are opposite types.
20.         val oppositeTypeRN: Either[(robert.Knight, nancy.Knave), (robert.Knave,
nancy.Knight)] = saysOtherKnaveThenDiffType(robert, nancy)(robertSays2)

21.         // Since Robert is a knight, then Nancy is a knave.
22.         val nancyIsKnave: nancy.Knave =
                diffTypeAndIsKnightThenOtherKnave(oppositeTypeRN, robertIsKnight)
23.
24.         (quentinIsKnave, beatrixIsKnave, robertIsKnight, nancyIsKnave)

 X.     defined function puzzle2
```

Ilustración 56: Desarrollo completo del puzle 6.2

Como se explica previamente, empezamos por la declaración de Quentin, que nos lleva a conocer que es Knave. Gracias a esto, también sacamos que Robert es Knight, lo que nos facilita el uso de las dos últimas funciones para terminar con la conclusión necesaria, y resultado completo de la función: “quentinIsKnave, beatrixIsKnave, robertIsKnight & nancyIsKnave”.

6.3. **Tercer puzle, trickier**

Para finalizar con los puzles, buscamos uno de la máxima dificultad disponible:

You have met a group of 6 islanders. Their names are Bob, Neil, Francine, Zelda, Henry, and Wallace.

* Bob says: Francine always tells the truth.

* Neil says: Bob never tells the truth.

* Francine says: Neil is untruthful.

* Francine says: Bob is my type.

* Henry says: Wallace never lies.

* Henry says: Zelda always lies.

* Francine says: Wallace is not my type.

Answer:

The knaves were Neil, Henry, and Wallace, and the knights were Bob, Francine, and Zelda.

Reasoning:

* A knight or a knave will say they are the same type as a knight. So when Francine says they are the same type as Bob, we know that Bob is a knight.

* Both knights and knaves will say they are not the same type as a knave. So when Francine says they are a different type than Wallace, we know that Wallace is a knave.

* All islanders will call one of their same kind a knight. So when Bob says that Francine is a knight, we know that Francine and Bob are the same type. Since Bob is a knight, then Francine is a knight.

* All islanders will call a member of the opposite type a knave. So when Neil says that Bob is a knave, we know that Bob and Neil are opposite types. Since Bob is a knight, then Neil is a knave.

* All islanders will call one of their same kind a knight. So when Henry says that Wallace is a knight, we know that Wallace and Henry are the same type. Since Wallace is a knave, then Henry is a knave.

* All islanders will call a member of the opposite type a knave. So when Henry says that Zelda is a knave, we know that Zelda and Henry are opposite types. Since Henry is a knave, then Zelda is a knight.

Ilustración 57: Enunciado, solución y explicación del tercer puzle

Teniendo seis habitantes y siete frases, se debe empezar por revisar las frases de los habitantes que más hayan dicho. En este caso, son Francine y Henry, con tres y dos frases cada uno, respectivamente. Se empezará por las frases de Francine, ya que habla de ser del mismo tipo que otro habitante, y ser del tipo opuesto a otro, algo que nos puede dar bastantes pistas.

A pesar de tener varios razonamientos reutilizables, los volvemos a definir para poder separar este puzle de los anteriores. Como ya conocemos los trucos para conocer los tipos de otros habitantes, vamos aplicándolos en cada frase de Francine.

El primer razonamiento, donde siempre que alguien diga que es del mismo tipo que otro habitante, será porque el otro es Knight:

```
 1. def saysSameTypeThenOtherKnight(using KnightsAndKnaves)(x: Inhabitant, y: Inhabitant):
 2.         x.Says[Either[(x.Knight, y.Knight), (x.Knave, y.Knave)]] => y.Knight =
 3.     xSays => eitherKnightOrKnave(x)(
 4.         xIsKnight ?=>
 5.             knightsAreTruthful(x)(xSays) match
 6.                 case Left(xIsKnight, yIsKnight) => yIsKnight
 7.                 case Right(xIsKnave, _) => xIsKnave contradicts xIsKnight,
 8.         xIsKnave ?=>
 9.             val l: (Not[(x.Knight, y.Knight)], Not[(x.Knave, y.Knave)]) =
                           deMorgan(knavesAreLiers(x)(xSays))
10.             eitherKnightOrKnave(y)(
11.                 yIsKnight ?=> yIsKnight ,
12.                 yIsKnave ?=> l._2 contradicts (xIsKnave, yIsKnave)))

 X. defined function saysSameTypeThenOtherKnight
```

Ilustración 58: Primera función auxiliar del puzle 6.3

El segundo, que muestra lo contrario, estando ante un Knave en caso de que alguien diga que es de distinto tipo que él:

```
 1. def saysDiffTypeThenOtherKnave(using KnightsAndKnaves)(x: Inhabitant, y: Inhabitant):
 2.         x.Says[Either[(x.Knight, y.Knave), (x.Knave, y.Knight)]] => y.Knave =
 3.     xSays => eitherKnightOrKnave(x)(
 4.         xIsKnight ?=>
 5.             knightsAreTruthful(x)(xSays) match
 6.                 case Left(xIsKnight, yIsKnave) => yIsKnave
 7.                 case Right(xIsKnave, _) => xIsKnave contradicts xIsKnight,
 8.         xIsKnave ?=>
 9.             val l: (Not[(x.Knight, y.Knave)], Not[(x.Knave, y.Knight)]) =
deMorgan(knavesAreLiers(x)(xSays))
10.             eitherKnightOrKnave(y)(
11.                 yIsKnight ?=> l._2 contradicts(xIsKnave, yIsKnight),
12.                 yIsKnave ?=> yIsKnave)
13.         )

 X. defined function saysDiffTypeThenOtherKnave
```

Ilustración 59: Segunda función auxiliar del puzle 6.3

Tercer razonamiento, si alguien llama a otro Knight, es porque son del mismo tipo:

```
1. def saysOtherKnightThenSameType(using KnightsAndKnaves)(x: Inhabitant, y: Inhabitant):
2.         x.Says[y.Knight] => Either[(x.Knight, y.Knight), (x.Knave, y.Knave)] =
3.     xSays => eitherKnightOrKnave(x)(
4.         xIsKnight ?=>
5.             Left(xIsKnight, knightsAreTruthful(x)(xSays)),
6.         xIsKnave ?=>
7.             Right(xIsKnave, knavesAreLiers(x)(xSays)))

X. defined function saysOtherKnightThenSameType
```

Ilustración 60: Tercera función auxiliar del puzle 6.3

Una función secundaria para apoyar esta anterior:

```
1. def sameTypeAndIsKnightThenOtherKnight[P, Q]: (Either[(P, Q), (Not[P], Not[Q])], P) =>
Q =
2.     case (Left(_, q),_) => q
3.     case (Right(np, _),p) => np(p)

X. defined function sameTypeAndIsKnightThenOtherKnight
```

Ilustración 61: Cuarta función auxiliar del puzle 6.3

Cuarto razonamiento y opuesto del tercero, con dos habitantes de distinto tipo en caso de que uno llame al otro Knave:

```
1. def saysOtherKnaveThenDiffType(using KnightsAndKnaves)(using dn: DN)(x: Inhabitant, y:
Inhabitant):
2.     x.Says[y.Knave] => Either[(x.Knight, y.Knave), (x.Knave, y.Knight)] =
3.     xSays => eitherKnightOrKnave(x)(
4.         xIsKnight ?=>
5.             Left((xIsKnight, knightsAreTruthful(x)(xSays))),
6.         xIsKnave ?=>
7.             Right((xIsKnave, dn.left(knavesAreLiers(x)(xSays))))
8.     )

X. defined function saysOtherKnaveThenDiffType
```

Ilustración 62: Quinta función auxiliar del puzle 6.3

Con su respectiva función secundaria:

```
1. def diffTypeAndOtherKnightThenIsKnave[P, Q]: (Either[(P, Not[Q]), (Not[P], Q)], Q) =>
Not[P] =
2.     case (Left(_, nq),q) => nq(q)
3.     case (Right(np, _),_) => np

X. defined function diffTypeAndOtherKnightThenIsKnave
```

Ilustración 63: Sexta función auxiliar para el puzle 6.3

Y para los dos últimos razonamientos, al partir de frases cuya base ya tenemos definida (llamar a alguien Knight o Knave), sólo vamos a necesitar las funciones secundarias de cada uno, siendo las siguientes:

```
1. def sameTypeAndOtherKnaveThenIsKnave[P, Q]: (Either[(P, Q), (Not[P], Not[Q])], Not[Q])
=> Not[P] =
2.     case (Left(_, q),nq) => nq(q)
3.     case (Right(np,_),_) => np

X. defined function sameTypeAndOtherKnaveThenIsKnave
```

Ilustración 64: Séptima función auxiliar para el puzle 6.3

```
1. def diffTypeAndIsKnaveThenOtherKnight[P, Q]: (Either[(P, Not[Q]), (Not[P], Q)], Not[P])
=> Q =
2.     case (Left(p, _),np) => np(p)
3.     case (Right(_, q),_) => q

X. defined function diffTypeAndIsKnaveThenOtherKnight
```

Ilustración 65: Octava función auxiliar para el puzle 6.3

Tras todas estas funciones para demostrar los razonamientos, podemos finalizar desarrollando el puzle por completo, devolviendo la conclusión requerida:

- Neil, Henry y Wallace: Knaves.
- Bob, Francine y Zelda: Knights.

```
 1. def puzzle3(using KnightsAndKnaves)(
 2.     bob: Inhabitant, neil: Inhabitant, francine: Inhabitant, zelda: Inhabitant, henry:
Inhabitant, wallace: Inhabitant)(using dn: DN):

 3.         (bob.Says[francine.Knight],
 4.          neil.Says[bob.Knave],
 5.          francine.Says[neil.Knave],
 6.          francine.Says[Either[(francine.Knight, bob.Knight),(francine.Knave,
bob.Knave)]],
 7.          henry.Says[wallace.Knight],
 8.          henry.Says[zelda.Knave],
 9.          francine.Says[Either[(francine.Knight, wallace.Knave),(francine.Knave,
wallace.Knight)]]) => (neil.Knave, henry.Knave, wallace.Knave, bob.Knight,
francine.Knight, zelda.Knight) =

10.         (bobSays, neilSays, francineSays1, francineSays2, henrySays1, henrySays2,
francineSays3) =>
```

```
11.                 // A knight or a knave will say they are the same type as a knight.
12.                 // So when Francine says they are the same type as Bob, we know that
Bob is a knight.
13.                 val bobIsKnight: bob.Knight =
                    saysSameTypeThenOtherKnight(francine, bob)(francineSays2)

14.                 // Both knights and knaves will say they are not the same type as a
knave.
15.                 // So when Francine says they are a different type than Wallace, we
know that Wallace is a knave.
16.                 val wallaceIsKnave: wallace.Knave =
                    saysDiffTypeThenOtherKnave(francine, wallace)(francineSays3)

17.                 // All islanders will call one of their same kind a knight.
18.                 // So when Bob says that Francine is a knight, we know that Francine
and Bob are the same type.
19.                 val sameTypeBF: Either[(bob.Knight, francine.Knight),(bob.Knave,
francine.Knave)] = saysOtherKnightThenSameType(bob, francine)(bobSays)

20.                 // Since Bob is a knight, then Francine is a knight.
21.                 val francineIsKnight: francine.Knight =
                    sameTypeAndIsKnightThenOtherKnight(sameTypeBF, bobIsKnight)

22.                 // All islanders will call a member of the opposite type a knave.
23.                 // So when Neil says that Bob is a knave, we know that Bob and Neil
are opposite types.
24.                 val oppositeTypeNB: Either[(neil.Knight, bob.Knave), (neil.Knave,
bob.Knight)] = saysOtherKnaveThenDiffType(neil, bob)(neilSays)

25.                 // Since Bob is a knight, then Neil is a knave.
26.                 val neilIsKnave: neil.Knave =
                    diffTypeAndOtherKnightThenIsKnave(oppositeTypeNB, bobIsKnight)

27.                 // All islanders will call one of their same kind a knight.
28.                 // So when Henry says that Wallace is a knight, we know that Wallace
and Henry are the same type.
29.                 val sameTypeHW: Either[(henry.Knight, wallace.Knight),(henry.Knave,
wallace.Knave)] = saysOtherKnightThenSameType(henry, wallace)(henrySays1)

30.                 // Since Wallace is a knave, then Henry is a knave.
31.                 val henryIsKnave: henry.Knave =
                    sameTypeAndOtherKnaveThenIsKnave(sameTypeHW, wallaceIsKnave)

32.                 // All islanders will call a member of the opposite type a knave.
33.                 // So when Henry says that Zelda is a knave, we know that Zelda and
Henry are opposite types.
34.                 val oppositeTypeHZ: Either[(henry.Knight, zelda.Knave), (henry.Knave,
zelda.Knight)] = saysOtherKnaveThenDiffType(henry, zelda)(henrySays2)

35.                 // Since Henry is a knave, then Zelda is a knight.
36.                 val zeldaIsKnight: zelda.Knight =
                    diffTypeAndIsKnaveThenOtherKnight(oppositeTypeHZ, henryIsKnave)

37.                 (neilIsKnave, henryIsKnave, wallaceIsKnave, bobIsKnight,
francineIsKnight, zeldaIsKnight)

 X. defined function puzzle3
```

Ilustración 66: Desarrollo completo del puzle 6.3

Como en los dos puzles anteriores, el desarrollo de código contiene en los comentarios, antes de cada variable respectiva, los razonamientos usados para sacar dicha variable.

Con este tercer puzle demostrado de forma exitosa, se da por concluido el apartado de los puzles de Knights & Knaves, y con ello, los tres apartados de ejercicios a demostrar por completo.

7. CONCLUSIONES Y COMENTARIOS PERSONALES

Para concluir con este trabajo, debemos primeramente revisar los objetivos expuestos en el segundo capítulo, para verificar que hayamos conseguido completar todos. Con esto, podemos presentar que hemos:

1. Aprendido las bases de Scala y los TADs, con sus usos para estas demostraciones lógicas. Entendida la correspondencia de Curry-Howard para Scala
2. Demostrado la aplicación de Scala para ejercicios de deducción natural, siguiendo la correspondencia de Curry-Howard, además de con un funcionamiento correcto.
3. Demostrado la aplicación de Scala para ejercicios de resolución y refutación, con el uso de la función de resolución y funcionamiento correcto.
4. Demostrado la aplicación de Scala a los puzles lógicos de Knights and Knaves, siguiendo un razonamiento acorde y traduciéndolo al lenguaje para conseguir los tipos de los habitantes requeridos.

Como problemas que he encontrado a lo largo de la elaboración de este trabajo, debo mencionar la falta de material en la red, el cual, a pesar de no ser inexistente, se notaba escaso en referencia a otros lenguajes y aplicaciones. Este problema fue solventado en cada aparición con una consulta al tutor, el cual facilitó una grandísima cantidad de notebooks y vídeos explicativos sobre todas las dudas que planteaba, por lo que es una cuestión de facilidad e independencia que se me ha complicado.

El contenido a elaborar de este trabajo, por suerte, no se me ha terminado haciendo demasiado extraño, debido a que a lo largo de la carrera se dan varias asignaturas que sirven de base para poder llevar a cabo todo lo expuesto. Las dos asignaturas del grado de Ingeniería de Computadores que pueden ser prácticamente “aperitivos” para el trabajo son las de **Lógica y Matemática Discreta**, para todo el apartado lógico y matemático, obviamente, y **Lenguajes de Programación**, cuya parte de programación funcional en Haskell ayuda a comprender de forma sencilla todo lo relacionado con Scala.

Hablando desde el apartado personal, yo mismo veía los inicios de este trabajo de esa forma, códigos con un “peso menor”. La realidad es que, tras haber estado varios meses diseñando códigos, mejorando su legibilidad, e incluso diseccionando funciones para llegar a su optimización, me ha servido para valorar de una forma inconmensurable el trabajo continuo y el valor que puede llegar a aportar este conocimiento.

He de reconocer que las matemáticas siempre han formado una pequeña parte de mis intereses, a pesar de que siempre se tengan que dejar algunas cosas aparcadas para centrarse en el trabajo, ya que con este trabajo he podido recordar lo que me agrada comerme la cabeza con la lógica.

Como último apunte, mencionar las líneas futuras a desarrollar que tengo a la vista. En un futuro cercano, teniendo unas prácticas programando en el sector ferroviario, me veo ampliando conocimientos generalmente sobre lenguajes de programación y desarrollo de aplicaciones, e incluso aprendiendo nuevos idiomas para las conexiones internacionales. Hablando de un futuro más lejano, no descarto haber vuelto de vez en cuando a las matemáticas, sin descartar la idea de hacer algún estudio universitario que me llame la atención y tenga un alto contenido algorítmico que pueda aplicar a la programación.

5. REFERENCIAS

Hidalgo, J. M. (06 de 10 de 2021). *CurryHoward*. Obtenido de https://github.com/jserranohidalgo/urjc-pd/blob/master/PF-2/2.3-CurryHoward.ipynb

Hidalgo, J. M. (08 de 09 de 2021). *Scala orientada a objetos*. Obtenido de https://github.com/jserranohidalgo/urjc-pd/blob/master/PF-1/PF-1.1.ipynb

Hidalgo, J. M. (25 de 09 de 2022). *Lógica en Scala*. Obtenido de https://github.com/jserranohidalgo/urjc-pd/tree/master

Hidalgo, J. M. (25 de 09 de 2022). *Tipos de datos algebraicos*. Obtenido de https://github.com/jserranohidalgo/urjc-pd/blob/master/PF-2/2.1.2-Data%20types.ipynb

MacKinnon, D. (10 de 08 de 2018). *The Island if Knights and Knaves*. Obtenido de https://dmackinnon1.github.io/knaves/

University of California San Diego. (2022). *A List of Tautologies*. Obtenido de https://mathweb.ucsd.edu/~jeggers/Archive/2022Spring/Math109/tautologies.pdf

Wikipedia. (15 de 11 de 2023). *Resolución (lógica)*. Obtenido de https://es.wikipedia.org/wiki/Resolución_(lógica)

Wikipedia. (19 de 01 de 2024). *Deducción natural*. Obtenido de https://es.wikipedia.org/wiki/Deducción_natural

Wikipedia. (26 de 01 de 2024). *Knights and Knaves*. Obtenido de https://en.wikipedia.org/wiki/Knights_and_Knaves

Wikipedia. (19 de 03 de 2024). *Scala (programming language)*. Obtenido de https://en.wikipedia.org/wiki/Scala_(programming_language)