Título del TFM

Facultad de Matemáticas
Departamento de Ciencias de la Computación e Inteligencia Artificial
Trabajo Fin de Máster

Autor

Agradecimientos

El presente Trabajo Fin de Máster se ha realizado en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Sevilla.

Supervisado por

Tutor

Abstract

Resumen en inglés

Índice general

Capítulo 1

Introducción

En este capítulo se hace una breve introducción a la programación funcional en Haskell suficiente para entender su aplicación en los siguientes capítulos. Para una introducción más amplia se pueden consultar los apuntes de la asignatura de Informática de 1º del Grado en Matemáticas ([2]).

El contenido de este capítulo se encuentra en el módulo `PFH`

```
module PFH where
import Data.List
```

1.1. Introducción a Haskell

En esta sección se introducirán funciones básicas para la programación en Haskell. Como método didáctico, se empleará la definición de funciones ejemplos, así como la redefinición de funciones que Haskell ya tiene predefinidas, con el objetivo de que el lector aprenda "*a montar en bici, montando*".

A continuación se muestra la definición `(cuadrado x)` es el cuadrado de `x`. Por ejemplo, La definición es

```
-- |
-- >>> cuadrado 3
-- 9
-- >>> cuadrado 4
-- 16
cuadrado :: Int -> Int
cuadrado x = x * x
```

.

Capítulo 2

Interpretación algebraica de la lógica

En este capítulo se estudiarán las principales relaciones entre la lógica proposicional y los polinomios con coeficientes en cuerpos finitos, centrando la atención en $\mathbb{F}_2$, el cuerpo finito con dos elementos.

La idea principal que subyace en la interpretación algebraica de la lógica es la de hacerle corresponder a cada fórmula un polinomio, de forma que la función valor de verdad inducida por la fórmula se pueda entender como una función polinomial de $\mathbb{F}_2$. En otras palabras, se persigue que si la fórmula es verdadera, el valor del polinomio que tiene asociado sea 1; mientras que si la fórmula es falsa, el polinomio valga 0.

En la Figura 2.1 se muestra una representación gráfica de la relación entre las fórmulas proposicionales y $\mathbb{F}_2[\mathbf{x}]$. Destacar que se usa el ideal $\mathbb{I}_2 := (x_1 + x_1^2, ..., x_n + x_n^2) \subseteq \mathbb{F}_2[\mathbf{x}]$ y que *proj* es la proyección natural sobre el anillo cociente.

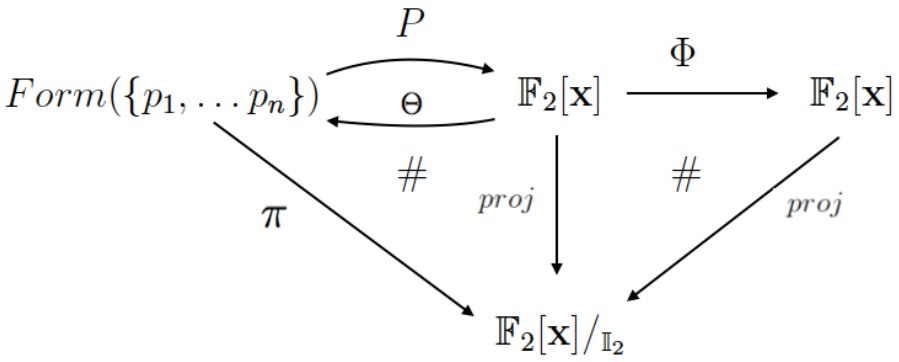

Figura 2.1: Relación entre las fórmulas proposicionales y $\mathbb{F}_2[\mathbf{x}]$

Durante todo el capítulo se paralelizará la exposición de la teoría y el desarrollo de las implementaciones en Haskell. Teniendo en cuenta que el objetivo principal de dichos programas es sentar las bases teóricas de una herramienta eficiente para resolver el problema SAT.

2.1. Lógica proposicional

En esta sección se introducirán brevemente los principales conceptos de la lógica proposicional, además de fijar la notación que se usará durante todo el trabajo. Para su implementación en Haskell se define el módulo `Lógica`:

```
module Logica where
```

Para conseguir este objetivo se utilizarán las siguientes librerías auxiliares:

```
import Control.Monad ( liftM
                     , liftM2)
import Data.List     ( union
                     , subsequences)
import Test.QuickCheck ( Arbitrary
                       , arbitrary
                       , oneof
                       , elements
                       , sized
                       , quickCheck)
import qualified Data.Set as S
```

Antes de describir el lenguaje de la lógica proposicional es importante recordar las definiciones formales de alfabeto y lenguaje:

Definición 2.1.1. Un *alfabeto* es un conjunto finito de símbolos.

Definición 2.1.2. Un *lenguaje* es un conjunto de cadenas sobre un alfabeto.

Especificando, un lenguaje formal es un lenguaje cuyos símbolos primitivos y reglas para unir esos símbolos están formalmente especificados. Al conjunto de las reglas se le denomina gramática formal o sintaxis. A las cadenas de símbolos que siguen las idicaciones de la gramática se les conoce como fórmulas bien formadas o simplemente fórmulas.

Para algunos lenguajes formales existe también una semántica formal que puede interpretar y dar significado a las fórmulas bien formadas del lenguaje. El lenguaje de la lógica proposicional es un caso particular de lenguaje formal con semántica.

2.1.1. Alfabeto y sintaxis

El alfabeto de la lógica proposicional está formado por tres tipos de elementos: las variables proposicionales, las conectivas lógicas y los símbolos auxiliares.

Definición 2.1.3. Las *variables proposicionales* son un conjunto finito de símbolos proposicionales que, tal y como su propio nombre indica, representan proposiciones. Dichas proposiciones son sentencias que pueden ser declaradas como verdaderas o falsas, es por esto que se dice que las variables proposicionales toman valores discretos (True o False). Es comúnmente aceptado (y así será en este trabajo) llamar al conjunto finito de las variables ($\mathcal{L} = \{p_1, ..., p_n\}$) lenguaje proposicional. A la hora de implementar las variables proposicionales se representarán mediante cadenas:

```
type VarProp = String
```

El conjunto de las fórmulas $Form(\mathcal{L})$ se construye usando las conectivas lógicas estándar:

- Monarias:
  - Negación ($\neg$)
  - Constante *true* ($\top$)
  - Constante *false* ($\bot = \neg\top$)
- Binarias:
  - Conjunción ($\wedge$)
  - Condicional o implicación ($\rightarrow$)
  - Disyunción ($\vee$)
  - Bicondicional ($\leftrightarrow$)

Los símbolos auxiliares con los que se trabajará serán los paréntesis, que se utilizan para indicar precedencia y ya vienen implementados en Haskell.

Definición 2.1.4. Una fórmula proposicional es una cadena sobre el alfabeto de la lógica proposicional descrito anteriormente (variables proposicionales, conectivas lógicas y símbolos auxiliares). Destacar que se construyen de forma recursiva, esto es, las variables se combinan mediante conectivas para formar fórmulas y éstas, a su vez, se combinan dando lugar a nuevas fórmulas.

Con todo ello se define el tipo de dato de las fórmulas proposicionales (`FProp`) usando constructores de tipo, es decir, una fórmula es $\top$, $\bot$, una variable proposicional o una combinación mediante una conectiva de dos fórmulas. En Haskell:

```
data FProp = T
           | F
           | Atom VarProp
           | Neg FProp
           | Conj FProp FProp
           | Disj FProp FProp
           | Impl FProp FProp
           | Equi FProp FProp
  deriving (Eq,Ord)
```

Por razones estéticas, además de facilitar el uso de este tipo de dato, se declara el procedimiento de escritura de las fórmulas:

```
instance Show FProp where
    show (T)        = "⊤"
    show (F)        = "⊥"
    show (Atom x)   = x
    show (Neg x)    = "¬" ++ show x
    show (Conj x y) = "(" ++ show x ++ " ∧ " ++ show y ++ ")"
    show (Disj x y) = "(" ++ show x ++ " ∨ " ++ show y ++ ")"
    show (Impl x y) = "(" ++ show x ++ " → " ++ show y ++ ")"
    show (Equi x y) = "(" ++ show x ++ " ↔ " ++ show y ++ ")"
```

Las fórmulas atómicas carecen de una estructura formal más profunda; es decir, son aquellas fórmulas que no contienen conectivas lógicas. En la lógica proposicional, las únicas fómulas atómicas que aparecen son las variables proposicionales. Por ejemplo:

```
p, q, r :: FProp
p  = Atom "p"
q  = Atom "q"
r  = Atom "r"
```

Combinando las fórmulas atómicas mediante el uso de las conectivas lógicas anteriormente enumeradas obtenemos lo que se denomina como fórmulas compuestas. Aunque estén definidas en el propio tipo de dato, con el objetivo de facilitar su uso, implementaremos las conectivas lógicas como funciones entre fórmulas:

`(no f)` es la negación de la fórmula f.

```
no :: FProp -> FProp
no = Neg
```

`(f ∨ g)` es la disyunción de las fórmulas f y g.

```
(∨) :: FProp -> FProp -> FProp
(∨)  = Disj
infixr 5 ∨
```

`(f ∧ g)` es la conjunción de las fórmulas f y g.

```
(∧) :: FProp -> FProp -> FProp
(∧)  = Conj
infixr 4 ∧
```

`(f → g)` es la implicación de la fórmula f a la fórmula g.

```
(→) :: FProp -> FProp -> FProp
(→)  = Impl
infixr 3 →
```

`(f ↔ g)` es la equivalencia entre las fórmulas f y g.

```
(↔) :: FProp -> FProp -> FProp
(↔) = Equi
infixr 2 ↔
```

Durante el desarrollo del trabajo se definirán distintas propiedades sobre las fórmulas proposicionales. Es bien sabido que una ventaja que nos ofrece Haskell a la hora de trabajar es poder definir también dichas propiedades y comprobarlas. Sin embargo, como las fórmulas proposicionales se han definido por el usuario el sistema no es capaz de generarlas automáticamente. Es necesario declarar que `FProp` sea una instancia de `Arbitrary` y definir un generador de objetos del tipo `FProp` como sigue:

```
instance Arbitrary FProp where
  arbitrary = sized prop
    where
      prop n  | n <= 0     = atom
              | otherwise  = oneof [
                    atom
                    , liftM Neg subform
                    , liftM2 Conj subform  subform
                    , liftM2 Disj subform  subform
                    , liftM2 Impl subform  subform
                    , liftM2 Equi subform' subform' ]
        where
          atom     = oneof [liftM Atom (elements ["p","q","r","s"]),
                            elements [F,T]]
          subform  = prop ( n `div` 2)
          subform' = prop ( n `div` 4)
```

Dadas dos fórmulas F, G y p una variable proposicional, se denota por $F\{p/G\}$ a la fórmula obtenida al sustituir cada ocurrencia de p en F por la fórmula G.

Se implementa $f\{p/g\}$ mediante la función `(sustituye f p g)`, donde f es la fórmula original, p la variable proposicional a sustituir y g la fórmula proposicional por la que se sustituye. Al ser `FProp` un tipo recursivo se hará dicha definición usando patrones:

```
-- | Por ejemplo,
-- >>> sustituye (no p) "p" q
-- ¬q
-- >>> sustituye (no (q ∧ no p)) "p" (q ↔ p)
-- ¬(q ∧ ¬(q ↔ p))
sustituye :: FProp -> VarProp -> FProp -> FProp
sustituye T            _ _ = T
sustituye F            _ _ = F
sustituye (Atom f)     p g | f == p = g
                           | otherwise = Atom f
sustituye (Neg f)      p g = Neg (sustituye f p g)
sustituye (Conj f1 f2) p g = Conj (sustituye f1 p g) (sustituye f2 p g)
sustituye (Disj f1 f2) p g = Disj (sustituye f1 p g) (sustituye f2 p g)
sustituye (Impl f1 f2) p g = Impl (sustituye f1 p g) (sustituye f2 p g)
sustituye (Equi f1 f2) p g = Equi (sustituye f1 p g) (sustituye f2 p g)
```

2.1.2. Semántica

Definición 2.1.5. Una interpretación o valoración es una función $i : \mathcal{L} \to \{0,1\}$.

En Haskell se representará como un conjunto de fórmulas atómicas. Las fórmulas que aparecen en dicho conjunto se suponen verdaderas, mientras que las restantes fórmulas atómicas se suponen falsas.

```
type Interpretacion = [FProp]
```

El significado de la fórmula f en la interpretación i viene dado por la función de verdad en su sentido más clásico, tal y como se detalla en el código.

`(significado f i)` es el significado de la fórmula f en la interpretación i:

```
-- | Por ejemplo,
--
-- >>> significado ((p ∨ q) ∧ ((no q) ∨ r)) [r]
-- False
-- >>> significado ((p ∨ q) ∧ ((no q) ∨ r)) [p,r]
-- True
significado :: FProp -> Interpretacion -> Bool
significado T          _ = True
significado F          _ = False
significado (Atom f)   i = (Atom f) `elem` i
significado (Neg f)    i = not (significado f i)
significado (Conj f g) i = (significado f i) && (significado g i)
significado (Disj f g) i = (significado f i) || (significado g i)
significado (Impl f g) i = significado (Disj (Neg f) g) i
significado (Equi f g) i = significado (Conj (Impl f g) (Impl g f)) i
```

Una interpretación i es modelo de la fórmula $F \in Form(\mathcal{L})$ si hace verdadera la fórmula en el sentido clásico definido anteriormente. `(esModeloFormula i f)` se verifica si la interpretación i es un modelo de la fórmula f.

```
-- | Por ejemplo,
--
-- >>> esModeloFormula [r]   ((p ∨ q) ∧ ((no q) ∨ r))
-- False
-- >>> esModeloFormula [p,r] ((p ∨ q) ∧ ((no q) ∨ r))
-- True
esModeloFormula :: Interpretacion -> FProp -> Bool
esModeloFormula i f = significado f i
```

Se denota por $Mod(F)$ al conjunto de modelos de F. La idea de la implementación en Haskell es la siguiente: En primer lugar, extraer los símbolos de F; posteriormente, calcular el conjunto potencia de los símbolos, que corresponde a las interpretaciones. Finalmente, devolver las interpretaciones que sean modelo de F.

`(simbolosPropForm f)` es el conjunto formado por todos los símbolos proposicionales que aparecen en la fórmula f.

```
-- | Por ejemplo,
--
-- >>> simbolosPropForm (p ∧ q → p)
-- [p,q]
simbolosPropForm :: FProp -> [FProp]
simbolosPropForm T          = []
simbolosPropForm F          = []
simbolosPropForm (Atom f)   = [(Atom f)]
simbolosPropForm (Neg f)    = simbolosPropForm f
simbolosPropForm (Conj f g) = simbolosPropForm f `union` simbolosPropForm g
simbolosPropForm (Disj f g) = simbolosPropForm f `union` simbolosPropForm g
simbolosPropForm (Impl f g) = simbolosPropForm f `union` simbolosPropForm g
simbolosPropForm (Equi f g) = simbolosPropForm f `union` simbolosPropForm g
```

`(interpretacionesForm f)` es la lista de todas las interpretaciones de la fórmula f.

```
-- |  Por ejemplo,
--
-- >>> interpretacionesForm (p ∧ q → p)
-- [[],[p],[q],[p,q]]
interpretacionesForm :: FProp -> [Interpretacion]
interpretacionesForm = subsequences . simbolosPropForm
```

`(modelosFormula f)` es la lista de todas las interpretaciones de la fórmula f que son modelo de la misma.

```
-- | Por ejemplo,
--
-- >>> modelosFormula ((p ∨ q) ∧ ((no q) ∨ r))
-- [[p],[p,r],[q,r],[p,q,r]]
modelosFormula :: FProp -> [Interpretacion]
modelosFormula f = [i | i <- interpretacionesForm f, esModeloFormula i f]
```

2.1.3. Validez, satisfacibilidad e insatisfacibilidad

Definición 2.1.6. Una fórmula F se dice válida si toda interpretación i de F es modelo de la fórmula. `(esValida f)` se verifica si la fórmula f es válida.

```
-- | Por ejemplo,
--
-- >>> esValida (p → p)
-- True
-- >>> esValida (p → q)
-- False
-- >>> esValida ((p → q) ∨ (q → p))
-- True
esValida :: FProp -> Bool
esValida f = modelosFormula f == interpretacionesForm f
```

Definición 2.1.7. Una fórmula F se dice insatisfacible si no existe ninguna interpretación i de F que sea modelo de la fórmula. La función `(esInsatisfacible f)` se verifica si la fórmula f es insatisfacible.

```
-- | Por ejemplo,
--
-- >>> esInsatisfacible (p ∧ (no p))
-- True
-- >>> esInsatisfacible ((p → q) ∧ (q → r))
-- False
esInsatisfacible :: FProp -> Bool
esInsatisfacible = null . modelosFormula
```

Definición 2.1.8. Una fórmula F se dice satisfacible si existe al menos una interpretación i de F que sea modelo de la fórmula.

La función `(esSatisfacible f)` se verifica si la fórmula f es satisfacible.

```
-- | Por ejemplo,
--
-- >>> esSatisfacible (p ∧ (no p))
-- False
-- >>> esSatisfacible ((p → q) ∧ (q → r))
-- True
esSatisfacible :: FProp -> Bool
esSatisfacible = not . null . modelosFormula
```

2.1.4. Bases de conococimiento

Definición 2.1.9. Una *base de conocimiento* o *Knowledge Basis* (*KB*) es un conjunto finito de fórmulas proposicionales. Para la implementación, se importa de forma cualificada la librería `Data.Set` como `S`. Esta es una práctica muy común con esta librería y lo único que ces que las funciones de dicha librería deben ir precedidas por `S.`. Por lo que se define el tipo de dato `KB` como:

```
type KB = S.Set FProp
```

Destacar que se denotará al lenguaje proposicional de K como $\mathcal{L}(K)$.

Antes de extender las definiciones anteriores a más de una fórmula, se implementarán dos funciones auxiliares. El objetivo de dichas funciones es obtener toda la casuística de interpretaciones posibles de un conjunto de fórmulas o *KB*.

(simbolosPropKB k) es el conjunto de los símbolos proposicionales de la base de conocimiento k.

```
-- | Por ejemplo,
--
-- >>> simbolosPropKB (S.fromList [p ∧ q → r, p → r])
-- [p,r,q]
simbolosPropKB :: KB -> [FProp]
simbolosPropKB = foldl (\acc f -> union acc (simbolosPropForm f)) []
```

(interpretacionesKB k) es la lista de las interpretaciones de la base de conocimiento k.

```
-- |  Por ejemplo,
--
-- >>> interpretacionesKB (S.fromList [p → q, q → r])
-- [[],[p],[q],[p,q],[r],[p,r],[q,r],[p,q,r]]
interpretacionesKB :: KB -> [Interpretacion]
interpretacionesKB = subsequences . simbolosPropKB
```

Definición 2.1.10. Análogamente al caso de una única fórmula, se dice que i es *modelo* de K ($i \vDash K$) si lo es de cada una de las fórmulas de K. `(esModeloKB i k)` se verifica si la interpretación i es un modelo de todas las fórmulas de la base de conocimiento k.

```
-- | Por ejemplo,
--
-- >>> esModeloKB [r] (S.fromList [q,no p ,r])
-- False
-- >>> esModeloKB [q,r] (S.fromList [q,no p ,r])
-- True
esModeloKB :: Interpretacion -> KB -> Bool
esModeloKB i = all (esModeloFormula i)
```

Al conjunto de modelos de K se le denota por $Mod(K)$. En Haskell, `(modelosKB k)` es la lista de modelos de la base de conocimiento k.

```
-- | Por ejemplo,
--
-- >>> modelosKB $ S.fromList [(p ∨ q) ∧ ((no q) ∨ r), q → r]
-- [[p],[p,r],[q,r],[p,q,r]]
-- >>> modelosKB $ S.fromList [(p ∨ q) ∧ ((no q) ∨ r), r → q]
-- [[p],[q,r],[p,q,r]]
modelosKB :: KB -> [Interpretacion]
modelosKB s = [i | i <- interpretacionesKB s, esModeloKB i s]
```

2.1.5. Consistencia e inconsistencia

La consistencia es una propiedad de las bases de conocimiento que se puede definir de dos maneras distintas:

Definición 2.1.11. Un conjunto de fórmulas se dice *consistente* si y sólo si tiene al menos un modelo.

La función `(esConsistente k)` se verifica si la base de conocimiento k es consistente.

```
-- |Por ejemplo,
--
-- >>> esConsistente $ S.fromList [(p ∨ q) ∧ ((no q) ∨ r), p → r]
-- True
-- >>> esConsistente $ S.fromList [(p ∨ q) ∧ ((no q) ∨ r), p → r, no r]
-- False
esConsistente :: KB -> Bool
esConsistente = not . null . modelosKB
```

Definición 2.1.12. Un conjunto de fórmulas se dice *inconsistente* si y sólo si no tiene ningún modelo.

La función `(esInconsistente k)` se verifica si la base de conocimiento *k* es inconsistente.

```
-- |Por ejemplo,
--
-- >>> esInconsistente $ S.fromList [(p ∨ q) ∧ ((no q) ∨ r), p → r]
-- False
-- >>> esInconsistente $ S.fromList [(p ∨ q) ∧ ((no q) ∨ r), p → r, no r]
-- True
esInconsistente :: KB -> Bool
esInconsistente = null . modelosKB
```

2.1.6. Consecuencia lógica

La consecuencia lógica es la relación entre las premisas y la conclusión de lo que se conoce como un argumento deductivamente válido. Esta relación es un concepto fundamental en la lógica y aparecerá con asiduidad en el desarrollo del trabajo.

Definición 2.1.13. Se dice que F es *consecuencia lógica* de K ($K \models F$) si todo modelo de K lo es a su vez de F, es decir, $Mod(K) \subseteq Mod(F)$. Equivalentemente, $K \models F$ si no es posible que las premisas sean verdaderas y la conclusión falsa.

La función `(esConsecuencia k f)` se verifica si la fórmula proposicional f es consecuencia lógica de la base de conocimiento o conjunto de fórmulas k.

```
-- |Por ejemplo,
--
-- >>> esConsecuencia (S.fromList [p → q, q → r]) (p → r)
-- True
-- >>> esConsecuencia (S.fromList [p]) (p ∧ q)
-- False
esConsecuencia :: KB -> FProp -> Bool
esConsecuencia k f =
  null [i | i <- interpretacionesKB (S.insert f k)
          , esModeloKB i k
          , not (esModeloFormula i f)]
```

Con el objetivo de hacer más robusto el sistema se implementarán dos propiedades de la relación de *ser consecuencia* en lógica proposicional. Dichas propiedades se comprobarán con la librería `QuickCheck` propia del lenguaje Haskell. Es importante saber

que estas evaluaciones de propiedades con quickCheck son sólo comprobaciones de que la propiedad se cumple para una batería de ejemplos. En ningún caso se puede confiar en que dicha propiedad se cumple el 100necesaria una verificación formal, un problema más costoso y en ningún caso trivial.

Proposición 2.1.14. Una fórmula f es válida si y sólo si es consecuencia del conjunto vacío.

```
-- |
-- >>> quickCheck prop_esValida
-- +++ OK, passed 100 tests.
prop_esValida :: FProp -> Bool
prop_esValida f =
   esValida f == esConsecuencia S.empty f
```

Proposición 2.1.15. Una fórmula f es consecuencia de un conjunto de fórmulas k si y sólo si dicho el conjunto formado por k y $\neg f$ es inconsistente.

```
-- |
-- >>> quickCheck prop_esConsecuencia
-- +++ OK, passed 100 tests.
prop_esConsecuencia :: KB -> FProp -> Bool
prop_esConsecuencia k f =
   esConsecuencia k f == esInconsistente (S.insert (Neg f) k)
```

De manera natural se extiende la definición de *ser consecuencia* a bases de conocimiento en lugar de fórmulas, preservando la misma notación. La función `(esConsecuenciaKB k k')` se verifica si todas las fórmulas del conjunto k' son consecuencia de las del conjunto k.

```
-- |Por ejemplo,
--
-- >>> esConsecuenciaKB (S.fromList [p → q, q → r]) (S.fromList [p → q, p → r])
-- True
-- >>> esConsecuenciaKB (S.fromList [p]) (S.fromList [p ∧ q])
-- False
esConsecuenciaKB :: KB -> KB -> Bool
esConsecuenciaKB k = all (esConsecuencia k)
```

2.1.7. Equivalencia

Definición 2.1.16. Sean F y G dos fórmulas proposicionales, se dice que son equivalentes ($F \equiv G$) si tienen el mismo significado lógico, es decir, si tienen el mismo valor de verdad en todas sus interpretaciones.

La función `(equivalentes f g)` se verifica si las fórmulas proposicionales son equivalentes.

```
-- |Por ejemplo,
--
-- >>> equivalentes (p → q) (no p ∨ q)
-- True
-- >>> equivalentes (p) (no (no p))
-- True
equivalentes :: FProp -> FProp -> Bool
equivalentes f g = esValida (f ↔ g)
```

Definición 2.1.17. Dadas K y K' dos bases de conocimiento, se dice que son equivalentes ($K \equiv K'$) si $K \vDash K'$ y $K' \vDash K$.

La función `(equivalentesKB k k')` se verifica si las bases de conocimiento k y k' son equivalentes.

```
-- |Por ejemplo,
--
-- >>> equivalentesKB (S.fromList [p → q,r ∨ q]) (S.fromList [no p ∨ q, q ∨ r])
-- True
-- >>> equivalentesKB (S.fromList [p ∧ q]) (S.fromList [q,p])
-- True
equivalentesKB :: KB -> KB -> Bool
equivalentesKB k k' = esConsecuenciaKB k k' && esConsecuenciaKB k' k
```

La siguiente propiedad comprueba si las definiciones implementadas anteriormente son iguales en el caso de una única fórmula en la base de conocimiento.

Proposición 2.1.18. Sean F y g dos fórmulas proposicionales, son equivalentes como fórmulas si y sólo si lo son como bases de conocimiento.

```
-- |
-- >>> quickCheck prop_equivalentes
-- +++ OK, passed 100 tests.
prop_equivalentes :: FProp -> FProp -> Bool
prop_equivalentes f g =
  equivalentes f g == equivalentesKB (S.singleton f) (S.singleton g)
```

2.1.8. Retracción conservativa

Dada una base de conocimiento resulta interesante estudiar qué fórmulas se pueden deducir de la misma o incluso buscar si existe otra manera de expresar el conocimiento, en definitiva otras fórmulas, de las que se deduzca exactamente la misma información. A esta relación entre bases de conocimiento se le conoce con el nombre de extensión:

Definición 2.1.19. Sean K y K' bases de conocimiento, se dice que K es una *extensión* de K' si $\mathcal{L}(K') \subseteq \mathcal{L}(K)$ y

$$\forall F \in Form(\mathcal{L}(K')) \;\; [K' \vDash F \Rightarrow K \vDash F]$$

Definición 2.1.20. Sean K y K' bases de conocimiento, se dice que K es una *extensión conservativa* de K' si es una extensión tal que toda consecuencia lógica de K expresada en el lenguaje $\mathcal{L}(K')$, es también consecuencia de K',

$$\forall F \in Form(\mathcal{L}(K')) \;\; [K \vDash F \Rightarrow K' \vDash F]$$

es decir, K extiende a K' pero no se añade nueva información expresada en términos de $\mathcal{L}(K')$.

Definición 2.1.21. K' es una *retracción conservativa* de K si y sólo si K es una extensión conservativa de K'.

Dado $\mathcal{L}' \subseteq \mathcal{L}(K)$, siempre existe una retracción conservativa de K al lenguaje $\mathcal{L}'$. La *retracción conservativa canónica* de K a $\mathcal{L}'$ se define como:

$$[K, \mathcal{L}'] = \{F \in Form(\mathcal{L}') : K \vDash F\}$$

Esto es, $[K, \mathcal{L}']$ es el conjunto de $\mathcal{L}'$-fórmulas que son consecuencia de K. De hecho, cualquier retración conservativa sobre $\mathcal{L}'$ es equivalente a $[K, \mathcal{L}']$.

2.2. El anillo $\mathbb{F}_2[\mathbf{x}]$

Para una natural interpretación algebraica de la lógica, el marco de trabajo escogido es el anillo $\mathbb{F}_2[\mathbf{x}]$. Con la idea de facilitar la identificación entre variables proposicionales e incógnitas, se fija la notación de forma que a una variable proposicional p_i (ó p) le corresponde la incógnita x_i (ó x_p).

La notación usada en los polinomios es la estándar. Dado $\alpha = (\alpha_1, ..., \alpha_n) \in \mathbb{N}^n$, se define $|\alpha| := \text{máx}\{\alpha_1, ..., \alpha_n\}$ y x^α es el monomio $x_1^{\alpha_1}...x_n^{\alpha_n}$.

Definición 2.2.1. El *grado* de $a(\mathbf{x}) \in \mathbb{F}_2[\mathbf{x}]$ es

$$deg_\infty(a(\mathbf{x})) := \text{máx}\{|\alpha| : \mathbf{x}^\alpha \text{ es un monomio de } a\}$$

Si $deg(a(x)) \leq 1$, el polinomio $a(\mathbf{x})$ se denomina *fórmula polinomial*. El grado de $a(x)$ respecto una variable x_i se denota por $deg_i(a(\mathbf{x}))$.

A continuación se tratará de implementar $\mathbb{F}_2[\mathbf{x}]$ en Haskell, así como algunas operaciones polinomiales que serán necesarias más adelante.

2.2.1. Polinomios en Haskell

Si se quiere trabajar con polinomios en Haskell, no es necesario "hacer tabla rasa" y tratar de implementarlo todo desde el principio. Parafraseando a Newton,

> *Si he conseguido ver más lejos es porque me he aupado en hombros de gigantes*
>
> Isaac Newton

así que conviene apoyarse en la multitud de librerías existentes de la comunidad Haskell. Aunque, continuando con la metáfora de Newton, no es fácil saber qué gigante es el más alto si miramos desde el suelo. Por tanto, es necesario un estudio de las distintas librerías, a fin de escoger la que se adecúe en mayor medida a las necesidades de este proyecto.

Seguidamente se comentarán los detalles más relevantes de las distintas librerías estudiadas, centrando la atención en los detalles prácticos como la utilidad y la eficiencia.

Cryptol

Cryptol es un lenguaje de programación especialmente desarrollado para implementar algoritmos criptográficos. Su ventaja es su similitud con el lenguaje matemático respecto a un lenguaje de propósito general. Está escrito en Haskell, siendo un trabajo muy pulido con un tutorial muy completo.

Sin embargo, no resulta intuitivo aislar la librería de polinomios por lo que para realizar las pruebas se ha optado por cargar todo el paquete apoyándonos en la documentación.

Esta librería sólo trabaja con polinomios en una única variable. Por ejemplo, los polinomios de $\mathbb{F}_2[\mathbf{x}]$ con grado menor o igual que 7 se codifican mediante un vector de unos y ceros donde se almacenan los coeficientes de cada x^i con $i \leq 7$. Debido a este tipo de codificación no es trivial extenderlo a más de una variable.

The polynomial package

El módulo `MAth.Polynomial` de esta librería implementa la aritmética en una única variable, debiendo especificar con qué orden monomial se quiere trabajar. Por el contrario, no es necesario dar como entrada el cuerpo en el que se defina la K-álgebra sino que construye los polinomios directamente de una lista ordenada de coeficientes. Por tanto, hereda el tipo de dichos elementos de la lista (`[a]`), formando lo que se denomina como tipo polinomio (`Poly a`).

La librería, poco documentada, se centra en definir distintos tipos de polinomios de la literatura matemática clásica como los polinomios de Hermite, de Bernouilli, las bases de Newton o las bases de Lagrange.

ToySolver.Data.Polynomial

Este módulo se enmarca en una librería que trata de implementar distintos procedimientos y algoritmos para resolver varios problemas de decisión. Debido a que su función es únicamente servir como auxiliar para otros módulos el código no está comentado, lo que dificulta su uso.

La implementación es muy completa, incluye multitud de operaciones y procedimientos de polinomios y $\mathcal{K}$-álgebras. Por ejemplo, permite construir polinomios sobre cuerpos finitos. Sin embargo, la estructura de dato de los polinomios no es intuitiva para su uso.

SBV

El parecido a la librería de Cryptol es notable, por lo que se encuentra con inconvenientes similares.

Gröebner bases in Haskell

El objetivo principal de esta librería es, tal y como su nobre indica, el cálculo de bases de Gröebner mediante operaciones con ideales.

En lo que respecta al código, al tipo de dato polinomio se le debe especificar el Anillo de coeficientes así como el orden monomial. Además, se deben declarar desde el principio las variables que se quieren usar.

La documentación está muy detallada aunque el autor comenta que se prioriza la claridad del código y el rigor matemático ante la efieciencia.

HaskellForMaths

El módulo de polinomios es `Math.CommutativeAlgebra.Polynomial`. Al igual que en la librería anterior, se permite escoger el orden monomial entre los tres más usuales (lexicográfico, graduado lexicográfico y graduado reverso lexicográfico) e incluso definir otros nuevos.

También se debe dar como entrada el cuerpo sobre el que se trabajará. Para ello se incorpora un módulo especifico llamado `MAth.Core.Field` en el que están implementados los cuerpos más comunes, como los números racionales o los cuerpos finitos.

Destacar que el tipo de dato polinomio tiene estructura vectorial, donde la base es cada monomio que exista (combinaciones de todas las variables) y el número en la posición i-ésima del vector corresponde con el coeficiente del monomio i-ésimo (según el orden especificado) del polinomio.

Las operaciones básicas de los polinomios están ya implementadas de forma eficiente, destacando la multiplicación, desarrollada en un módulo aparte se apoya en otra librería de productos tensoriales.

Esta librería responde en líneas generales a las necesidades del proyecto ya que es modular, el código está documentado y la mayoría de algoritmos son eficientes. Por dichas razones se escoge esta librería como auxiliar en el proyecto a desarrollar.

2.2.2. Introducción a HaskellForMaths

Se muestran a continuación diversos ejemplos de funciones de `HaskellForMaths` que aparecerán de forma recurrente en el resto del trabajo.

```
module Haskell4Maths
    ( F2
    , MonImpl(..)
    , Vect(..)
    , linear
    , zerov
    , Lex (..)
    , Glex
    , Grevlex
    , var
    , mindices
    , lm
    , lt
    , eval
    , (%%)
    , vars) where

import Math.Core.Field
import Math.Algebras.VectorSpace
import Math.CommutativeAlgebra.Polynomial
```

Math.Core.Field

En este módulo se definen el cuerpo $\mathbb{Q}$ de los racionales y los cuerpos finitos o de Galois: $\mathbb{F}_2$, $\mathbb{F}_3$, $\mathbb{F}_4$, $\mathbb{F}_5$, $\mathbb{F}_7$, $\mathbb{F}_8$, $\mathbb{F}_9$, $\mathbb{F}_{11}$, $\mathbb{F}_{13}$, $\mathbb{F}_{16}$, $\mathbb{F}_{17}$, $\mathbb{F}_{19}$, $\mathbb{F}_{23}$, $\mathbb{F}_{25}$.

Veamos unos ejemplos de cómo se trabaja con los números racionales:

```
-- |
-- >>> (7/14 :: Q)
-- 1/2
-- >>> (0.6 :: Q)
-- 3/5
-- >>> (2.3 + 1/5 * 4/7) :: Q
-- 169/70
```

Para este trabajo, el cuerpo que nos interesa es $\mathbb{F}_2$, cuyos elementos pertenecen a la lista `f2`:

```
-- |
-- >>> f2
-- [0,1]
```

Y cuyas operaciones aritméticas se definen de forma natural:

```
-- |
-- >>> (2 :: F2)
-- 0
-- >>> (1 :: F2) + (3 :: F2)
-- 0
-- >>> (7 :: F2) * (1 :: F2)
-- 1
```

Además, cuenta con la función auxiliar `fromInteger` para transformar números de tipo `Integer` en el tipo `Fp` dónde `p` es número de elementos del cuerpo:

```
-- |
-- >>> (fromInteger (12345 :: Integer)) :: F2
-- 1
```

Math.Algebras.VectorSpace

En este módulo se define el tipo y las operaciones de los espacios de k-vectores libres sobre una base b, con k un cuerpo, de la siguiente manera:

```
newtype Vect k b = V [(b,k)]
```

Intuitivamente, un vector es una lista de pares donde la primera coordenada es un elemento de la base y la segunda coordenada el coeficiente de dicho elemento. Notar que el coeficiente pertenece al cuerpo k, que se debe especificar en el tipo.

También destacar que este nuevo tipo tiene las instancias `Eq`, `Ord` y `Show` además de estar definida la suma y la multiplicación de vectores (en función de la base y los coeficientes).

La función `(zerov)` representa al vector cero independientemente del cuerpo k y la base b. Por ejemplo,

```
-- |
-- >>> zerov :: (Vect Q [a])
-- 0
-- >>> zerov :: (Vect F2 F3)
-- 0
```

Una función que conviene destacar es la función `(linear f v)`, que es una función lineal entre dos espacios vectoriales ($A =$ `Vect k a` y $B =$ `Vect k b`). La función `(f :: a ->Vect k b)` va de los elementos de la base de A a B. Por lo que `(linear)` es muy útil si se necesita transformar vectores de forma interna.

Math.ConmutativeAlgebra.Polynomial

En el siguiente módulo se define el álgebra conmutativa de los polinomios sobre el cuerpo k. Los polinomios se representan como el espacio de k-vectores libres con los monomios como su base.

Para poder trabajar con los polinomios es necesario especificar un orden monomial. En este módulo están implementados los tres más comunes: el lexicográfico (`Lex`), el graduado lexicográfico (`Glex`) y el graduado reverso lexicográfico (`Grevlex`). Asimismo, es posible añadir otros nuevos en caso de que fuera necesario.

La función `(var v)` crea una variable en espacio vectorial de polinomios. Por ejemplo, si se quiere trabajar en $\mathbb{Q}[x, y, z]$, debemos definir:

```
-- |
-- >>> [x,y,z] = map var ["x","y","z"] :: [GlexPoly Q String]
```

Destacar que, en general, es necesario proporcionar los tipos de datos de forma que el compilador sepa qué cuerpo y qué orden monomial usar. A continuación se mostrarán diversos ejemplos de operaciones entre polinomios, variando el orden monomial y el cuerpo.

```
-- |
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly Q String]
-- >>> x^2+x*y+x*z+x+y^2+y*z+y+z^2+z+1
-- x^2+xy+xz+x+y^2+yz+y+z^2+z+1
-- >>> [x,y,z] = map var ["x","y","z"] :: [GlexPoly Q String]
-- >>> x^2+x*y+x*z+x+y^2+y*z+y+z^2+z+1
-- x^2+xy+xz+y^2+yz+z^2+x+y+z+1
-- >>> [x,y,z] = map var ["x","y","z"] :: [GrevlexPoly Q String]
-- >>> x^2+x*y+x*z+x+y^2+y*z+y+z^2+z+1
-- x^2+xy+y^2+xz+yz+z^2+x+y+z+1
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly Q String]
-- >>> (x+y+z)^2
-- x^2+2xy+2xz+y^2+2yz+z^2
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly F2 String]
-- >>> (x+y+z)^2
-- x^2+y^2+z^2
```

Como se mencionó anteriormente la base del espacio vectorial que es un polinomio, está formada por monomios. El tipo de dato monomio está formado por un coeficiente i y una lista de pares. Un ejemplo de monomio es:

```
-- |
-- >>> monomio
-- x^2y
monomio :: MonImpl [Char]
monomio = (M 1 [("x",2),("y",1)])
```

Dichos pares se obtienen mediante la función `(mindices m)` y representan los elementos canónicos que forman cada monomio de la base, así como su exponente:

```
-- |
-- >>> mindices monomio
-- [("x",2),("y",1)]
```

En este módulo también se implementan tres funciones auxiliares que resultarán de gran utilidad más adelante. La función auxiliar que resulta más natural implementar cuando se trabaja con polinomios y que además goza de una gran importancia es `(vars p)`. Ésta devuelve la lista de variables que aparecen en el polinomio p.

```
-- |Por ejemplo,
--
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly F2 String]
-- >>> vars (x*z*y+y*x^2+z^4)
-- [x,y,z]
```

La segunda función auxiliar es `(lt m)` (término líder) que devuelve un par `(m,i)` donde m es el monomio líder e i su coeficiente ($i \in k$).

```
-- |
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly F2 String]
-- >>> lt (x*z*y+y*x^2+z^4)
-- (x^2y,1)
```

La tercera es la función `(lm p)` que devuelve el monomio líder del polinomio p:

```
-- |
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly F2 String]
-- >>> lm (x*z*y+y*x^2+z^4)
-- x^2y
```

Otra función natural es `(eval p vs)`, que evalúa el polinomio `p` en el punto descrito por `vs`, siendo ésta una lista de pares variable-valor.

```
-- |
-- >>> [x,y] = map var ["x","y"] :: [LexPoly F2 String]
-- >>> eval (x^2+y^2) [(x,1),(y,0)]
-- 1
```

Por último, destacar las función `(p %% xs)` que calcula la reducción del polinomio p respecto de los polinomios de la lista xs.

```
-- |
-- >>> [x,y,z] = map var ["x","y","z"] :: [LexPoly F2 String]
-- >>> (x^2+y^2) %% [x^2+1]
-- y^2+1
```

2.2.3. $\mathbb{F}_2[\mathbf{x}]$ en Haskell

En esta subsección se realizarán las implementaciones necesarias para poder trabajar en Haskell con $\mathbb{F}_2[\mathbf{x}]$, definiendo el módulo `F2`:

```
{-# LANGUAGE FlexibleInstances, FlexibleContexts #-}
module F2 ( VarF2
          , PolF2
          , unbox ) where
```

Que importa las librerías:

```
import Haskell4Maths ( Vect
                     , Lex
                     , F2
                     , var)
import Test.QuickCheck ( Arbitrary
                       , Gen
                       , arbitrary
                       , vectorOf
                       , choose
                       , quickCheck)
```

El primer paso tras el análisis realizado sobre la librería HaskellForMaths, es definir el tipo de dato que representa $\mathbb{F}_2[\mathbf{x}]$ (`PolF2`) , así como sus variables (`VarF2`). Para ello se representarán los polinomios en un espacio vectorial cuya base son los monomios (`Lex String`) y sus coeficientes están en $\mathbb{F}_2$ (`F2`).

```
newtype VarF2 = Box (Vect F2 (Lex String))
  deriving (Eq, Ord)

type PolF2 = Vect F2 (Lex String)
```

Notar que el tipo de las variables es simplemente un cambio de nombre respecto a los polinomios que ha sido metido dentro del constructor `Box`. Este artificio es necesario ya que no se pueden declarar instancias repetidas (como se hará a continuación) sobre un mismo tipo de dato aunque tengan nombres distintos.

Sin embargo, es necesario definir la función auxiliar `(unbox x)` que saca a x de la mónada `Var`:

```
unbox :: VarF2 -> PolF2
unbox (Box x) = x
```

Para poder mostrar por consola las variables de forma estética; es decir, sin mostrar el constructor `Box`, declaramos la instancia `Show`:

```
instance Show VarF2 where
  show = show . unbox
```

Para poder definir propiedades que involucren a estos tipos de datos y comprobarlas con `QuickCheck` es necesario añadir la instacia `Arbitrary`, así como definir generadores de dichos tipos. Se comenzará por el tipo `VarF2` ya que servirá como apoyo para el de los polinomios:

```
instance Arbitrary VarF2 where
  arbitrary = varGen
```

La función `varGen` es un generador de variables:

```
varGen :: Gen VarF2
varGen = do
  n <- choose ((1::Int),100)
  return (Box (var ('x':(show n))))
```

Se declara la instancia `Arbitrary` para el tipo de dato de los polinomios:

```
instance Arbitrary PolF2 where
  arbitrary = polGen
```

El generador aleatorio de polinomios seguirá la siguiente estructura: en primer lugar se generarán aleatoriamente pares de variable-exponente, con los que se formarán monomios. Por último, se suman éstos para obtener los polinomios. En Haskell, `varExpGen` es el generador de pares:

```
varExpGen :: Gen (PolF2,Int)
varExpGen = do
  Box x <- varGen
  i <- choose ((1::Int),5)
  return $ (x,i)
```

El generador de monomios `monGen` se implementa de la siguiente forma:

```
monGen :: Gen PolF2
monGen = do
  n <- choose ((1::Int),5)
  xs <- vectorOf n varExpGen
  return $ product [ x ^ i | (x,i) <- xs]
```

Finalmente, el generador de polinomios `polGen` se implementa tal y como y sigue:

```
polGen :: Gen PolF2
polGen = do
  n <- choose ((1::Int),5)
  xs <- vectorOf n monGen
  return $ sum xs
```

Propiedades de $\mathbb{F}_2[\mathbf{x}]$

Es importante comprobar que el nuevo tipo de dato que hemos definido cumple las propiedades básicas. Ya que el trabajo se basa en este tipo de dato y sus propiedades. Se comprobarán las propiedades de la suma y del producto de polinomios de $\mathbb{F}_2[\mathbf{x}]$:

La suma de polinomios es conmutativa,

$$\forall p, q \in \mathbb{F}_2[\mathbf{x}](p + q = q + p)$$

En Haskell:

```
-- |
-- >>> quickCheck prop_suma_conmutativa
-- +++ OK, passed 100 tests.
prop_suma_conmutativa :: PolF2 -> PolF2 -> Bool
prop_suma_conmutativa p q = p+q == q+p
```

La suma de polinomios es asociativa:

$$\forall p, q, r \in \mathbb{F}_2[\mathbf{x}](p + (q + r) = (p + q) + r)$$

En Haskell:

```
-- |
-- >>> quickCheck prop_suma_asociativa
-- +++ OK, passed 100 tests.
prop_suma_asociativa :: PolF2 -> PolF2 -> PolF2 -> Bool
prop_suma_asociativa p q r = p+(q+r) == (p+q)+r
```

El cero es el elemento neutro de la suma de polinomios:

$$\forall p \in \mathbb{F}_2[\mathbf{x}] p + 0 = 0 + p = p$$

En Haskell:

```
-- |
-- >>> quickCheck prop_suma_neutro
-- +++ OK, passed 100 tests.
prop_suma_neutro :: PolF2 -> Bool
prop_suma_neutro p = (p + 0 == p) && (0 + p == p)
```

Todo polinomio es simétrico de sí mismo respecto a la suma:

$$\forall p \in \mathbb{F}_2[\mathbf{x}] : p + p = 0$$

En Haskell:

```
-- |
-- >>> quickCheck prop_suma_simetrico
-- +++ OK, passed 100 tests.
prop_suma_simetrico :: PolF2 -> Bool
prop_suma_simetrico p = p+p == 0
```

La multiplicación de polinomios es conmutativa:

$$\forall p, q \in \mathbb{F}_2[\mathbf{x}](p * q = q * p)$$

En Haskell:

```
-- |
-- >>> quickCheck prop_prod_conmutativa
-- +++ OK, passed 100 tests.
prop_prod_conmutativa :: PolF2 -> PolF2 -> Bool
prop_prod_conmutativa p q = p*q == q*p
```

El producto es asociativo:

$$\forall p, q, r \in \mathbb{F}_2[\mathbf{x}](p * (q * r) = (p * q) * r)$$

En Haskell:

```
-- |
-- >>> quickCheck prop_prod_asociativo
-- +++ OK, passed 100 tests.
prop_prod_asociativo :: PolF2 -> PolF2 -> PolF2 -> Bool
prop_prod_asociativo p q r = p*(q*r) == (p*q)*r
```

El 1 es el elemento neutro de la multiplicación de polinomios:

$$\forall p \in \mathbb{F}_2[\mathbf{x}] p * 1 = 1 * p = p$$

En Haskell:

```
-- |
-- >>> quickCheck prop_prod_neutro
-- +++ OK, passed 100 tests.
prop_prod_neutro :: PolF2 -> Bool
prop_prod_neutro p = (p * 1 == p) && (1 * p == p)
```

Distributividad del producto respecto la suma:

$$\forall p, q, r \in \mathbb{F}_2[\mathbf{x}](p * (q + r) = p * q + p * r)$$

En Haskell:

```
-- |
-- >>> quickCheck prop_distributiva
-- +++ OK, passed 100 tests.
prop_distributiva :: PolF2 -> PolF2 -> PolF2 -> Bool
prop_distributiva p q r = p*(q+r) == (p*q)+(p*r)
```

2.3. Transformaciones entre fórmulas y polinomios

La traducción o transformación de la lógica proposicional en álgebra polinomial viene dada por [4] y se ilustra en la figura 2.1.

La idea principal es que las fórmulas se pueden ver como polinomios sobre fórmulas atómicas cuando éstas están expresadas en términos de las conectivas booleanas *o exclusivo* e *y*; así como de las constantes 1 y 0, que equivalen a los conceptos de *Verdad* y *Falsedad*, respectivamente. Las operaciones básicas de suma y multiplicación se corresponden con las conectivas booleanas *o exclusivo* e *y*, respectivamente. Por tanto, la función $P : Form(\mathcal{L} \to \mathbb{F}_2[\mathbf{x}])$ que aparece en la página 11 se define por:

- $P(\bot) = 0, P(p_i) = x_i, P(\neg F) = 1 + P(F)$
- $P(F_1 \wedge F_2) = P(F_1) \cdot P(F_2)$
- $P(F_1 \vee F_2) = P(F_1) + P(F_2) + P(F_1) \cdot P(F_2)$
- $P(F_1 \to F_2) = 1 + P(F_1) + P(F_1) \cdot P(F_2)$
- $P(F_1 \leftrightarrow F_2) = 1 + P(F_1) + P(F_2)$

En resumen, consiste en hacer corresponder las fórmulas falsas con el valor cero y las verdaderas con el uno. Por ejemplo, si una fórmula $(p_1 \wedge p_2)$ dada una valoración $(p_1 = True, p_2 = True)$ es verdadera, su correspondiente polinomio $(x_1 * x_2)$ teniendo en cuenta la interpretación $(x_1 = 1, x_2 = 1)$ debe valer uno $(1 + 1 =_{\mathbb{F}_2} 1)$. La implementación se hará en el módulo `Transformaciones`:

```
module Transformaciones
    ( phi
    , theta
    , proyeccion) where

import Logica
import Haskell4Maths (Vect(..)
                     , var
                     , vars
                     , mindices
                     , eval
                     , linear
                     , (%%))
import F2 (PolF2)

import Test.QuickCheck (quickCheck
                       , maxSize
                       , quickCheckWith
                       , stdArgs)
```

La función encargada de hacer dicha traducción es la función `tr`., que equivale a la función P descrita anteriormente. Ésta recibe una fórmula proposicional del tipo `FProp` y devuelve un polinomio con coeficientes en $\mathbb{F}_2$, es decir, del tipo `PolF2`.

```
-- | Por ejemplo,
--
-- >>> let [p1,p2] = [Atom "p1",Atom "p2"]
-- >>> tr p1
-- x1
-- >>> tr (p1 ∧ p2)
-- x1x2
-- >>> tr (p ∧ (q ∨ r))
-- qrx+qx+rx
tr :: FProp -> PolF2
tr T               = 1
tr F               = 0
tr (Atom ('p':xs)) = var ('x':xs)
tr (Atom xs)       = var xs
tr (Neg a)         = 1 + tr a
tr (Conj a b)      = tr a * tr b
tr (Disj a b)      = a' + b' + a' * b'
                     where a' = tr a
                           b' = tr b
tr (Impl a b)      = 1 + a' + a' * tr b
                     where a' = tr a
tr (Equi a b)      = 1 + tr a + tr b
```

Para la transformación recíproca (de polinomios a fórmulas) se usará la función $\Theta : \mathbb{F}_2[\mathbf{x}] \to Form(\mathcal{L})$ definida por:

- $\Theta(0) = \bot$
- $\Theta(1) = \top$
- $\Theta(x_i) = p_i$
- $\Theta(a + b) = \neg(\Theta(a) \leftrightarrow \Theta(b))$
- $\Theta(a \cdot b) = \Theta(a) \wedge \Theta(b)$

La función (`theta p`) transforma el polinomio `p` en la fórmula proposicional que le corresponde según la definición anterior.

```
-- | Por ejemplo,
--
-- >>> let [x1,x2] = [var "x1", var "x2"] :: [PolF2]
-- >>> theta 0
-- ⊥
-- >>> theta (x1*x2)
-- (p1 ∧ p2)
-- >>> theta (x1 + x2 +1)
-- ¬(p1 ↔ ¬(p2 ↔ ⊤))

theta :: PolF2 -> FProp
theta 0          = F
theta 1          = T
theta (V [m])    = (theta' . mindices . fst) m
theta (V (x:xs)) = no (((theta' . mindices . fst) x) ↔ (theta (V xs)))

theta' :: [(String, t)] -> FProp
theta' []                = T
theta' [(('x':v),i)]     = Atom ('p':v)
theta' ((('x':v),i):vs) = Conj (Atom ('p':v)) (theta' vs)
theta' [(v,i)]           = Atom v
theta' ((v,i):vs)        = Conj (Atom v) (theta' vs)
```

A continuación se definen dos propiedades que deben cumplir las funciones `tr` y `theta`.

Proposición 2.3.1. Sea f una fórmula proposicional cualquiera, $\Theta(P(f))$ es equivalente a f. La implementación de esta propiedad es:

```
-- |
-- >>> quickCheckWith (stdArgs {maxSize = 30}) prop_theta_tr
-- +++ OK, passed 100 tests.
prop_theta_tr :: FProp -> Bool
prop_theta_tr f = equivalentes (theta (tr f)) f
```

Notar que a la hora de comprobar la propiedad anterior se ha acotado el tamaño máximo de las fórmulas proposicionales ya que en caso contrario se demora demasiado en ejecutarse.

Se define ahora la propiedad inversa:

Proposición 2.3.2. Sea p un polinomio de $\mathbb{F}_2[x]$, $P(\Theta(p)) = p$. Cuya implementación es:

```
prop_tr_theta :: PolF2 -> Bool
prop_tr_theta p = tr (theta p) == p
```

Sin embargo, al ejecutarlo nos devuelve `Failed`:

```
-- >>> quickCheck prop_tr_theta
--  *** Failed! Falsifiable (after 1 test):
--  x29^3x87^5+x30x74^2x80^4+x38^5x62^2
```

Esto se debe a los exponentes, que se pierden al transformar el polinomio en una fórmula proposicional. Por tanto, al reescribir el polinomio, éste es idéntico pero sin exponentes. Se tratará esto en la siguiente subsección y se comprobará que realmente ambos polinomios son iguales al estar en $\mathbb{F}_2[x]$.

2.3.1. Correspondencia entre valoraciones y puntos en $\mathbb{F}_2^n$

El comportamiento similar como funciones de la fórmula F y su traducción polinomial $P(F)$ son la base entre la semántica y las funciones polinomiales. Con idea de esclarecer qué se quiere decir con esto se explicará qué quiere decir *un comportamiento similar*:

- *De valoraciones a puntos*: Dada una valoración o interpretación $v : \mathcal{L} \to \{0,1\}$ el valor de verdad de F respecto de v coincide con el valor de $P(F)$ en el punto $o_v \in \mathbb{F}^n$ definido por los valores dados por v; $(o_v)_i = v(p_i)$. Es decir, para cada fórmula $F \in Form(\mathcal{L})$,

$$v(F) = P(F)((o_v)_1, \ldots, (o_v)_n)$$

- *De puntos a valoraciones*: Cada $o = (o_1 \ldots o_n) \in \mathbb{F}_2^n$ define una valoración v_o de la siguiente forma:

$$v_o(p_i) = 1 \text{ si y sólo si } o_i = 1$$

Entonces,

$$v_o \vDash F \quad \Leftrightarrow \quad P(F)(o_v) + 1 = 0 \quad \Leftrightarrow o_v \in \mathcal{V}(1 + P(F))$$

donde $V(.)$ se define como: Dado $a(\mathbf{x}) \in \mathbb{F}_2[x]$,

$$V(a(\mathbf{x})) = \{o \in \mathbb{F}_2^n : a(o) = 0\}$$

Por consiguiente, hay dos transformaciones entre el conjunto de interpretaciones o valoraciones y los puntos de $\mathbb{F}_2^n$, que definen biyecciones entre modelos de F y puntos de la variedad algebraica $\mathcal{V}(1+P(F))$;

$$\begin{array}{ccc} Mod(F) \to \mathcal{V}(1+P(F)) & & \mathcal{V}(1+P(F)) \to Mod(F) \\ v \to o_v & & o \to v_o \end{array}$$

Por ejemplo, consideremos la fórmula $F = p_1 \to p_2 \wedge p_3$. El polinomio asociado es $P(F) = 1 + x_1 + x_1x_2x_3$. La valoración $v = \{(p_1,0),(p_2,1),(p_3,0)\}$ es modelo de F e induce el punto $o_v = (0,1,0) \in \mathbb{F}_2^3$ que a su vez pertenece a $\mathcal{V}(1+P(F)) = \mathcal{V}(x_1 + x_1x_2x_3)$.

```
-- |
-- >>> [p1,p2,p3] = map Atom ["p1","p2","p3"]
-- >>> f = p1 → p2 ∧ p3
-- >>> tr f
-- x1x2x3+x1+1
-- >>> esModeloFormula [p3] f
-- True
-- >>> eval (1+(tr f)) [(var "x1",0),(var "x2",1),(var "x3",0)]
-- 0
```

2.3.2. Proyección polinomial

Consideremos ahora la parte derecha de la figura 2.1. Para simplificar la relación entre la semántica de la lógica proposicional y la geometría sobre cuerpos finitos se usará la función:

$$\Phi : \mathbb{F}_2[\mathbf{x}] \to \mathbb{F}_2[\mathbf{x}]$$
$$\Phi(\sum_{\alpha \in I} \mathbf{x}^{\alpha}) := \sum_{\alpha \in I} \mathbf{x}^{sg(\alpha)}$$

siendo $sg(\alpha) := (\delta_1, \ldots, \delta_n)$ donde δ_i es 0 si $\alpha_i = 0$ y 1 en cualquier otro caso.

En la librería `HaskellForMaths` ya existe una función que calcula el representante de un polinomio un el grupo cociente por un ideal. Esta es la función `(%%)`. Sin embargo, ya que la búsqueda de la eficiencia es una máxima en este trabajo, se aprovechará el hecho de que calcular dicho representante equivale a reemplazar cada ocurrencia de x_i^k (con $k \in \mathbb{N}$) por x_i.

La función `(phi p)` calcula el representante de menor grado del polinomio p en el grupo cociente $\mathbb{F}_2[\mathbf{x}]/_{\mathbb{I}_2}$, siendo $\mathbb{I}_2 = \{x_1 + x_1^2, ..., x_n + x_n^2\}$ y $n \in \mathbb{N}$ el número total de variables.

```
-- | Por ejemplo,
-- >>> [x1,x2] = [var "x1", var "x2"] :: [PolF2]
-- >>> phi (1+x1+x1^2*x2)
-- x1x2+x1+1
phi :: PolF2  -> PolF2
phi = linear (\m -> product [ var x | (x,i) <- mindices m])
```

Para poder comprobar la propiedad clave que justifica la redefinición de phi, es necesaria la función `(ideal p)` que devuelve el ideal (con menos generadores) respecto al cual se calcula el grupo cociente para buscar el representante.

```
-- | Por ejemplo,
--
-- >>> [x1,x2] = [var "x1", var "x2"] :: [PolF2]
-- >>> ideal (1+x1+x1^2*x2)
-- [x1^2+x1,x2^2+x2]
ideal :: PolF2 -> [PolF2]
ideal p = [v+v^2| v<-vars p]
```

La propiedad implementada queda:

```
-- |
-- >>> quickCheck prop_phi
-- +++ OK, passed 100 tests.
prop_phi :: PolF2 -> Bool
prop_phi p = phi p == p %% (ideal p)
```

Tal y como se ha descrito anteriormente, Φ selecciona un representante de la clase de equivalencia de $\mathbb{F}_2[\mathbf{x}]/_{\mathbb{I}_2}$. Dicho representante resulta ser también un polinomio, por lo que cuando se quiere asociar un polinomio a una fórmula proposicional basta aplicar la composición $\pi := \Phi \circ P$, que se llamará *proyección polinomial*.

Esta proyección es muy útil para manejar los polinomios ya que los simplifica en gran medida. Por ejemplo, sea $F = p_1 \to p_1 \wedge p_2$, entonces $P(F) = 1 + x_1 + x_1^2 x_2$ mientras que $\pi(F) = 1 + x_1 + x_1 x_2$.

La función `proyeccion p` es la implementación de la función $\pi(p)$:

```
-- | Por ejemplo,
-- >>> [p1,p2] = [Atom "p1",Atom "p2"]
-- >>> proyeccion p1
-- x1
```

```
-- >>> tr (p1 → p1 ∧ p2)
-- x1^2x2+x1+1
-- >>> proyeccion (p1 → p1 ∧ p2)
-- x1x2+x1+1
proyeccion :: FProp -> PolF2
proyeccion = (phi . tr)
```

Conveniene comprobar si se verifica que cualquier fórmula f es equivalente a $\theta(\pi(f))$:

```
-- |
-- >>> quickCheckWith (stdArgs {maxSize = 50}) prop_theta_proyeccion
-- +++ OK, passed 100 tests.
prop_theta_proyeccion :: FProp -> Bool
prop_theta_proyeccion f = equivalentes (theta (proyeccion f)) f
```

Además, como se ha solucionado el problema de los exponentes se puede comprobar la propiedad recíproca:

```
-- |
-- >>> quickCheck prop_proyeccion_theta
-- +++ OK, passed 100 tests.
prop_proyeccion_theta :: PolF2 -> Bool
prop_proyeccion_theta p = phi p == (proyeccion . theta) p
```

2.3.3. Bases de conocimiento e ideales

En esta subsección se recordará la correspondencia entre conjuntos algebraicos e ideales polinomiales (en el cuerpo de coeficientes $\mathbb{F}_2$) y la lógica proposicional.

Definición 2.3.3. Dado un subconjunto $X \subseteq (\mathbb{F}_2)^n$, se denota por $I(X)$ al ideal de polinomios de $\mathbb{F}_2[\mathbf{x}]$ que se anulan en X:

$$I(X) = \{a(\mathbf{x}) \in \mathbb{F}_2[\mathbf{x}] \quad : \quad a(u) = 0 \ \text{ para cualquier } \ u \in X\}$$

Simétricamente, a partir de un subconjunto $J \subseteq \mathbb{F}_2[\mathbf{x}]$ es posible definir el conjunto algebraico $\mathcal{V}(J)$ comentado anteriormente:

$$\mathcal{V}(J) = \{u \in (\mathbb{F}_2)^n \quad : \quad a(u) = 0 \ \text{ para cualquier } \ a(\mathbf{x}) \in J\}$$

Antes de enunciar y demostrar el teorema de Nullstellensatz para cuerpos finitos [1] (concretamente el cuerpo $\mathbb{F}_2$) es necesario un lema:

Lema 2.3.4. Sea un polinomio $p \in \mathbb{F}_2[\mathbf{x}]$, entonces $p \in \mathbb{I}_2^n \ \Leftrightarrow \ p(\mathbf{z}) = 0 \ \forall \mathbf{z} \in (\mathbb{F}_2)^n$

Prueba: La implicación hacia la derecha es trivial ya que:
Si $p \in \mathbb{I}_2^n$ entonces $p \in \langle x_1^2 + x_1, \ldots, x_n^2 + x_n \rangle$, es decir:

$$p = \sum_{i=0}^{n} q_i (x_i^2 + x_i)$$

donde $q_i \in \mathbb{F}_2[\mathbf{x}]$ con $i = 1, \ldots, n$.
Como todos los $(x_i^2 + x_i)$ se anulan en todo punto de $(\mathbb{F}_2)^n$ entonces p también.

La implicación hacia la izquierda se probará por inducción en el número de variables.
En el caso de una única variable ($n = 1$), la división euclídea de p por $\mathbb{I}_2^1 = x_1^2 + x_1$ queda:

$$p = a * (x_1^2 + x_1) + b \text{ con } a \in \mathbb{F}_2[\mathbf{x}] \text{ y } b = b_0 + b_1 x_1$$

De la hipótesis tenemos que $b(0) = b(1) = 0$; luego $b = 0$, y por tanto $p \in \mathbb{I}_2^1$.
Sea $n \geq 1$. Se usará también la división respecto $\mathbb{I}_1$, por lo que

$$p = a * \mathbb{I}_2^1 + b$$

donde $b = b_0 + b_1 x$; $b_0, b_1 \in \mathbb{F}_2[x_2, \ldots, x_n]$.
Fijando un punto cualquiera $z \in \mathbb{F}_2^{n-1}$ tal que $z = (z_2, \ldots, z_n)$, el polinomio b respecto de la variable x_1 queda:

$$b_0(z) + b_1(z) x_1 = 0$$

para $x_1 = 0$ y para $x_1 = 1$. Como, $b(z)(x_1)$ es de grado 1 y tiene 2 raíces, entonces

$$b_0(z) = b_1(z) = 0$$

Aplicando la hipótesis de inducción $b_0, b_1 \in \langle x_2^2 + x_2, \ldots, x_n^2 + x_n \rangle$ luego $p \in \mathbb{I}_2^n$. □

Teorema 2.3.5. Teorema de Nullstellensatz con coeficientes en el cuerpo $\mathbb{F}_2$

1. Si $A \subseteq (\mathbb{F}_2)^n$, entonces $\mathcal{V}(I(A)) = A$
2. Para todo $\mathfrak{J} \in \mathit{Ideales}(\mathbb{F}_2[\mathbf{x}])$, $I(\mathcal{V}(\mathfrak{J})) = \mathfrak{J} + \mathbb{I}_2$

Prueba:

1. Se prueba por doble contención:
   La primera ($A \subseteq \mathcal{V}(I(A))$) es trivial ya que si $a \in A$, por definición de I, se tiene que

   $$\forall p \in I(A),\ p(a) = 0$$

   Y por lo tanto, $a \in \mathcal{V}(I(A))$.
   La prueba de $\mathcal{V}(I(A)) \subseteq A$ se hará por *reductio ad absurdum*. Supongamos que

existe un punto $a \in \mathcal{V}(I(A))$ pero que $a \notin A$. Sea el polinomio

$$p_A(\mathbf{x}) = 1 + \sum_{\alpha \in A} \prod_{i=1}^{n} (x_i + \alpha_i + 1)$$

Es fácil ver que

$$p_A(u) = 0 \text{ si y sólo si } u \in A$$

De la hipótesis, se tiene que $p_A(a) \neq 0$, y de la definición de p_A que $p_A \in I(A)$. De esto se deduce que $a \notin \mathcal{V}(I(A))$, lo cual es una contradicción. □

2. Del teorema de las bases de Hilbert se deduce que

$$\exists j_1, \ldots, j_s \in \mathbb{F}_2[\mathbf{x}] \text{ tales que } \mathfrak{J} = \langle j_1, \ldots, j_s \rangle$$

Entonces, la prueba de $J + \mathbb{I}_2 \subseteq I(\mathcal{V}(\mathfrak{J}))$, es inmediata porque todos los polinomios j_k e $i_{k'}$ con $1 \leq k \leq s$, $1 \leq k' \leq n$ se anulan en $\mathcal{V}(\mathfrak{J})$.
Para probar $I(\mathcal{V}(\mathfrak{J})) \subseteq J + \mathbb{I}_2$, se fijan el polinomio $p \in I(\mathcal{V}(\mathfrak{J}))$ y el subconjunto $A \subseteq (\mathbb{F}_2)^n$ definido como

$$A := (\mathbb{F}_2)^n \setminus \mathcal{V}(\mathfrak{J})$$

Además se tiene que

$$\forall a = (a_1, \ldots, a_n) \in A \ , \ \exists i_a \in \{1, \ldots, s\} \text{ tal que } j_{i_a} \neq 0$$

Entonces, el polinomio

$$g = p \cdot \left(\prod_{a \in A} (j_{i_a} - j_{i_a}(a))\right)$$

se anula en todo $(\mathbb{F}_2)^n$ (porque p se anula en $\mathcal{V}(\mathfrak{J})$ y $\prod_{a \in A}(j_{i_a} - j_{i_a}(a))$ en A). Además, por el lema 2.3.4, $g \in \mathbb{I}_2^n$. Y desarrollando el producto en g se puede escribir el polinomio como

$$g = bp + h$$

, donde $h \in \mathfrak{J}$ y $b = \prod_{a \in A}(-j_{i_a}(a))$.
Por tanto, $bp = g - h \in \mathfrak{J} + \mathbb{I}_2^n$ y $b \neq 0$. De esto se sigue que $p \in \mathfrak{J} + \mathbb{I}_2^n$. □

Del teorema de Nullstellensatz se sigue que:

$$F \equiv F' \text{ si y sólo si } P(F) = P(F') \pmod{\mathbb{I}_2}$$

Por consiguiente, $F \equiv F'$ si y sólo si $\pi(F) = \pi(F')$. Para la prueba del Teorema 2.3.7 es necesario el siguiente lema:

Lema 2.3.6. Sean los polinomios $R, P_1, \ldots, P_m \in \mathbb{F}_2[\mathbf{x}]$, sea el ideal $\mathfrak{J} = \langle P_1, \ldots, P_m \rangle$ y sea $\mathcal{R} = \mathfrak{J} + \mathbb{I}_2^n$. Entonces,

$$R \in \mathcal{R} \iff R(a) = 0,\ \forall a \in A = \{z \in (\mathbb{F}_2)^n : P_1(z) = \cdots = P_m(z) = 0\}$$

Prueba: La implicación hacia la derecha ($\Rightarrow$) es trivial ya que $P_i(a) = 0$ y $a_j^2 + a_j = 0$ para todo $i = 1, \ldots, m$ y $j = 1, \ldots, n$.
Para la otra implicación ($\Leftarrow$), se define el conjunto $B = (\mathbb{F}_2)^n \setminus A$ de forma que

$$\forall z \in B\ ,\ \exists i_z \in \{1, \ldots, m\} \text{ tal que } P_{i_z}(z) \neq 0$$

A continuación, se define el polinomio

$$S = R \cdot \prod_{z \in B} (P_{i_z} - P_{i_z}(z))$$

Notar que este polinomio se anula en todo $(\mathbb{F}_2)^n$, ya que es producto de R, que se anula en A; y de $\prod_{z \in B}(P_{i_z} - P_{i_z}(z))$, que se anula en B. El lema 2.3.4 implica que $S \in \mathbb{I}_2^n$. Por lo que si se reescribe S desarrollando el producto:

$$S = b \cdot R + P', \text{ con } P' \in \mathfrak{J} \text{ y } b = \prod_{z \in B} (-P_{i_z}(z)) \in \mathbb{F}_2$$

se deduce que

$$b \cdot R = S - P' \in \mathfrak{J} + \mathbb{I}_2^n = \mathcal{R}$$

Finalmente, como $b \neq 0$, se tiene que $R \in \mathcal{R}$. □

El siguiente teorema resume la relación entre la lógica proposicional y $\mathbb{F}_2[\mathbf{x}]$:

Teorema 2.3.7. Sea $K = \{F_1, \ldots F_m\}$ un conjunto de fórmulas proposicionales y G una fórmula proposicional. Las siguientes sentencias son equivalentes:

1. $\{F_1, \ldots F_m\} \vDash G$
2. $1 + P(G) \in \langle 1 + P(F_1), \ldots, 1 + P(F_m) \rangle + \mathbb{I}_2$
3. $\mathcal{V}(1 + P(F_1), \ldots, 1 + P(F_m)) \subseteq \mathcal{V}(1 + P(G))$

Prueba: La estructura que se seguirá es probar $(2 \Leftrightarrow 1)$ y $(2 \Leftrightarrow)$. Por el lema 2.3.6:

$$1 + P(G) \in \langle 1 + P(F_1), \ldots, 1 + P(F_m) \rangle + \mathbb{I}_2 \iff$$
$$\iff 1 + P(G)(a) = 0\ \ \forall a \in A = \{z \in (\mathbb{F}_2)^n : 1 + P(F_1)(z) = \cdots = P(F_m)(z) = 0\}$$

En otras palabras, si se anulan todos los $(1 + P(F_i))$, $i = 1, \ldots, m$, entonces se anula $(1 + P(G))$. Esto pasa si y sólo si:

$$\mathcal{V}(1 + P(F_1), \ldots, 1 + P(F_m)) \subseteq \mathcal{V}(1 + P(G))$$

quedando así probado (2 $\Leftrightarrow$ 3).
Además, es fácil ver que $A = \{o_v : v \in Mod(\{F_1, \ldots F_m\})\}$, y por tanto,

$$1 + P(G)(a) = 0 \ \ \forall a \in A \Longleftrightarrow P(G)(a) = 1 \ \ \forall a \in A \Longleftrightarrow A \subseteq \{o'_v : v \in Mod(G)\} \Longleftrightarrow$$

$$\Longleftrightarrow Mod(\{F_1, \ldots, F_m\}) \subseteq Mod(G) \Longleftrightarrow \{F_1, \ldots F_m\} \vDash G$$

quedando así probado (2 $\Leftrightarrow$ 1). □

Se sabe que todo conjunto $X \subseteq (\mathbb{F}_2)^n$ es un conjunto algebraico; de hecho, existen $a_X \in \mathbb{F}_2(\mathbf{x})$ tal que $\mathcal{V}(a_X) = X$. Por lo que, aplicando el teorema de Nullstellensatz se tiene que $I(\mathcal{V}(a_X)) = (a_X) + \mathbb{I}_2$, de lo que se sigue que el anillo de coordenadas de X como variedad algebraica es:

$$\mathbb{F}_2[\mathbf{x}]/_{I(X)} \cong (\mathbb{F}_2[\mathbf{x}]/_{(a_X)})/_{\mathbb{I}_2}$$

Cualquier ideal J_X tal que $\mathcal{V}(J_X) = X$ se puede usar para describir el anillo de coordenadas. Por consiguiente y con el objetivo de simplificar la notación, se asumirá que $\mathbb{I}_2 \subseteq J_X$ si es necesario. De manera similar, dada una base de conocimiento K, se define el ideal:

$$J_K = (\{1 + P(F) : F \in K\})$$

y entonces

$$v \vDash K \iff o_v \in \mathcal{V}(J_K)$$

Definición 2.3.8. El *anillo de coordenadas* de K se define como el correspondiente a la variedad algebraica $V(J_K)$, que, por el teorema de Nullstellensatz, es $\mathbb{F}_2[\mathbf{x}]/_{(J_K)})/_{\mathbb{I}_2}$.

Capítulo 3

Regla de independencia y prueba no clausal de teoremas

En el presente capítulo se expondrá el diseño de un método de prueba de teoremas basado en la consistencia o inconsistencia del conjunto de fórmulas de partida, así como de la negación de la fórmula que se quiere deducir. En definitiva se basa en el hecho de que, si K es una base de conocimiento y F una fórmula proposicional:

$$K \models F \text{ si y sólo si } K \cup \{\neg F\} \text{ es inconsistente}$$

La idea principal será determinar la inconsistencia del conjunto de fórmulas mediante la saturación de dicho conjunto. Para saturar, se usarán las ya mencionadas retracciones conservativas, que se calcularán mediante lo que se denominará un *operador de omisión de variables*.

Este operador eliminará una de las variables cada vez que se aplique, obteniendo a cada paso un conjunto equivalente de fórmulas en los que se usa una variable menos. Finalmente, como hay un número finito de variables en $K \cup \{\neg F\}$, llegará un momento en el que no queden variables y que sólo queden las constantes $\top$ o $\bot$ (sólo una de ellas), habiendo así refutado o probado, respectivamente, el teorema.

Al igual que en el anterior capítulo, la metodología escogida para describir las bases teóricas que fundamentan el modelo lógico-algebraico de razonamiento así como su posterior implementación en el lenguaje Haskell, será la de paralelizar ambas exposiciones en la medida de lo posible, con la idea de facilitar la compresión al lector.

3.1. Retracción conservativa mediante omisión de variables

En esta sección se presenta cómo calcular retracciones conservativas usando los ya mencionados operadores *de omisión (o de olvido)*. Dichos operadores son funciones del tipo:

$$\delta : Form(\mathcal{L}) \times Form(\mathcal{L}) \longrightarrow Form(\mathcal{L})$$

Definición 3.1.1. Sea δ un operador: $\delta : Form(\mathcal{L}) \times Form(\mathcal{L}) \longrightarrow Form(\mathcal{L} \setminus \{p\})$ se dice que es:

1. *robusto* si $\{F, G\} \vDash \delta(F, G)$.
2. un *operador de omisión* para la variable $p \in \mathcal{L}$ si:

$$\delta(F, G) \equiv [\{F, G\}, \mathcal{L} \setminus \{p\}]$$

Una caracterización muy útil de los operadores se puede deducir de la siguiente propiedad semántica: Si δ es un operador de omisión, los modelos de $\delta(F, G)$ son precisamente las *proyecciones* de los modelos de $\{F, G\}$ (ver figura 3.1).

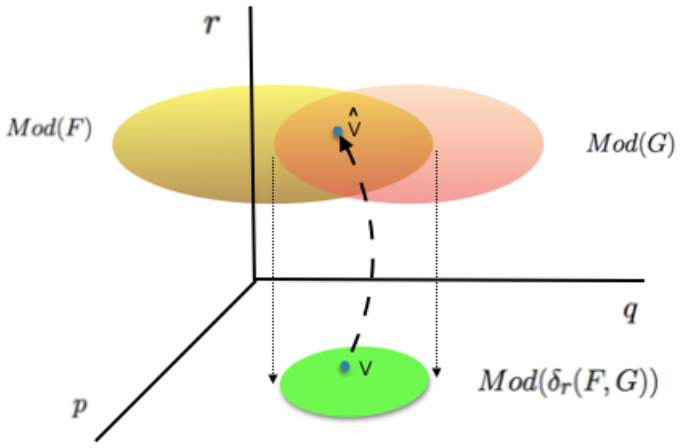

Figura 3.1: Interpretación semántica del operador de omisión (Lema de elevación)

Lema 3.1.2. (**Lema de elevación**) Sean $v : \mathcal{L} \setminus \{p\} \to \{0, 1\}$ una valoración o interpretación, $F, G \in Form(\mathcal{L})$ fórmulas y δ un operador de omisión de la variable p. Las siguientes condiciones son equivalentes:

1. $v \vDash \delta(F, G)$
2. Existe una valoración $\hat{v} : \mathcal{L} \to \{0, 1\}$ tal que $\hat{v} \vDash F \wedge G$ y $\hat{v} \restriction_{\mathcal{L}\setminus\{p\}} = v$

Prueba: (1 $\Rightarrow$ 2): Dada una valoración v, se considera la fórmula

$$H_v = \bigwedge_{q \in \mathcal{L} \setminus \{p\}} q^v$$

donde q^v es q si $v(q) = 1$ y $\neg q$ en otro caso. Es claro que v es la única valoración de $\mathcal{L} \setminus \{p\}$ que es modelo de H_v.
Supongamos que existe $v : \mathcal{L} \setminus \{p\} \to \{0,1\}$ modelo de $\delta(F,G)$, pero que no se puede extender a un modelo de $F \wedge G$. Entonces la fórmula

$$H_v \to \neg(F \wedge G)$$

es una tautología. En particular:

$$\{F,G\} \models H_v \to \neg(F \wedge G)$$

Como $\{F,G\} \models F \wedge G$, usando *modus tollens* se tiene $\{F,G\} \models \neg H_v$. Así que $\delta(F,G) \models \neg H_v$, por ser δ un retracción conservativa. Este hecho es una contradicción porque $v \models \delta(F,G) \wedge H_v$.
(2 $\Rightarrow$ 1): La extensión $\hat{v}$ verifica que:

$$\hat{v} \models F \wedge G \models [\{F,G\}, \mathcal{L} \setminus \{p\}] \models \delta(F,G)$$

Como $\delta(F,G) \in Form(\mathcal{L} \setminus \{p\})$, la valoración $v = \hat{v} \restriction_{\mathcal{L} \setminus \{p\}}$ también es modelo de $\delta(F,G)$. □

En particular, el resultado es cierto para la propia retracción conservativa canónica $[K, \mathcal{L} \setminus \{p\}]$, porque si consideramos la fórmula $\bigwedge K := \bigwedge_{F \in K} F$,

$$[K, \mathcal{L} \setminus \{p\}] \equiv \delta_p(\bigwedge K, \bigwedge K)$$

Un caso interesante aparece cuando $\delta_p(F_1, F_2) \equiv \top$. En este caso, toda valoración parcial en $\mathcal{L} \setminus \{p\}$ se puede extender a un modelo de $\{F_1, F_2\}$.

La siguiente caracterización será útil más adelante:

Corolario 3.1.3. Sea $\delta : Form(\mathcal{L}) \times Form(\mathcal{L}) \longrightarrow Form(\mathcal{L} \setminus \{p\})$ un operador robusto. Las siguientes condiciones son equivalentes:

1. δ es un operador de omisión de la variable p.
2. Para cualesquiera $F, G \in Form(\mathcal{L})$ y $v \models \delta(F,G)$ valoración sobre $\mathcal{L} \setminus \{p\}$, existe una extensión de v modelo de $\{F,G\}$.

Prueba: (1 $\Rightarrow$ 2): Cierta por el Lema de Elevación (Lema 3.1.2).
(2 $\Rightarrow$ 1): Sean F y G dos fórmulas. Como δ es robusto, basta probar que:

$$\delta(F,G) \models [\{F,G\}, \mathcal{L} \setminus \{p\}]$$

Supongamos que no es cierto. En ese caso, existe una fórmula $H \in Form(\mathcal{L} \setminus \{p\})$ tal que:

$$[\{F,G\}, \mathcal{L} \setminus \{p\}] \models H$$

Luego H también es consecuencia lógica de $\{F,G\}$. Sin embargo, existe una valoración v que satisface:

$$v \models \delta(F,G) \wedge \neg H$$

Por (2), existe $\hat{v}$ extensión de v que es modelo de $\{F,G\}$. Por tanto, $\{F,G\} \models \neg H$, lo que es una contradicción. □

Corolario 3.1.4. Si $p \notin var(F)$, y δ_p es un operador de omisión de p, entonces

$$\delta_p(F,F) \equiv F \qquad \text{y} \qquad \delta_p(F,G) \equiv \{F, \delta_p(G,G)\}$$

Prueba: Si $p \notin var(F)$, entonces

$$\{F\} \equiv [\{F\}, \mathcal{L} \setminus \{p\}] \equiv \delta_p(F,F)$$

Por otro lado,

$$\delta_p(F,G) \equiv [\{F,G\}, \mathcal{L} \setminus \{p\}] \models \{F, \delta_p(G,G)\}$$

Para probar que en realidad la última consecuencia lógica es una equivalencia se mostrará que tienen los mismos modelos.
Sea v una valoración sobre $\mathcal{L} \setminus \{p\}$ tal que

$$v \models \{F, \delta_p(G,G)\}$$

Entonces existe $\hat{v}$ (una extensión de v) tal que $\hat{v} \models G$. Como $\hat{v} \models F$, se tiene por el Lema de Elevación que $v \models \delta(F,G)$. □

Retracciones conservativas inducidas por un operador de omisión

En el artículo [5] J. Lang et al. presentan un método de omisión de X (un conjunto de variables de la fórmula F), denotado por $\texttt{forget}(F, X)$ y basado en la construcción de disyunciones de la siguiente forma:

$$\begin{array}{lcl} \texttt{forget}(F, \emptyset) & = & F \\ \texttt{forget}(F, \{x\}) & = & F\{x/\top\} \cup F\{x/\bot\} \\ \texttt{forget}(F, \{x\} \cup Y) & = & \texttt{forget}(\texttt{forget}(F, Y), \{x\}) \end{array}$$

Notar que con este método el tamaño de $\texttt{forget}(F, Y)$ puede ser realmente grande. En el método que se expone en el trabajo se pretende simplificar la representación mediante el uso de operaciones algebraicas sobre proyecciones polinomiales.

Tal y como se ha descrito anteriormente, el operador de omisión de la variable p actúa entre pares de fórmulas. A continuación, se extenderá la definición del operador de forma que se pueda aplicar a conjuntos de fórmulas o bases de conocimiento.

Definición 3.1.5. Sea δ_p un operador de omisión de la variable p y K una base de conocimiento. Se define $\delta_p[\cdot]$ como:

$$\delta_p[\cdot] : 2^{Form(\mathcal{L})} \rightarrow 2^{Form(\mathcal{L})}$$
$$\delta_p[K] := \{\delta_p(F, G) : F, G \in K\}$$

Definición 3.1.6. Se supone que se tiene un operador de omisión δ_p para cada $p \in \mathcal{L}$. Se llamará *saturación* de la base de conocimiento K al proceso de aplicar los operadores $\delta_p[\cdot]$ (en algún orden) respecto a todas las variables proposicionales de $\mathcal{L}(K)$, denotando al resultado como $sat_\delta(K)$ (el cual será un subconjunto de $\{\top, \bot\}$).

Posteriormente se verá que $sat_\delta(K)$ no depende del orden de aplicación de los operadores. Además, se probará que debido a que los operadores de omisión son robustos, si K es consistente entonces necesariamente $sat_\delta(K) = \{\top\}$.

A partir de los operadores de omisión resulta natural definir el siguiente cálculo lógico:

Definición 3.1.7. Sea K una base de conocimiento, $F \in Form(\mathcal{L})$ y $\{\delta_p : p \in \mathcal{L}(K)\}$ una familia de operadores de omisión.

- Una $\vdash_\delta$-prueba en K es una secuencia de fórmula $F_1, \ldots, F_n$ tal que para todo $i \leq n$, $F_i \in K$ ó existen $F_j, F_k (j, k < i)$ tal que $F_i = \delta_p(F_j, F_k)$ para algún $p \in \mathcal{L}$.

- $K \vdash_\delta F$ si existe una $\vdash_\delta$-prueba en $K, F_1, \ldots, F_n$, con $F_n = F$.
- Una $\vdash_\delta$-refutación es una $\vdash_\delta$-prueba de $\bot$.

La completitud (refutacional) del cálculo asociado a los operadores de omisión se enuncia como sigue:

Teorema 3.1.8. Sea $\{\delta_p : p \in \mathcal{L}\}$ una familia de operadores de omisión. Entonces $\vdash_\delta$ es refutacionalmente completo, es decir, K es inconsistente si y sólo si $K \vdash_\delta \bot$.

Prueba: La idea es saturar la base de conocimiento como en la Figura (3.2).
Si $sat_\delta(K) = \{\top\}$, entonces, aplicando repetidas veces el lema de elevación, se puede extender la valoración vacía (la cual es modelo de $\{\top\}$) a un modelo de K.
Si $\bot \in sat_\delta(K)$ entonces K es inconsistente, porque $K \vDash sat_\delta(K)$ por robustez de los operadores de omisión. □

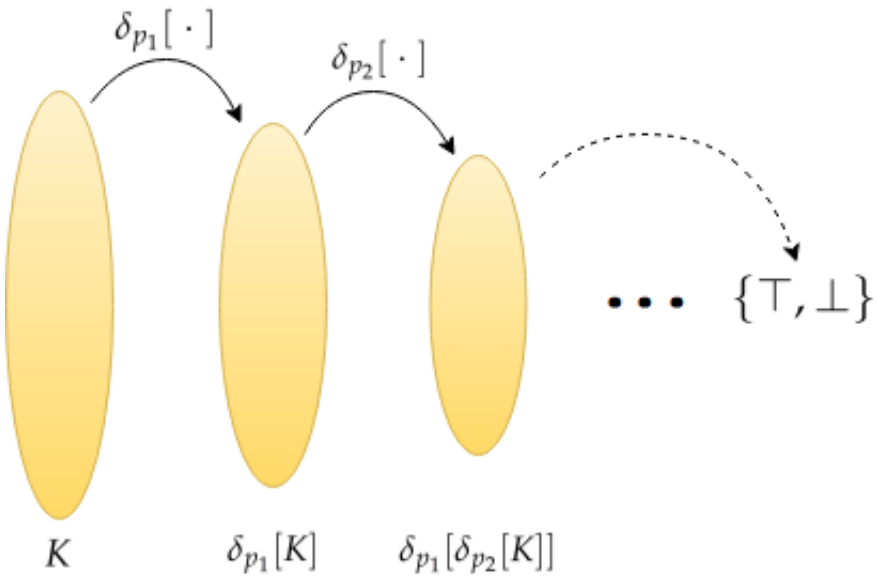

Figura 3.2: Decidir la consistencia usando operadores de omisión (δ)

Corolario 3.1.9. $\delta_p[K] \equiv [K, \mathcal{L} \setminus \{p\}]$
Prueba: Por robustez del operador de omisión δ_p se tiene:

$$[K, \mathcal{L} \setminus \{p\}] \vDash \delta_p[K]$$

Para probar la otra dirección, se considera $F \in [K, \mathcal{L} \setminus \{p\}]$ y se supone que $\delta_p[K] \nvDash F$. Entonces $\delta_p[K] + \{\neg F\}$ es consistente. En particular, si se satura se tiene: $sat_\delta(\delta_p[K] \cup \{\neg F\}) = \{\top\}$.

Como $p \notin var(\neg F)$ se puede usar el corolario 3.1.4, obteniendo que para todo $G \in K$:

$$\delta_g(\neg F, G) \equiv \{\neg F, \delta_g(G, G)\} \qquad \text{y} \qquad \delta_p(\neg F, \neg F) \equiv \neg F$$

Por consiguiente,

$$\delta_p[K \cup \{\neg F\}] \equiv \delta_p[K] \cup \{\neg F\}$$

así que, aplicando saturación empezando por p:

$$sat_\delta(K \cup \{\neg F\}) \equiv sat_\delta(\delta_p[K] \cup \{\neg F\})$$

Lo que indica que $K \cup \{\neg F\}$ es consistente, luego $K \nvDash F$, que es una contradicción. □

Dados $Q \subseteq \mathcal{L}$ y un orden lineal $q_1 < \cdots < q_k$ sobre Q, se define el operador:

$$\delta_{Q,<} := \delta_{q_1} \circ \cdots \circ \delta_{q_k}$$

Que no es más que la aplicación sucesiva del operador de omisión δ respecto de cada una de las variables de Q en un orden dado. De hecho, si prescindimos de hacer explícito el orden, el operador está bien definido módulo equivalencia lógica. En otras palabras, dados dos órdenes sobre Q cualesquiera, producen bases de conocimiento equivalentes. Esto es cierto puesto que para cada $p, q \in \mathcal{L}$, usando el corolario 3.1.9 se sigue que:

$$\delta_p \circ \delta_q[K] \equiv \delta_q \circ \delta_p[K]$$

ya que ambas son equivalentes a $[K, \mathcal{L} \setminus \{p, q\}]$. Por consiguiente y con la idea de simplificar la notación, se escribirá δ_Q cuando no importe la presentación sintáctica de dicha base de conocimiento.

Como consecuencia del teorema 3.1.8 y el corolario 3.1.9 tenemos que el hecho de saber si una fórmula es o no consecuencia lógica de otra (o de una base de conocimiento) se puede reducir a otro problema similar en el que sólo se trabaje con las variables de la fórmula objetivo. Basta con eliminar de forma sucesiva todas las variables de la base de conocimiento que no ocurran en la fórmula objetivo. A esta propiedad se le conoce como *Propiedad de localización*:

Corolario 3.1.10. (**Propiedad de Localización [3]**) Las siguientes condiciones son equivalentes:

1. $K \vDash F$
2. $\delta_{\mathcal{L} \setminus var(F)}[K] \vDash F$

Prueba: Trivial, porque $\delta_{\mathcal{L} \setminus var(F)}[K] \equiv [K, var(F)]$

3.2. Derivadas Booleanas

Con el objetivo de definir el operador de omisión que se usará en la herramienta, se hará uso de las derivadas polinomiales aunque no de forma directa. En la práctica, se traducirá la derivada tradicional en $\mathbb{F}_2[\mathbf{x}]$ a un operador sobre fórmulas proposicionales. Antes de continuar se expondrán diversas propiedades básicas. Recalcar que una derivación sobre un anillo R es una función $d : R \to R$ verificando que $\forall a, b \in R$:

- $d(a+b) = d(a) + d(b)$
- $d(a \cdot b) = d(a) \cdot b + a \cdot d(b)$

La traducción de las derivaciones al contexto de la lógica se construye como se describe a continuación:

Definición 3.2.1. [3] Una función $\partial : Form(\mathcal{L}) \to Form(\mathcal{L})$ es una *derivada Booleana* si existe una derivación d sobre $\mathbb{F}_2[\mathbf{x}]$ tal que el siguiente diagrama sea conmutativo:

$$\begin{array}{ccc} Form(\mathcal{L}) & \xrightarrow{\partial} & Form(\mathcal{L}) \\ \pi \downarrow & \# & \uparrow \Theta \\ \mathbb{F}_2[\mathbf{x}] & \xrightarrow{d} & \mathbb{F}_2[\mathbf{x}] \end{array}$$

O lo que es lo mismo:

$$\partial = \Theta \circ d \circ \pi$$

En este trabajo, la atención recae sobre una derivada booleana en particular, denotada por $\frac{\partial}{\partial p}$, e inducida por la derivada $d = \frac{\partial}{\partial x_p}$. A continuación se implementará dicha derivada. Para ello se crea el módulo `Derivada` y se importan los módulos necesarios.

```
module Derivada (derivaPol) where

import Logica
import Haskell4Maths ( lm
                     , var
                     , vars
                     , linear
                     , mindices
                     , Lex (..)
                     , MonImpl(..))
import F2 (PolF2)
import Transformaciones (proyeccion
                         , theta)

import Data.List (union)
import Test.QuickCheck (quickCheck)
```

Destacar que se restringirá la definición de la derivada $\frac{\partial}{\partial x_p}$ a polinomios en el espacio cociente. Esto implica que se supondrá que los polinomios son proyecciones de fórmulas y que por lo tanto, las variables que aparecen en dichos polinomios tienen a lo sumo exponente igual a 1. La función `(derivaMononomio m v)` es la derivada del monomio m respecto de la variable v, siempre que el exponente de todas sus variables sea menor o igual que 1.

```
-- | Por ejemplo,
--
-- >>> x1 = var "x1" :: PolF2
-- >>> exampleMonomial1 = (Lex (M 1 []))
-- >>> exampleMonomial1
-- 1
-- >>> derivaMononomio exampleMonomial1 x1
-- 0
-- >>> exampleMonomial2 = (Lex (M 1 [("x1",1)]))
-- >>> exampleMonomial2
-- x1
-- >>> derivaMononomio exampleMonomial2 x1
-- 1
-- >>> exampleMonomial3 = (Lex (M 1 [("x1",1),("x2",1)]))
-- >>> exampleMonomial3
-- x1x2
-- >>> derivaMononomio exampleMonomial3 x1
-- x2
exampleMonomial2 = (Lex (M 1 [("x1",1)]))
derivaMononomio :: (Lex String) -> PolF2 -> PolF2
derivaMononomio m v
  | varDif `elem` mIndices =
      product [var x ^ i | (x,i) <- mIndices, x /= fst varDif]
  | otherwise =  0
  where mIndices = mindices m
        varDif   = head (mindices (lm v))
```

La función `(derivaPol p v)` es la derivada del polinomio p respecto de la variable v, siempre que el exponente de todas sus variables sea menor o igual que 1.

```
-- | Por ejemplo,
-- >>> [x1,x2,x3,x4] = (map var ["x1","x2","x3","x4"]) :: [PolF2]
-- >>> derivaPol x1 x1
-- 1
-- >>> derivaPol (1+x1+x2+x1*x2) x1
-- x2+1
-- >>> derivaPol (x1*x2+x1+x3*x4+1) x1
-- x2+1

derivaPol :: PolF2 -> PolF2 -> PolF2
derivaPol p v = linear (`derivaMononomio` v) p
```

El siguiente resultado muestra una expresión semánticamente equivalente de esta derivada:

Proposición 3.2.2. $\frac{\partial}{\partial x_p}F \equiv \neg(F\{p/\neg p\} \leftrightarrow F)$
Prueba: Es fácil ver que

$$\pi(F\{p/\neg p\})(x) = \pi(F)(x_1, \ldots, x_p + 1, \ldots, x_n)$$

Por otro lado, se cumple que para fórmulas polinomiales:

$$\frac{\partial}{\partial x}a(x) = a(x+1) + a(x)$$

Así que,

$$\frac{\partial}{\partial x_p}\pi(F) = \pi(F)(x_1, \ldots, x_p + 1, \ldots, x_n) + \pi(F)(x_1, \ldots, x_n)$$

Y por tanto, aplicando Θ se tiene

$$\frac{\partial}{\partial p}F = \Theta(\frac{\partial}{\partial x_p}\pi(F)) \equiv \neg(F\{p/\neg p\} \leftrightarrow F)$$

□

Esta propiedad se implementa en Haskell de la siguiente manera:

```
-- |
-- >>> quickCheck prop_derivada
-- +++ OK, passed 100 tests.

prop_derivada :: FProp -> Int -> Bool
prop_derivada f n = equivalentes (theta (derivaPol pol v))
                                 (no (Equi (sustituye f varP (no p)) f))
  where pol           = proyeccion f
        vs            = union (vars pol) [((var "x") :: PolF2)]
        v             = vs !! (n `mod` (length vs))
        p             = theta v
        aux (Atom xs) = xs
        varP          = aux p
```

Notar que el valor de verdad de $\frac{\partial}{\partial p}F$ respecto de una valoración no depende del valor de verdad de la propia p; luego se pueden aplicar valoraciones sobre $\mathcal{L} \setminus \{p\}$ a dicha fórmula. De hecho, se puede describir la estructura de F aislando el rol de la variable p de la siguiente manera:

Lema 3.2.3. [3] (Forma p-normal) Sea $F \in Form(\mathcal{L})$ y sea p una variable proposicional. Existe $F_0 \in Form(\mathcal{L} \setminus \{p\})$ tal que

$$F \equiv \neg(F_0 \leftrightarrow p \wedge \frac{\partial}{\partial p} F)$$

Prueba: Como $\pi(F)$ es una fórmula polinomial, se puede suponer que

$$\pi(F) = a + x_p b \quad \text{con} \quad deg_{x_p}(a) = deg_{x_p}(b) = 0$$

Por consiguiente,

$$F \equiv \Theta(\pi(F)) \equiv \neg(\theta(a) \leftrightarrow p \wedge \Theta(b))$$

Entonces, $F_0 = \Theta(a)$. Además, como $b = \frac{\partial}{\partial x_p}\pi(F)$, se tiene que $\Theta(b) = \frac{\partial}{\partial p} F$. □

Por ejemplo, sea $F = p \wedge q \rightarrow r$. Entonces,

$$\pi(F) = 1 + x_p x_q + x_p x_q x_r = 1 + x_p(x_q + x_q x_r)$$

Siguiendo la prueba anterior, $a = 1$ y $b = x_q + x_q x_r$ (notar que $\frac{\partial}{\partial x_p}(\pi(F)) = x_q + x_q + x_r$). Por tanto, $\Theta(a) = \top$ y $\Theta(b) = q \wedge \neg r$, así que:

$$F \equiv \neg(\top \leftrightarrow p \wedge (q \wedge \neg r))$$

3.3. Regla de independencia

El operador de omisión que se va a definir en esta sección se llama regla de independencia y trata de representar los modelos de la retracción conservativa como aquellos que pueden ser extendidos a modelos de $F \wedge G$. Destacar que ésta es la idea tras el Lema de elevación (3.1.2).

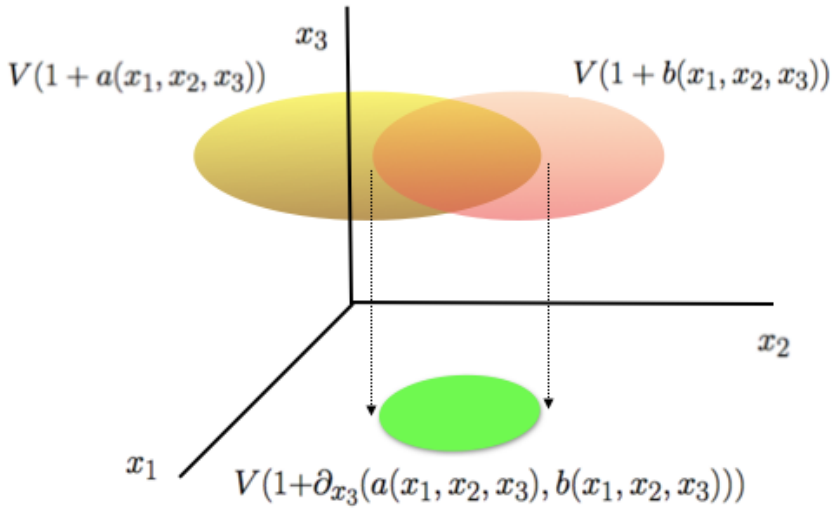

Figura 3.3: Interpretación geométrica de la regla de independencia

Geométricamente, si a y b son los polinomios $\pi(F)$ y $\pi(G)$, respectivamente; entonces el conjunto $\mathcal{V}(1+a, 1+b)$ (que correspondería al conjunto de modelos tanto de F como de G) se proyecta mediante ∂_p (ver Figura 3.3). La expresión algebraica de dicha proyección se describe a continuación en forma de regla:

Definición 3.3.1. La *regla de independencia* (o regla ∂) sobre fórmulas polinomiales se define como sigue: dados $a_1, a_2 \in \mathbb{F}_2[\mathbf{x}]$ y x una variable indeterminada

$$\frac{a_1, a_2}{\partial(a_1, a_2)}$$

donde:

$$\partial_x(a_1, a_2) = 1 + \Phi[(1 + a_1 \cdot a_2) \cdot (1 + a_1 \cdot \frac{\partial}{\partial x} a_2 + a_2 \cdot \frac{\partial}{\partial x} a_1 + \frac{\partial}{\partial x} a_1 \cdot \frac{\partial}{\partial x} a_2)]$$

La implementación en Haskell de dicha regla se hará en el módulo `Regla`

```
module Regla where

import Logica
import Haskell4Maths ( var
                     , vars)
import F2 (PolF2)
import Transformaciones ( phi
                        , theta
                        , proyeccion)
import Derivada (derivaPol)

import Data.List (union)
import Test.QuickCheck (quickCheck)
import qualified Data.Set as S
```

La función `(reglaIndependencia x a1 a2)` es el polinomio obtenido de aplicar la regla de Independencia a los polinomios $a1$ y $a2$, respecto de la variable x.

```
-- | Por ejemplo,
-- >>> [x1,x2,x3,x4] = (map var ["x1","x2","x3","x4"]) :: [PolF2]
-- >>> reglaIndependencia x1 1 1
-- 1
-- >>> reglaIndependencia x1 1 0
-- 0
-- >>> reglaIndependencia x1 x1 x1
-- 1
-- >>> reglaIndependencia x1 x1 x1*x2
-- x2
-- >>> reglaIndependencia x1 (x1*x3) (x1*x2)
-- x2x3
-- >>> reglaIndependencia x1 (1+x1*x3) (x1*x2)
-- x2x3+x2

reglaIndependencia :: PolF2 -> PolF2 -> PolF2 -> PolF2
reglaIndependencia x a1 a2 = aux + a1a2 + aux2
  where da1       = derivaPol a1 x
        da2       = derivaPol a2 x
        a1a2      = phi $ a1*a2
        a1da2     = phi $ a1*da2
        a2da1     = phi $ a2*da1
        da1da2    = phi $ da1*da2
        aux       = phi $ a1da2 + a2da1 + da1da2
        aux2      = phi $ a1a2*aux
```

Recordar que la función `(phi p)` ó $\Phi(p)$ escogía el representante de menor grado del polinomio $p + \mathbb{I}_2$.

Además, destacar que se aplica la función `phi` a cada multiplicación de forma aislada ya que es más eficiente que realizar las operaciones necesarias y finalmente aplicarla.

Como caso particular, dados los polinomios $a_i = b_i + x_p \cdot c_i$, con $deg_{x_p}(b_i) = deg_{x_p}(c_i) = 0$ para $i = 1, 2$, la regla se puede reescribir de la siguiente forma:

$$\partial_{x_p}(a_1, a_2) = \Phi[1 + (1 + b_1 \cdot b_2) \cdot [1 + (b_1 + c_1)(b_2 + c_2)]]$$

Por ejemplo, para hallar a tal que:

$$a = \partial_{x_2}(1 + x_2x_3x_5 + x_3x_5, 1 + x_1x_2x_3x_4x_5 + x_1x_2x_3x_5)$$

basta con saber que

- $b_1 = 1 + x_3x_5$
- $c_1 = x_3x_5$
- $b_2 = 1$
- $c_2 = (1 + x_4)x_1x_3x_5$

luego el resultado es $a = 1 + x_1x_3x_4x_5 + x_1x_3x_5$. Si se comprueba el resultado en Haskell se verá que coinciden:

```
-- |
-- >>> [x1,x2,x3,x4,x5] = (map var ["x1","x2","x3","x4","x5"])::[PolF2]
-- >>> reglaIndependencia x2 (1+x2*x3*x5+x3*x5)
-- x1x3x4x5+x1x3x5+1
```

Notar que la regla de independencia es simétrica. Se comprobará aplicando `quickCheck` a la siguiente propiedad:

```
-- |
-- >>> quickCheck prop_reglaIndep_simetrica
-- +++ OK, passed 100 tests.

prop_reglaIndep_simetrica :: PolF2 -> PolF2 -> Int -> Bool
prop_reglaIndep_simetrica a1 a2 n = reglaIndependencia x a1' a2' ==
                                    reglaIndependencia x a2' a1'
  where a1' = phi a1
        a2' = phi a2
        xs  = union (vars a1') (vars a2')
        xss = if (null xs) then [((var "x") :: PolF2)]
                           else xs
        x   = xss !! (n `mod` (length xss))
```

Para fórmulas, la regla de independencia se define como:

$$\partial_p(F_1, F_2) := \Theta(\partial_{x_p}(\pi(F_1), \pi(F_2)))$$

Mientras que su implementación es:

```
reglaIndForm :: VarProp -> FProp -> FProp -> FProp
reglaIndForm p f1 f2 = theta $ reglaIndependencia x p1 p2
  where x  = proyeccion (Atom p)
        p1 = proyeccion f1
        p2 = proyeccion f2
```

Siguiendo con el ejemplo anterior,

$$\begin{aligned}
& \partial_{p_2}(p_3 \wedge p_5 \to p_2\,,\; p_1 \wedge p_2 \wedge p_3 \wedge p_5 \to p_4) && = \\
= \;& \Theta(\partial_{x_2}(1 + x_2x_3x_5 + x_3x_5, 1 + x_1x_2x_3x_4x_5 + x_1x_2x_3x_5)) && = \\
= \;& \Theta(1 + x_1x_3x_4x_5 + x_1x_3x_5) = \neg(p_1 \wedge p_3 \wedge p_4 \wedge p_5 \leftrightarrow p_1 \wedge p_3 \wedge p_5) && = \\
= \;& p_1 \wedge p_3 \wedge p_5 \to p_4
\end{aligned}$$

Mientras que en Haskell:

```
-- |
-- >>> [p1,p2,p3,p4,p5] = map Atom ["p1","p2","p3","p4","p5"]
-- >>> f1 = p3 ∧ p5 → p2
-- >>> f2 = p1 ∧ p2 ∧ p3 ∧ p5 → p4
-- >>> g = p1 ∧ p3 ∧ p5 → p4
-- >>> reglaIndForm "p2" f1 f2
-- ¬((p1 ∧ (p3 ∧ (p4 ∧ p5))) ↔ ¬((p1 ∧ (p3 ∧ p5)) ↔ ⊤))
-- >>> equivalentes g (reglaIndForm "p2" f1 f2)
-- True
```

Es importante destacar las siguientes características sobre la regla ∂_p:

- Si $\partial_p(F, G)$ es una tautología, entonces $\partial_p(F, G) = \top$.
- Si $\partial_p(F, G)$ es inconsistente, entonces $\partial_p(F, G) = \bot$.

Ambas características son consecuencia de la transformación a polinomios, pues las fórmulas polinomiales que corresponden a tautologías o inconsistencias son algebraicamente simplificadas a 1 ó 0 en $\mathbb{F}_2[\mathbf{x}]/\mathbb{I}_2$, respectivamente.

De hecho, se trabaja con las proyecciones polinomiales para explotar la propiedad anterior. Por ejemplo,

```
-- |
-- >>> p = Atom "p"
-- >>> proyeccion $ p ∨ (no p)
-- 1
-- >>> proyeccion $ p ∧ (no p)
-- 0
```

A continuación, se expondrán diversos resultados sobre la regla de independencia, que justificarán el uso de la misma como herramienta para probar teoremas.

Proposición 3.3.2. Sea p una variable proposicional, entonces ∂_p es robusto.
Prueba: Hay que probar que $F_1 \wedge F_2 \vDash \partial_p(F_1, F_2)$. Para ello, se supone que:

$$\pi(F_1) = b_1 + x_p \cdot c_1 \ , \ \pi(F_2) = b_2 + x_p \cdot c_2$$

De acuerdo al teorema 2.3.7, basta probar que

$$\mathcal{V}(1 + \pi(F_1) \cdot \pi(F_2)) \subseteq \mathcal{V}(1 + \partial_{x_p}(\pi(F_1), \pi(F_2)))$$

Sea $\mathbf{u} \in \mathcal{V}(1 + \pi(F) \cdot \pi(G)) \subseteq \mathbb{F}_2^n$, es decir,

$$(b_1 + x_p c_1)(b_2 + x_p c_2)|_{x=u} = 1 \tag{3.1}$$

Ahora se pueden distinguir dos casos:

- La coordenada p-ésima de $\mathbf{u}$ es 0. Entonces por 3.1 se tiene que

  $$b_1|_{x=u} = b_2|_{x=u} = 1$$

  Y por lo tanto, $1 + b_1 b_2|_{x=u} = 0$

- La coordenada p-ésima de $\mathbf{u}$ es 1. En este caso, $(b_1 + c_1)(b_2 + c_2)|_{x=u} = 1$

Examinando la definición de ∂_p se concluye que en ambos casos

$$\partial_{x_p}(\pi(F_1), \pi(F_2))|_{x=u} = 1$$

así que $\mathbf{u} \in \mathcal{V}(1 + \partial_{x_p}(\pi(F_1), \pi(F_2)))$. □

El siguiente resultado es

Teorema 3.3.3. ∂_p es un operador de omisión.
Prueba: El objetivo es probar que

$$[\{F_1, F_2\}, \mathcal{L} \setminus \{p\}] \equiv \partial_p(F_1, F_2)$$

Se supone que $F_1, F_2 \in Form(\mathcal{L})$ y que $\pi(F_i) = b_i + x_p c_i$ con $i = 1, 2$, donde b_i y c_i son polinomios sin la variable x_p. Recordar que en este caso la expresión de la regla es:

$$\partial_{x_p}(\pi(F_1), \pi(F_2)) = \Phi[1 + (1 + b_1 \cdot b_2) \cdot [1 + (b_1 + c_1)(b_2 + c_2)]]$$

Como se ha probado la robustez del operador ∂_p en la proposición anterior, por el corolario 3.1.3 es suficiente probar que cualquier valoración v sobre $\mathcal{L} \setminus \{p\}$ que sea modelo de $\partial_p(F_1, F_2)$ se puede extender a $\hat{v} \vDash \{F_1, F_2\}$.

Sea $v \vDash \partial_p(F_1, F_2)$. Se considerará el punto de $\mathbb{F}_2^n$ asociado a v, o_v. Se sigue que,

$$\begin{aligned} o_v \in & \quad \mathcal{V}(1 + \pi(\partial_p(F_1, F_2))) & = \\ = & \quad \mathcal{V}(1 + (\partial_{x_p}(\pi(F_1), \pi(F_2)))) & = \\ = & \quad \mathcal{V}((1 + b_1 \cdot b_2)[1 + (b_1 + c_1)(b_2 + c_2)]) \end{aligned}$$

luego

$$((1 + b_1 \cdot b_2)[1 + (b_1 + c_1)(b_2 + c_2)])|_{x=o_v} = 0$$

Con el objetivo de construir la extensión requerida $\hat{v}$, se distinguen dos casos:

- Si $(1 + b_1 \cdot b_2)|_{x=o_v} = 0$, entonces

$$\hat{v} = v \cup \{(x_p, 0)\} \vDash F_1 \wedge F_2$$

- Si $[1 + (b_1 + c_1)(b_2 + c_2)]|_{x=o_v} = 0$, entonces

$$\hat{v} = v \cup \{(x_p, 1)\} \vDash F_1 \wedge F_2$$

□

Abusando de notación, se usará el mismo símbolo $\vdash_\partial$ tanto para fórmulas (3.1.7) como para polinomios. De esta forma, se pueden describir $\vdash_\partial$-pruebas sobre polinomios. Por ejemplo, una $\vdash_\partial$-refutación para el conjunto $\pi[\{p \to q, q \vee r \to s, \neg(p \to s)\}]$ es:

1.	$1 + x_1 + x_1x_2$	$[\![\pi(p \to q)]\!]$
2.	$1 + (x_2 + x_3 + x_2x_3)(1 + x_4)$	$[\![\pi(q \vee r \to s)]\!]$
3.	$x_1(1 + x_4)$	$[\![\pi(\neg(p \to s))]\!]$
4.	$1 + x_1 + x_3 + x_1x_4 + x_3x_4 + x_1x_3 + x_1x_3x_4$	$[\![\partial_{x_2}((1.)(2.))]\!]$
5.	0	$[\![\partial_{x_1}((3.)(4.))]\!]$

El mismo ejemplo en Haskell, salvando que se cambiarán las variables proposicionales p, q, r, s por p_1, p_2, p_3, p_4:

```
-- |
-- >>> [p1,p2,p3,p4] = map Atom ["p1","p2","p3","p4"]
-- >>> [f1,f2,f3] = [p1 → p2, p2 ∨ p3 → p4, no (p1 → p4)]
-- >>> proyeccion f1
-- x1x2+x1+1
-- >>> proyeccion f2
-- x2x3x4+x2x3+x2x4+x2+x3x4+x3+1
-- >>> proyeccion f3
-- x1x4+x1
-- >>> x1 = proyeccion p1
-- >>> x2 = proyeccion p2
-- >>> reglaIndependencia x2 (proyeccion f1) (proyeccion f2)
-- x1x3x4+x1x3+x1x4+x1+x3x4+x3+1
-- >>> reglaIndependencia x1 (proyeccion f3)
-- 0
```

Del teorema anterior se deduce que:

Corolario 3.3.4. [3] K es inconsistente si y sólo si $K \vdash_\partial \bot$.
Prueba: Es consecuencia directa de los teoremas 3.1.8 y 3.3.3. □

El resultado anterior en términos algebraicos queda:

Corolario 3.3.5. Sea $F \in Form(\mathcal{L})$ una base de conocimiento. Los siguientes enunciados son equivalentes:

1. $K \vDash F$
2. $J_K \vdash_\partial 0$, donde J_K es el ideal definido en la página 47.

Prueba: $(1) \Rightarrow (2)$: Supuesto $K \vDash F$, entonces $K \cup \{\neg F\}$ es inconsistente. Como ∂_p es refutacionalmente completo, $K \cup \{\neg F\} \vdash_\partial \bot$. Por tanto,

$$\{1 + \pi(G) : G \in K\} \cup \{\pi(F)\} \vdash_\partial 0$$

$(1) \Rightarrow (2)$: Si se encuentra una ∂-refutación con los polinomios, entonces, por el corolario anterior, $K \cup \{\neg F\}$ es inconsistente. □

Por ejemplo, si consideramos $G = p_4 \rightarrow p_3$ y sea K la base de conocimiento:

$$f(x) = \begin{cases} p_5 \wedge p_1 \leftrightarrow p_4 \\ p_5 \wedge p_3 \rightarrow p_4 \\ p_5 \wedge p_2 \rightarrow p_4 \\ p_1 \wedge p_2 \wedge p_4 \wedge p_5 \rightarrow p_3 \end{cases}$$

Para decidir si $K \vDash G$ se debe computar:

$$\partial_{\mathcal{L}\setminus\{p_3,p_4\}}[K] \equiv \partial_{p_1}[\partial_{p_2}[\partial_{p_5}]]$$

Previamente a los cálculos y debido a la manifiesta necesidad de extender la definición en Haskell de la regla de independencia, se procede a la implementación de la misma. Para lo cual, será necesaria la siguiente función auxiliar.

La función `(reglaIndependenciaAux v p ps)` aplica la regla de independecia respecto de la variable `v` a todos los pares de polinomios $(\mathtt{p}, p_i)$, con $p_i \in \mathtt{ps}$; y las incluye en el conjunto `acum`. Es decir, se fija en la primera coordenada al polinomio p y con la segunda coordenada se recorre el conjunto de polinomios ps.

```
reglaIndependenciaAux :: PolF2 -> PolF2 -> S.Set PolF2 ->
                         S.Set PolF2 -> S.Set PolF2
reglaIndependenciaAux v p ps acum
  | S.null ps = acum
  | otherwise = reglaIndependenciaAux v p ps' (S.insert dR acum)
                where (p',ps') = S.deleteFindMin ps
                      dR       = reglaIndependencia v p p'
```

Como la regla de independencia es simétrica, al aplicar una vez la función anterior no será necesario volver a aplicar la regla de independencia al polinomio distinguido p, y por consiguiente, se puede continuar aplicando la regla al resto de polinomios y decartar p.

De esta forma se define la función `(reglaIndependenciaKB v pps acum)` que aplica el operador de omisión ∂_v al conjunto `pps` y une todas las fórmulas obtenidas a las del conjunto `acum`.

```
reglaIndependenciaKB :: PolF2 -> S.Set PolF2 ->
                  S.Set PolF2 -> S.Set PolF2
reglaIndependenciaKB v pps acum
  | S.null pps   = acum
  | otherwise    = reglaIndependenciaKB v ps
                   (reglaIndependenciaAux v p pps acum)
      where (p,ps) = S.deleteFindMin pps
```

Volviendo al ejemplo anterior, ya se está en condiciones de realizar los cálculos con Haskell.

```
-- |
-- >>> [p1,p2,p3,p4,p5] = map Atom ["p1","p2","p3","p4","p5"]
-- >>> k = [p5 ∧ p1 ↔ p4,
-- >>> ps = S.fromList $ map proyeccion k
-- >>> ps
-- fromList [x1x2x3x4x5+x1x2x4x5+1,x1x5+x4+1,x2x4x5+x2x5+1,x3x4x5+x3x5+1]
-- >>> ps' = reglaIndependenciaKB (proyeccion p5) ps (S.fromList [])
-- >>> ps'
-- fromList [x1x2x3x4+x1x2x4+x1x4+x4+1,x1x4+x4+1,1]
-- >>> ps'' = reglaIndependenciaKB (proyeccion p2) ps' (S.fromList [])
-- >>> ps''
-- fromList [x1x4+x4+1,1]
-- >>> ps''' = reglaIndependenciaKB (proyeccion p1) ps'' (S.fromList [])
-- >>> ps'''
-- fromList [1]
```

Por tanto:

$$[K, \mathcal{L} \setminus \{p_3, p_4\}] \equiv \{\top\} \nvDash G$$

Se considerará ahora la fórmula $F = p_1 \wedge p_2 \wedge p_5 \rightarrow p_3$. Para saber si $K \vDash F$, hay que computar $[K, \mathcal{L}(F)] \equiv \partial_{p_3}[K]$ y luego probar que $\partial_{p_3}[K] \cup \{\neg F\}$ es inconsistente (saturando dicho conjunto):

```
-- |
-- >>> [p1,p2,p3,p4,p5] = map Atom ["p1","p2","p3","p4","p5"]
-- >>> k = [p5 ∧ p1 ↔ p4,
-- >>> ps = S.fromList $ map proyeccion k
-- >>> ps
-- fromList [x1x2x3x4x5+x1x2x4x5+1,x1x5+x4+1,x2x4x5+x2x5+1,x3x4x5+x3x5+1]
-- >>> ps' = reglaIndependenciaKB (proyeccion p3) ps (S.fromList [])
-- >>> ps'
-- fromList [x1x2x4x5+x1x2x5+x1x5+x2x4x5+x2x5+x4+1,x1x5+x4+1,x2x4x5+x2x5+1,1]
```

```
-- >>> f = p1 ∧ p2 ∧ p5 → p4
-- >>> no_f = proyeccion (no f)
-- >>> no_f
-- x1x2x4x5+x1x2x5
-- >>> ps'' = reglaIndependenciaKB (proyeccion p1) (S.insert (no_f) ps') (S.fromList
-- >>> ps''
-- fromList [0,x2x4x5+x2x5,x2x4x5+x2x5+x4x5+x4+1,x2x4x5+x2x5+1,x4x5+x4+1,1]
-- >>> ps''' = reglaIndependenciaKB (proyeccion p4) ps'' (S.fromList [])
-- >>> ps'''
-- fromList [0,x2x5,1]
-- >>> ps'''' = reglaIndependenciaKB (proyeccion p2) ps''' (S.fromList [])
-- >>> ps''''
-- fromList [0,x5,1]
-- >>> ps''''' = reglaIndependenciaKB (proyeccion p5) ps'''' (S.fromList [])
-- >>> ps'''''
-- fromList [0,1]
```

Luego, por el corolario 3.3.5, $\partial_{p_3}[K] \cup \{\neg F\}$ es inconsistente.

Capítulo 4

Herramienta `SAT_Solver`

En el presente capítulo se expondrá una herramienta para resolver el famoso problema de satisfacibilidad o más conocido como el problema *SAT*. Dicha herramienta se basará en la teoría presentada en los capítulos anteriores, mostrando así una de sus más relevantes aplicaciones.

Antes de tratar la herramienta, se pondrá al lector en situación dando una descripción detallada del problema de satisfacibilidad booleana; así como de su importancia, tanto para la comunidad científica internacional como para el resto de las personas.

Posteriormente, se describirán las tecnologías clave utilizadas en el desarrollo de la herramienta, que han dotado de estructura al proyecto además de concederle cierta robustez.

Finalmente, se realizará un análisis de eficiencia de la herramienta, durante el cual se detectaron ciertas "fugas de tiempo". Estas posibles mejoras se implementarán también dando lugar a una primera versión del `SAT-Solver`.

4.1. El problema SAT

Tal y como se definió en la página 18, una fórmula F se dice *satisfacible* si existe al menos una interpretación i de F que sea modelo de la fórmula. A partir de esta definición, es natural preguntarse cuándo una formula dada es satisfacible o no. Éste es el problema de la satisfacibilidad booleana o el comúnmente conocido como problema SAT. En definitiva consiste en saber si existe alguna forma de sustituir las variables proposicionales por *True* o *False*, de manera que la fórmula sea verdadera.

Es importante destacar que para una fórmula con 10 variables, existen $2^{10} = 1024$ valoraciones a comprobar. Si a esto se suma que para que una fórmula se considere "mínimamente interesante" (en el mundo empresarial, por ejemplo) debe tener cientos de variables; el orden de magnitud de comprobaciones que se han de hacer es similar al número de átomos que hay en el universo. Este crecimiento tan rápido de combinaciones se debe a la complejidad exponencial intrínseca al problema de satisfacibilidad.

En teoría de la complejidad computacional, es considerado el problema capital de la clase de complejidad NP al ser también un problema NP duro (NP-hard), y ganándose así el estatus de problema NP-completo.

Respecto a los problemas NP-completos, no hay manera eficiente conocida de obtener una solución. Es decir, el tiempo requerido para resolver el problema usando cualquier algoritmo conocido aumenta muy rápidamente a medida que el tamaño del problema crece.

Como consecuencia, el problema de determinar si es posible resolver estos problemas de forma rápida, llamado problema P versus NP (¿$P = NP$?), es uno de los principales problemas sin resolver de la informática actual. Es por esto que en el año 2000, el *Clay Mathematics Institute* lo declaró como el primer problema del milenio, junto a otros siete. Este título no es vano, ya que se cree que estos problemas marcarán el devenir de la comunidad científico-matemática.

Para comprender algo mejor la importancia del problema ¿$P = NP$? es importante conocer las consecuencias de su resolución. Si resultara que no son iguales, el impacto en la sociedad sería mínimo ya que únicamente se dejarían de buscar algoritmos de complejidad polinomial para resolver problemas de la clase NP.

Sin embargo, si se demostrase que ambas clases de complejidad son iguales, las implicaciones para la sociedad actual serían de una magnitud incomparable. Esto se debe a que la robustez de los sistemas criptográficos actuales se basan en la intratabilidad de ciertos problemas de la clase NP. Si éstos fueran resolubles en tiempo polinomial peligrarían todas las comunicaciones y las claves bancarias dejarían de ser seguras. Por otro lado, no todas las consecuencias son negativas ya que multitud de problemas de ámbito empresarial o social serían tratables; y por tanto, los costes se reducirían y los beneficios aumentarían.

Aunque, una vez vista la importancia del problema ¿$P = NP$? es natural preguntarse cómo podría resolverse. La respuesta es tan sencilla como dar un algoritmo que trabaje en tiempo polinomial en función del dato de entrada para un problema NP-completo, o demostrar que no existe dicho algoritmo.

El problema NP-completo objeto de la mayoría de estudios por parte de la comunidad científica es el problema SAT. Es tal su popularidad que cada año se publican multitud de nuevos algoritmos que dicen mejorar la eficiencia de resolución, y que se basan en las técnicas informáticas más innovadoras.

Con el objetivo de regular dicha competición, así como de identificar y promover nuevos retos relacionados con la resolución del problema SAT surge la *SAT Competition 2002*, enmarcada dentro del *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*. Y desde ese año, de forma anual o bianual, se celebra tal competición.

Enmarcando el trabajo aquí presente en el contexto actual, éste pretende sentar las bases de un programa que resuelva el problema SAT en un tiempo eficiente y que cumpla los estándares de dicha competición. Es por lo que a partir de este punto, la eficiencia será una máxima.

Además, para poder participar en dicha competición se debe seguir un estándar, cuya regla principal es que la fórmula (o base de conocimiento) de entrada está escrita en un fichero de texto en forma normal conjuntiva utilizando el formato $DIMACS$ (se tratará con detalle más adelante).

4.2. Gestión del proyecto

Es comúnmente aceptado que el proceso de creación de un programa consta de 5 etapas principales:

1. El desarrollo lógico del programa para resolver un problema en particular.
2. Escritura de la lógica del programa empleando un lenguaje de programación específico (codificación del programa).
3. Ensamblaje o compilación del programa hasta convertirlo en lenguaje de máquina.
4. Prueba y depuración del programa.
5. Desarrollo de la documentación.

En esta sección se tratarán las tecnologías auxiliares usadas en las distintas etapas a fin de comprender su importancia dentro del proyecto, así como la necesidad (ya comentada anteriormente) de usar las herramientas que están a disposición del programador para desarrollar una aplicación de mayor calidad en menor tiempo.
- El lenguaje de programación escogido es el lenguaje funcional Haskell, ya que encaja perfectamente con las necesidades del proyecto. Según la propia página web de Haskell, es un lenguaje que combina un fuerte sistema de tipos, inferencia de tipos y código de alto nivel, lo que afirman que ofrece al usuario la velocidad de desarrollo de lenguajes como Python o Ruby, además de una mayor robustez que lenguajes como Java o C++ .

A esto se le debe añadir que goza de una comunidad muy activa y un ecosistema de librerías muy rico, con multitud de librerías centradas en las matemáticas, y por ende, en los polinomios. Además, al ser objetivo del proyecto el definir diversas funciones matemáticas que actúen sobre el cuerpo $\mathbb{F}[\mathbf{x}]$, resulta natural escoger un lenguaje funcional tipado.

En lo que refiere a la tercera etapa, destacar la herramienta `stack` que, definiendo una jerarquía específica de ficheros en nuestro proyecto, asiste el ensamblado y compilación. Esto permite usar el programa en cualquier computador sin la necesidad de instalar librerías auxiliares ya que con tener instalado `stack` es suficiente. Aunque a la postre, el compilador primario que usa Haskell es GHC (Glasgow Haskell Compiler), un compilador en la vanguardia tecnológica, y de código libre, diseñado específicamente para Haskell.

Para la comprobación de la aplicación, el sistema de tipos de Haskell garantiza una cierta robustez, y es que asegura que cada función se utilice sobre los tipos para los que está diseñada. Por ejemplo, si intentamos calcular la suma de dos caracteres se devolverá un error al compilar.

Otro detalle a destacar de `stack` es que está diseñado de forma que la comprobación de los ejemplos sea muy natural y se haga automáticamente al compilar, con ayuda de la librería `doctest`. Cada vez que en el código hay (>>>) se ejecuta lo escrito a continuación y se corrobora si devuelve lo que aparezca en la línea siguiente (si es que se devolviese algo). Por ejemplo, en

```
-- |
-- >>> 2+2
-- 4
```

se comprueba si 2+2 devuelve 4. Notar que esto concede al proyecto de mucha robustez frente a modificaciones. Es decir, si se cambia el código de cierta función f buscando una mayor eficiencia pero se comete algún error, se puede detectar al hacer las comprobaciones de los casos base, tanto de esta función como de alguna otra en la que intervenga f.

Otra ventaja, relacionada con las comprobaciones, que ofrece el lenguaje Haskell es la posibilidad de implementar propiedades matemáticas de las funciones o los tipos. Estas propiedades pueden ser esenciales en el desarrollo teórico pues pueden justificar, por ejemplo, la corrección de la aplicación o parte de ella.

Por lo tanto, es parte fundamental del proyecto verificar dichas propiedades mediante la librería `quickCheck`. Esta librería manda ejecutar 100 ejemplos y casos límites (el número 0, listas vacías, etc) para ver si sobre ellos se sigue verificando la propiedad. En el proyecto se comprobarán estas propiedades cada vez que se compile ya que junto a cada propiedad *prop* se ha incluido:

```
-- |
-- >>> quickCheck prop
--  +++ OK, passed 100 tests
```

Así que, si en algún momento del desarrollo se deja de cumplir dicha propiedad para alguna instancia, el sistema devuelve el error.

En la etapa de documentación se ha utilizado una versión modificada del lenguaje Haskell, llamada Haskell literario. Ésta permite escribir código LaTeXen el mismo archivo en el que se escribe el código Haskell. Por ejemplo, si se escribe en el archivo `.lhs`:

```
Es un ejemplo
\begin{code}
ejemplo = "helloworld"
\end{code}
```

Al compilar se verá lo siguiente en el pdf:

Es un ejemplo

```
ejemplo = "helloworld"
```

Con la ventaja principal de que la función *ejemplo* se puede utilizar normalmente, como si se hubiese escrito en un archivo `.hs` típico.

Por último, se ha utilizado la herramienta `git` de control de versiones, que junto a la plataforma `GitHub` ha sido de gran ayuda en la depuración del código y la búsqueda de eficiencia.

4.3. La herramienta

A continuación, se describirán los distintos módulos de la herramienta, detallando los procesos más relevantes. Es importante destacar que los cálculos llevados a cabo por el programa se dividen principalmente en dos etapas:

1. Preprocesado del fichero de entrada.

2. Decisión por saturación con la regla de independencia.

4.3.1. Fichero de entrada y el formato `DIMACS`

Previo a la descripción del formato `DIMACS` es necesaria una definición formal de qué es la forma normal conjuntiva. Con este objetivo se verán antes dos definiciones de la lógica proposicional.

Definición 4.3.1. Un literal es una fórmula atómica o su negación. Se dice que un literal es *positivo* si se trata de un átomo y *negativo* si es la negación de un átomo.

Por ejemplo, p_1, p_2 ó $\neg p_1$ son literales.

Definición 4.3.2. Una cláusula es una disyunción de literales. En otras palabras, es una colección finita de literales que es verdadera si alguno de ellos lo es.

Por ejemplo, $p_1 \vee p_2$ ó $p_1 \vee \neg p_2 \vee \neg p_3$ son cláusulas. Ya se está en condiciones de definir la forma normal conjuntiva.

Definición 4.3.3. Dada una fórmula proposicional F se dice que está en forma normal conjuntiva si se trata de una conjunción de cláusulas.

Con respecto al problema de satisfacibilidad es importante saber el que criterio debe cumplir una fórmula en forma normal conjuntiva para ser verdadera. Esta condición es que en todas sus cláusulas debe haber al menos un literal que sea verdadero.

Otra propiedad que se debe conocer sobre la forma normal conjuntiva es que dada cualquier fórmula proposicional existe otra equivalente en forma normal conjuntiva.

El algoritmo que la calcula es el siguiente [6] :

Algoritmo: Aplicando a una fórmula F los siguientes pasos se obtiene una forma normal conjuntiva de F:

1. **Eliminar los bicondicionales usando la equivalencia**
$$A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A) \quad (1)$$
2. **Eliminar los condicionales usando la equivalencia**
$$A \rightarrow B \equiv \neg A \vee B \quad (2)$$
3. **Interiorizar las negaciones usando las equivalencias**
$$\neg(A \wedge B) \equiv \neg A \vee \neg B \quad (3)$$
$$\neg(A \vee B) \equiv \neg A \wedge \neg B \quad (4)$$
$$\neg\neg A \equiv A \quad (5)$$
4. **Interiorizar las disyunciones usando las equivalencias**
$$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C) \quad (6)$$
$$(A \wedge B) \vee C \equiv (A \vee C) \wedge (B \vee C) \quad (7)$$

Como ya se ha comentado, se pretende presentar la herramienta a una competición, por tanto, debe cumplir ciertos requisitos para poder participar. El principal y más importante es que las instancias del problema SAT que debe resolver vienen dadas en lo que se denomina el formato `DIMACS`. Un ejemplo de cómo se codificaría la fórmula $(p_1 \vee \neg p_5 \vee p_4) \wedge (\neg p_1 \vee p_5 \vee p_3 \vee p_4) \wedge (\neg p_3 \vee \neg p_4)$ en formato `DIMACS` es

```
c

c start with comments

c

c

p cnf 5 3

1 -5 4 0

-1 5 3 4 0

-3 -4 0
```

Dicho formato es una estandarización simplificada de la forma normal conjuntiva de una fórmula. Las instancias serán archivos de texto (`.txt` o `.cnf`) con la siguiente estructura:

1. En primer lugar, se encuentran las líneas de comentarios. Se identifican gracias a que están encabezadas por la letra `c`. En el ejemplo, son las cuatro primeras líneas.

2. En segundo lugar, hay una línea distinguida que indica las propiedades de la fórmula como, por ejemplo, que está en forma normal conjuntiva (`cnf`). El primer número que aparece indica el mayor índice de una variable proposicional (normalmente coincide con el número de variables proposicionales) mientras que el segundo cuántas cláusulas.

3. Finalmente, en cada línea se encuentra codificada una cláusula distinta según las siguientes indicaciones:

   - Cada número entero representa un literal.
   - El signo del número, positivo o negativo, indica si dicho literal es positivo o negativo, respectivamente.
   - El final de la cláusula se codifica con un 0.

4.3.2. Preprocesado

Las funciones que intervienen en el preprocesado transforman un archivo `.txt` ó `.cnf` en formato `DIMACS` en el conjunto de polinomios que les corresponde según la función π. A continuación, se presentan estas funciones.

```
module Preprocesado where


import Haskell4Maths (var
                     , zerov)
import F2
import Transformaciones ( phi)

import Data.List (foldl')
import qualified Data.Set as S
```

La función `(literalAPolinomio lit)` recibe una cadena que codifica un literal en formato `DIMACS` y devuelve un par `(p,v)` donde p es el polinomio correspondiente a dicho literal (la proyección por π) y v es la variable que interviene en dicho polinomio.

```
-- | Por ejemplo,
--
-- >>> literalAPolinomio "0"
-- (0,0)
-- >>> literalAPolinomio "1"
-- (x1,x1)
-- >>> literalAPolinomio "-1"
-- (x1+1,x1)

literalAPolinomio :: String -> (PolF2,PolF2)
literalAPolinomio "0"       = (zerov,zerov)
literalAPolinomio ('-':lit) = (1 + x,x)
                   where x = var ('x':lit)
literalAPolinomio lit       = (x,x)
                   where x = var ('x':lit)
```

La función `(clausulaAPolinomio cs)` recibe una cadena que codifica una cláusula en formato `DIMACS` y devuelve un par `(p,vs)` donde p es el polinomio que le corresponde según la función π y vs las variables que intervienen en él.

```
-- | Por ejemplo,
--
-- >>> clausulaAPolinomio ["1"]
-- (x1,fromList [x1])
-- >>> clausulaAPolinomio ["1","-2"]
-- (x1x2+x2+1,fromList [x1,x2])

clausulaAPolinomio :: [String] -> (PolF2, S.Set (PolF2))
clausulaAPolinomio (c:cs) | c == "c" || c == "p" = (1, S.empty)
clausulaAPolinomio cs = aux $ foldl' (\acc x -> disj (literalAPolinomio x) acc)
                                     (zerov,S.empty) cs
  where aux (a,b) = (phi a,b)
        disj (x,v) (y,vs) | v == zerov = (y,vs)
                          | otherwise   = (x + y + x*y, S.insert v vs)
```

Además de las anteriores, serán necesarias tres funciones predefindas de Haskell. La primera es la función `readFile`, que transforma el archivo `.txt` en una cadena de caracteres.

También se usará la función `lines` que divide una cadena de caracteres en una lista de cadenas de la siguiente forma: empieza leyendo desde el principio y cada vez que

se encuentra un salto de línea acaba la cadena y comienza una nueva.

La tercera y última función necesaria es `words`, homóloga a la anterior pero que divide una cadena en función de los espacios.

Finalmente, ya se está en disposición de implementar la función que reciba un archivo `.txt` y devuelve un conjunto de polinomios. Además, se devolverá una lista ordenada que servirá más adelante para indicar las variables que aún no se han omitido.

En definitiva, la función `(dimacsAPolinomios f)` recibe el archivo f y devuelve el par `(ps,vs)`; donde ps es el conjunto de polinomios correspondientes (por la función π) a la fórmula codificada en el archivo f, mientras que vs es la lista de variables que intervienen en alguno de estos polinomios.

```
-- | Por ejemplo,
--
-- >>> dimacsAPolinomios "exDIMACS/easy/example1.txt"
-- (fromList [x1x2+x1+x2,1],[x1,x2])
-- >>> dimacsAPolinomios "exDIMACS/easy/example4.txt"
-- (fromList [x1x2+x1+x2,x1x2+x1+1,x1x2+x2+1,x1x2+1,1],[x1,x2])


dimacsAPolinomios f = do
  s0 <- readFile f
  return $
    aux1 $ (foldr (\x acc -> (aux2 ((clausulaAPolinomio . words) x) acc))
             (S.empty,S.empty)) $ lines $ s0
     where aux1 (a,b) = (a,S.toList b)
           aux2 (a,b) (acc,vs) = (S.insert  a acc, S.union vs b)
```

Con el objetivo de comprender las dimensiones del problema con el que se trabaja, se muestra a continuación un ejemplo del uso de la función `dimacsAPolinomios` en el fichero (no trivial) de menor tamaño:

```
> dimacsAPolinomios "exDIMACS/medium/exampleSat0.txt"
(fromList [x1x10x38+x1x10+x1x38+x1+x10x38+x10+x38,
x10x48x59+x10x59+x48x59+x59+1,x10x76x88+x10x76+x76x88+x76+1,
x14x34x50+x14x34+x14x50+x14+1,x14x59x87+x59x87+1,
x16x28x84+x16x28+x16x84+x16+x28x84+x28+x84,x18x55x64+x55x64+1,
x19x3x30+x19x3+1,x19x57x93+x19x57+x19x93+x19+1,x2x21x98+x21x98+1,
x20x4x80+x20x80+1,x23x7x72+x23x72+1,x28x60x8+x28x60+1,x3x35x96+x35x96+1,
x31x42x89+x31x42+x42x89+x42+1,x31x5x97+x5x97+1,x35x47x64+x35x47+1,
x35x48x58+x35x48+x35x58+x35+1,x36x56x78+x36x78+x56x78+x78+1,
x4x65x94+x4x65+x4x94+x4+1,x44x76x78+x44x78+1,x48x52x63+x48x63+1,
```

```
x52x94x99+x52x94+x94x99+x94+1,x55x63x90+x55x63+1,
x59x75x9+x59x75+x59x9+x59+x75x9+x75+x9,x65x83x89+x65x83+x65x89+x65+1,
x68x84x88+x68x88+1,x75x86x89+x75x86+1,x81x88x92+x81x88+1,1],
[x1,x10,x14,x16,x18,x19,x2,x20,x21,x23,x28,x3,x30,x31,x34,x35,x36,x38,
x4,x42,x44,x47,x48,x5,x50,x52,x55,x56,x57,x58,x59,x60,x63,x64,x65,x68,
x7,x72,x75,x76,x78,x8,x80,x81,x83,x84,x86,x87,x88,x89,x9,x90,x92,x93,
x94,x96,x97,x98,x99])
```

4.3.3. Saturación

Una vez que se tiene el conjunto de polinomios basta saturar dicho conjunto usando la regla de independencia. Las funciones encargadas de realizar este proceso son: `omiteVariableKB` y `saturaKB`. El módulo en el que se implementan estas se llama `Saturacion`.

```
module Saturacion where

import Haskell4Maths (var
                     , vars
                     , zerov)
import F2 (PolF2)
import Regla (reglaIndependencia)
import Preprocesado (dimacsAPolinomios)

import System.Environment
import qualified Data.Set as S
```

Sin embargo, antes de implementar estas funciones hay una modificación que mejora la eficiencia de las funciones `(reglaIndependenciaAux` y `(reglaIndependenciaKB)`.

Esta mejora se basa en el hecho de que si en algún momento de la computación hay un cero en el conjunto de polinomios (que traducido a fórmula es un $\bot$) éste permanecerá hasta finalizar la saturación. De hecho, tras saturar dicho conjunto, será el único polinomio que esté en el conjunto (junto con el 1, correspondiente a la tautología $\top$, y se puede obviar si aparece acompañado). Por tanto, aplicando el corolario 3.3.5, la base de conocimiento original es inconsistente.

Teniendo en cuenta lo comentado anteriormente, se pueden modificar las definiciones anteriores de `(reglaIndependenciaAux` y `(reglaIndependenciaKB` para obtener un método de saturación más eficiente.

Para ello basta añadir al bucle de la primera la siguiente línea de código:

```
| dR == 0 = S.fromList [0]
```

Quedando la implementación de dicha función tal y como sigue:

```
reglaIndependenciaAux :: PolF2 -> PolF2 -> S.Set PolF2 ->
                         S.Set PolF2 -> S.Set PolF2
reglaIndependenciaAux v p ps acum
  | S.null ps = acum
  | dR == 0   = S.fromList [0]
  | otherwise = reglaIndependenciaAux v p ps' (S.insert dR acum)
                where (p',ps') = S.deleteFindMin ps
                      dR       = reglaIndependencia v p p'
```

En cuanto a la función `reglaIndependenciaKB`, si en algún momento de la computación el acumulador es el conjunto cuyo único elemento es el 0, querrá decir que se ha obtenido al aplicar la regla de independencia. Por tanto la base de conocimiento de la que proviene dicho conjunto de polinomios es inconsistente. Para implementar esto sólo se debe añadir al principio del bucle la siguiente línea:

```
| acum == S.fromList [0] = S.fromList [0]
```

Y por tanto, la función queda:

```
reglaIndependenciaKB :: PolF2 -> S.Set PolF2 ->
                  S.Set PolF2 -> S.Set PolF2
reglaIndependenciaKB v pps acum
  | acum == S.fromList [0] = S.fromList [0]
  | S.null pps   = acum
  | otherwise    = reglaIndependenciaKB v ps
                   (reglaIndependenciaAux v p pps acum)
      where (p,ps) = S.deleteFindMin pps
```

Usando esta mejora, ya se está en condiciones de implementar las funciones que ejecutan la saturación.

La función `(omiteVariableKB v ps)` devuelve el conjunto de polinomios obtenido tras aplicar la regla de independencia respecto de la variable v entre cada uno de los polinomios del conjunto *ps*. En caso de que como resultado de aplicar dicha regla se obtenga el polinomio cero, los cálculos se detendrán y se devolverá un conjunto unitario cuyo único elemento sea el cero.

```
-- | Por ejemplo,
-- >>> x1 = (var "x1") :: PolF2
-- >>> x2 = (var "x2") :: PolF2
-- >>> omiteVariableKB x2 (S.fromList [x2,x1*x2,x1+1])
-- fromList [x1,x1+1,1]
-- >>> omiteVariableKB x1 (S.fromList [x1,x1+1,1])
-- fromList [0]

omiteVariableKB :: PolF2 -> S.Set PolF2 -> S.Set PolF2
omiteVariableKB v ps = reglaIndependenciaKB v ps1 ps2
       where (ps1,ps2) = S.partition (\p -> elem v (vars p)) ps
```

Conviene destacar que se divide el conjunto original de polinomios en dos, según si ocurre la variable que se quiere omitir o no. De esta forma se evitan cálculos redundantes.

La función que satura el conjunto *ps* usando la regla de independencia según un orden dado por la lista *vs* es `(saturaKB (ps,vs))`. Dicha función devuelve `True` si la fórmula original de la que proviene el conjunto de polinomios *ps* es satisfacible, y `False` si es insatisfacible.

```
-- | Por ejemplo,
--
-- >>> saturaKB (S.fromList[1],[])
-- True
-- >>> x1 = (var "x1") :: PolF2
-- >>> saturaKB (S.fromList[x1,x1+1],[x1])
-- False

saturaKB :: (S.Set PolF2,[PolF2]) -> Bool
saturaKB (ps,[])                  = S.notMember 0 ps
saturaKB (ps,v:vs) | S.member 0 ps = False
                   | otherwise     = saturaKB (omiteVariableKB v ps, vs)
                      where nextVPS = omiteVariableKB v ps
```

Finalmente, se combinan ambas etapas (Preprocesado y Saturación) en la función `satSolver f`. Esta función recibe un fichero donde se codifica una fórmula en formato `DIMACS` y devuelve `True` si dicha fórmula es satisfacible y `False` en caso contrario.

```
-- | Por ejemplo,
--
-- >>> satSolver "exDIMACS/easy/example1.txt"
-- True
-- >>> satSolver "exDIMACS/easy/example4.txt"
-- False
-- >>> satSolver "exDIMACS/medium/exampleSat0.txt"
-- True
satSolver f = do
  f' <- dimacsAPolinomios f
  return (saturaKB f')
```

Y así queda implementada una primera versión de la herramienta.

4.4. Análisis de la herramienta

En esta sección se probará la herramienta con instancias del problema, con el objetivo de detectar cotas de eficiencia. Las instancias se almacenan en el directorio `exDIMACS`, organizadas en carpetas según su dificultad.

4.4.1. Directorio `easy`

Este directorio contiene cuatro ejemplos muy sencillos, de hecho, las fórmulas que codifican sólo tienen 2 variables (p y q). El objetivo de estos ejemplos es servir como test para comprobar que el funcionamiento de la herramienta es el deseado.

Ejemplo 1

El archivo se llama `example1.txt`:

```
c example 1
c
c f = (p ∨ q)
c
p cnf 2 1
1 2 0
```

Y codifica la fórmula:

$$(p \vee q)$$

La ejecución en máquina es:

```
>>> satSolver "exDIMACS/easy/example1.txt"
True
(0.00 secs, 586,368 bytes)
```

Ejemplo 2

El archivo se llama `example2.txt`:

```
c example 2
c
c f = (p ∨ q) ∧ (¬p ∨ q)
c
p cnf 2 2
1 2 0
-1 2 0
```

Y codifica la fórmula:

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

La ejecución en máquina es:

```
>>> satSolver "exDIMACS/easy/example2.txt"
True
(0.00 secs, 792,784 bytes)
```

Ejemplo 3

El archivo se llama `example3.txt`:

```
c example 3
c
c f = (p ∨ q) ∧ (¬p ∨ q) ∧ (p ∨ ¬q)
c
p cnf 2 3
1 2 0
-1 2 0
1 -2 0
```

Y codifica la fórmula:

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

La ejecución en máquina es:

```
>>> satSolver "exDIMACS/easy/example3.txt"
True
(0.00 secs, 1,047,600 bytes)
```

Ejemplo 4

El archivo se llama `example4.txt`:

```
c example 4
c
c f = (p ∨ q) ∧ (¬p ∨ q) ∧ (p ∨ ¬q) ∧ (¬p ∨ ¬q)
c
p cnf 2 4
1 2 0
-1 2 0
1 -2 0
-1 -2 0
```

Y codifica la fórmula:

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

La ejecución en máquina es:

```
>>> satSolver "exDIMACS/easy/example4.txt"
False
(0.01 secs, 1,361,672 bytes)
```

4.4.2. Directorio `medium`

Tras comprobar que la herramienta responde de la forma deseada con los ejemplos anteriores, conviene probar su eficiencia. Para ello, se construye una batería de ejemplos truncando el archivo `sat100.cnf` del directorio `hard`.

Al ser conjuntos de cláusulas contenidas en un conjunto de cláusulas satisfacible son también satisfacibles y por tanto, el resultado de usar la herramienta debe ser `True`.

Únicamente se tratarán los tres primeros ejemplos ya que el resto no se ejecuta en un tiempo razonable.

Ejemplo 0

El archivo se llama `exampleSat0` y codifica una fórmula en forma normal conjuntiva con 30 cláusulas en las que intervienen hasta 59 variables distintas.

Destacar que tanto en este ejemplo como en los próximos, en cada cláusula habrá exactamente 3 literales.

La ejecución en máquina es:

```
>>> satSolver "exDIMACS/medium/exampleSat0.txt"
True
(0.02 secs, 10,307,480 bytes)
```

El archivo es:

```
c
c    clause length = 3
c
p cnf 59 30
30 -19 -3 0
89 31 -42 0
1 10 38 0
2 -21 -98 0
36 56 -78 0
14 -59 -87 0
89 -75 -86 0
-20 -80 4 0
-63 90 -55 0
59 75 9 0
-5 31 -97 0
48 -35 58 0
28 84 16 0
65 -4 94 0
-72 -23 7 0
18 -64 -55 0
-96 3 -35 0
89 -65 83 0
8 -60 -28 0
34 50 -14 0
64 -47 -35 0
-19 57 93 0
52 99 -94 0
-59 10 48 0
-78 -44 76 0
-63 -48 52 0
-88 84 -68 0
10 88 -76 0
92 -81 -88 0
```

Ejemplo 5

El archivo se llama `exampleSat1.txt` y codifica una fórmula satisfacible con 91 cláusulas y 82 variables. Por razones de espacio se omitirá mostar el archivo. La ejecución en máquina es:

```
>>> satSolver "exDIMACS/medium/exampleSat1.txt"
True
(5.87 secs, 2,122,138,840 bytes)
```

Ejemplo 6

El archivo se llama `exampleSat2.txt` y codifica una fórmula satisfacible de mayor tamaño que la anterior, de hecho, la contiene. Por razones de espacio se omitirá mostar el archivo. La ejecución en máquina es:

```
>>> satSolver "exDIMACS/medium/exampleSat2.txt"
True
(139.13 secs, 34,754,329,424 bytes)
```

4.4.3. Directorio `hard`

En este directorio se encuentras tres archivos `.cnf` que son tres ejemplos oficiales de instancias dadas por la organización de la competición *SAT* en la que se pretende participar.

Archivo `sat100.cnf`

Este ejemplo consta de 100 variables en 430 cláusulas. La fórmula es satisfacible pero tiempo de ejecución sobrepasa la hora.

Archivo `sat250.cnf`

Este ejemplo consta de 250 variables en 1065 cláusulas. La fórmula es satisfacible aunque el tiempo de ejecución excede la hora.

Archivo `unsat250.cnf`

Este ejemplo consta de 250 variables en 1065 cláusulas. La fórmula es insatisfacible y la ejecución queda:

```
>>> satSolver "exDIMACS/hard/unsat250.cnf"
False
(0.05 secs, 34,873,832 bytes)
```

4.4.4. Heurísticas

Ya en el ejemplo 5, el tiempo de ejecución es demasiado alto. Sin embargo, en el ejemplo 6 y posteriores (exceptuando `unsat250.cnf`) el coste en tiempo excede lo razonable, por lo que se pone de manifiesto la necesidad de una mejora en la implementación.

Tras diversas pruebas, la mejora que se ha implementado se debe al hecho observado de que aunque el resultado tras saturar permanezca invariante si se cambia el orden de las variables a omitir, el tiempo de ejecución sí cambia.

A raíz de esto se probarán experimentalmente varias heurísticas; y en base a estos experimentos, se escogerá la que tenga mejores resultados. Es importante tener en cuenta que no es seguro que el orden escogido sea el óptimo ya que nos basamos en un criterio absolutamente empírico.

Se define el módulo `Heuristicas`, en el que se implementarán dichas heurísticas.

```
module Heuristicas where

import Haskell4Maths (var
                     , vars)
import F2
import Preprocesado (dimacsAPolinomios)
import Saturacion (omiteVariableKB)

import Data.List (foldl', sortOn)
import qualified Data.Set as S
```

A continuación, se define el tipo `Heuristica`. Este es una función que recibe un conjunto de polinomios y una lista de sus variables, y devuelve una lista ordenada de dichas variables.

```
type Heuristica = S.Set PolF2 -> [PolF2] -> [PolF2]
```

La primera heurística es la inducida por el orden monomial, en este caso el lexicográfico. Como por construcción, *vs* ya está ordenada de tal forma, la heurística `(heuristicaOM ps vs)` devuelve invariante la lista *vs*.

```
-- | Por ejemplo:
--
-- >>> [x1,x2] = map var ["x1","x2"] :: [PolF2]
-- >>> heuristicaOrdMon (S.fromList [x1,x2,x1+1]) [x1,x2]
-- [x1,x2]
heuristicaOrdMon :: Heuristica
heuristicaOrdMon ps vs = vs
```

La segunda heurística (`heuristicaFrecuencia ps vs`) devuelve una lista con las variables de *vs* ordenadas de menor a mayor frecuencia de aparición en los polinomios de *ps*.

```
-- | Por ejemplo:
--
-- >>> [x1,x2] = map var ["x1","x2"] :: [PolF2]
-- >>> heuristicaFrecuencia (S.fromList [x1,x2,x1+1]) [x1,x2]
-- [x2,x1]

heuristicaFrecuencia :: Heuristica
heuristicaFrecuencia ps vs = sortOn frecuencia vs
   where frecuencia v = length ( filter (== v) ps')
         ps' = foldl' (\acc p -> (vars p) ++ acc) [] ps
```

Finalmente, la heurística (`heuristicaFrecRev ps vs`) devuelve una lista con las variables de *vs* ordenadas de mayor a menor frecuencia de aparición en los polinomios de *ps*.

```
-- | Por ejemplo:
--
-- >>> [x1,x2] = map var ["x1","x2"] :: [PolF2]
-- >>> heuristicaFrecRev (S.fromList [x1,x2,x1+1]) [x1,x2]
-- [x1,x2]

heuristicaFrecRev :: Heuristica
heuristicaFrecRev ps = (reverse . heuristicaFrecuencia ps)
```

Para introducir esta variante es necesario que se ordene la lista de variables cada vez que se olvide una de ellas ya que también se modifican los polinomios y, por ejemplo, las frecuencias cambiarían. Es por esto que se redefinirá las funciones `saturaKB` y `satSolver` tal y como sigue:

```
saturaKB :: (S.Set PolF2,[PolF2]) -> Heuristica -> Bool
saturaKB (ps,[])   h                = S.notMember 0 ps
saturaKB (ps,v:vs) h | S.member 0 ps = False
                     | otherwise     = saturaKB (aux, h aux vs) h
                       where aux     = omiteVariableKB v ps

satSolver h f = do
  f' <- dimacsAPolinomios f
  return (saturaKB f' h)
```

Quedando la solución del ejemplo 5:

```
>>> satSolver heuristicaOrdMon "exDIMACS/medium/exampleSat1.txt"
True
(5.82 secs, 2,122,150,944 bytes)
>>> satSolver heuristicaFrecuencia "exDIMACS/medium/exampleSat1.txt"
True
(0.24 secs, 62,955,680 bytes)
>>> satSolver heuristicaFrecRev "exDIMACS/medium/exampleSat1.txt"
True
(7.08 secs, 3,372,336,472 bytes)
```

Mientras que la del ejemplo 6:

```
>>> satSolver heuristicaOrdMon "exDIMACS/medium/exampleSat2.txt"
True
(137.98 secs, 34,754,335,488 bytes)
>>> satSolver heuristicaFrecuencia "exDIMACS/medium/exampleSat2.txt"
True
(0.30 secs, 92,607,744 bytes)
>>> satSolver heuristicaFrecRev "exDIMACS/medium/exampleSat2.txt"
Interrupted.
```

Tras varios minutos de espera queda patente que la tercera heurística no es eficiente. La segunda (`heuristicaFrecuencia`) con el ejemplo 7 y con `unsat250.cnf` da los siguientes resultados:

```
>>> satSolver heuristicaFrecuencia "exDIMACS/medium/exampleSat3.txt"
True
(5.87 secs, 2,317,863,248 bytes)
>>> satSolver heuristicaFrecuencia "exDIMACS/hard/unsat250.cnf"
False
(0.06 secs, 34,872,144 bytes)
```

En función de estos resultados, la heurística escogida es `heuristicaFrecuencia`. Sin embargo, el tiempo sigue siendo demasiado alto si se quiere resolver el ejemplo 8 o algunas de las dos instancias satisfacibles del directorio `hard`.

Capítulo 5

Conclusión

En este capítulo se hace una breve introducción a la programación funcional en Haskell suficiente para entender su aplicación en los siguientes capítulos. Para una introducción más amplia se pueden consultar los apuntes de la asignatura de Informática de 1º del Grado en Matemáticas ([2]).

El contenido de este capítulo se encuentra en el módulo `PFH`

```
module PFH where
import Data.List
```

5.1. Introducción a Haskell

En esta sección se introducirán funciones básicas para la programación en Haskell. Como método didáctico, se empleará la definición de funciones ejemplos, así como la redefinición de funciones que Haskell ya tiene predefinidas, con el objetivo de que el lector aprenda "*a montar en bici, montando*".

A continuación se muestra la definición `(cuadrado x)` es el cuadrado de `x`. Por ejemplo, La definición es

```
-- |
-- >>> cuadrado 3
-- 9
-- >>> cuadrado 4
-- 16
cuadrado :: Int -> Int
cuadrado x = x * x
```

.

Bibliografía

[1] J. C. Agudelo-Agudelo, C. A. Agudelo-González, and O. E. García-Quintero. On polynomial semantics for propositional logics. *Journal of Applied Non-Classical Logics*, 26(2):103–125, 2016.

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

[3] G. A. Aranda-Corral, J. Borrego-Díaz, and M. M. Fernández-Lebrón. *Conservative Retractions of Propositional Logic Theories by Means of Boolean Derivatives: Theoretical Foundations*, pages 45–58. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[4] D. Kapur and P. Narendran. An equational approach to theorem proving in first-order predicate calculus. *SIGSOFT Softw. Eng. Notes*, 10(4):63–66, Aug. 1985.

[5] J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *J. Artif. Int. Res.*, 18(1):391–443, May 2003.

[6] J. A. A. J. y Andrés Cordón Franco. Diapositivas de Lógica Informática. https://www.cs.us.es/~jalonso/cursos/li-03/temas/tema-2.pdf, 2003-04.

Índice alfabético