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 ([1]).

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 es 1; mientras que si la fórmula es falsa, el polinomio vale 0.

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

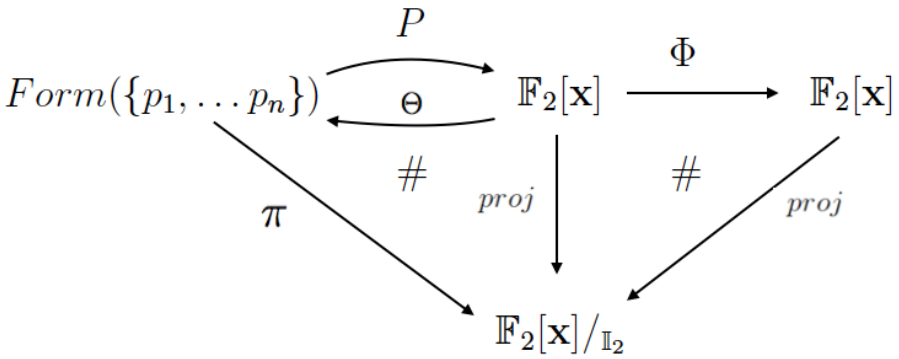

Figura 2.1: Relación entre las fórmulas proposicionales y $\mathbb{F}[x]_2$

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.

```
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
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 lo llama 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 además 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 comunmente aceptado (y así será en este trabajo) llamar al conjunto de las variables ($\mathcal{L} = \{p_1, ..., p_n\}$) lenguaje proposicional. A la hora de implementar las variables proposicionales se representarán por 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.

Finalmente, se define el tipo de dato de las fórmulas proposicionales (`FProp`) de la siguiente manera:

```
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. A continuación, 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 chequearlas. 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:

```
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 de sustituir cada ocurrencia de p en F por la fórmula G.

Se implementa $f\{p/g\}$ en 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:

```
-- | 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 2.1.4. 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. La función `(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. Para implementarlo se necesita de dos funciones auxiliares.

`(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 f = subsequences (simbolosPropForm f)
```

`(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 2.1.5. Una fórmula F se dice válida si toda interpretación i de F es modelo de la fórmula. La función `(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 2.1.6. 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 f = modelosFormula f == []
```

Definición 2.1.7. 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 f = modelosFormula f /= []
```

2.1.4. Bases de conococimiento

Definición 2.1.8. Una *base de conocimiento* o *Knowledge Basis* (*KB*) es un conjunto finito de fórmulas proposicionales. 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 2.1.9. Análogamente al caso de una única fórmula, se dice que i es *modelo* de K ($i \models K$) si lo es de cada una de las fórmulas de K. La función (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 k = foldr (\f acc -> (esModeloFormula i f) && acc) True k
```

Al conjunto de modelos de K se le denota por $Mod(K)$. `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 2.1.10. Un conjunto de fórmulas se dice *consistente* si y sólo si tiene al menos un modelo. Una definición alternativa es que dicho conjunto de fórmulas es *consistente* si y sólo si para toda fórmula f no es posible deducir tanto f como $\neg f$ a partir de él.

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 k = modelosKB k /= []
```

Definición 2.1.11. Un conjunto de fórmulas se dice *inconsistente* si y sólo si no tiene modelo. Una definición alternativa es que dicho conjunto de fórmulas es *consistente* si y sólo si alguna fórmula f es posible deducir tanto f como $\neg f$ a partir de él.

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 k = modelosKB k == []
```

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 aperecerá con asiduedad en el desarrollo del trabajo.

Existe una manera de caracterizar la relación de consecuencia lógica basada en los axiomas y las reglas de inferencia. Sin embargo, se abordará la definición desde otra perspectiva, teniendo en cuenta su implementación.

Definición 2.1.12. 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 comprobaciones no son más que meros chequeos de que una propiedad se cumple para una batería de ejemplos. En ningún caso se puede confiar en que dicha propiedad se cumple el 100problema más costoso y en ningún caso trivial.

Proposición 2.1.13. 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 2.1.14. 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 k' = foldr (\f acc -> acc && esConsecuencia k f) True k'
```

2.1.7. Equivalencia

Definición 2.1.15. Sean F y G dos fórmulas proposicionales, se dice que son equivalentes ($F \equiv G$) si tienen el mismo contenido 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 2.1.16. Dadas K y K' dos bases de conocimiento, se dice que son equivalentes ($K \equiv K'$) si $K \models K'$ y $K' \models K$.

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 2.1.17. 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 2.1.18. Sean K y K' bases de conocimiento, se dice que K es una *extension* de K' si $\mathcal{L}(K') \subseteq \mathcal{L}(K)$ y

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

Si nos restringimos a las fórmulas consecuencia de una base de conocimiento en el lenguaje de la otra:

Definición 2.1.19. Sean K y K' bases de conocimiento, se dice que K es una *extension 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 2.1.20. 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 $[K, \mathcal{L}']$. El problema real es dar una axiomatización de dicho conjunto de fórmulas.

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 empleabilidad 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 tutarial muy completo.

Sin embargo, no resulta intuitivo aislar la librería de polinomios por lo que para realizar las pruebas se 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 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. Finalmente, destacar que la documentación no es escasa por lo que su uso resulta tedioso.

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 `Haskell4Maths` que aparecerán de forma recurrente en el resto del trabajo.

```
module Haskell4Maths
    ( F2
    , 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
--    (Lex, Glex, Grevlex, var, mindices, lm, lt, (%%), vars)
```

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 un mapeo 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 jk. 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.

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

```
-- >>> let [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 es espacio vectorial deque es un polinomio, está formada por monomios. El tipo de dato monomio está formado por un coeficiente i y una lista de pares. En el caso de los polinomios, 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 se pueden entender como 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,
--
-- >>> let [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 ($i \in k$).

```
-- |
-- >>> let [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:

```
-- |
-- >>> let [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.

```
-- |
-- >>> let [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.

```
-- |
-- >>> let [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 $\mathbb{F}_2[\mathbf{x}]$.

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

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 en el apartado anterior es definir el tipo de dato que representa $\mathbb{F}_2[\mathbf{x}]$ (`PolF2`) , así como sus variables (`VarF2`):

```
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 de la mónada `Box`. Este artificio es necesario ya que no se pueden declarar instancias (como se hará a continuación) repetidas 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 la mónada `Bos`, declaramos la instancia `Show`:

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

Para poder definir propiedades que involucren a estos tipos de datos y chequearlas 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. A partir de la suma de éstos se obtendrán los polinomios:

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

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

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

```
-- |
-- >>> 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: $\forall p, q, r \in \mathbb{F}_2[\mathbf{x}](p + (q + r) = (p + q) + r)$.

```
-- |
-- >>> 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:

```
-- |
-- >>> 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: $\forall p \in \mathbb{F}_2[\mathbf{x}] : p + p = 0$.

```
-- |
-- >>> 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: $\forall p, q \in \mathbb{F}_2[\mathbf{x}](p * q = q * p)$.

```
-- |
-- >>> 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: $\forall p, q, r \in \mathbb{F}_2[\mathbf{x}](p * (q * r) = (p * q) * r)$.

```
-- |
-- >>> 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:

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

```
-- |
-- >>> 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)
```

2.2.4. Transformaciones entre fórmulas y polinomios

La traducción o transformación de la lógica proposicional en álgebra polinomial viene dada por [2] 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, el mapeo $P : Form(\mathcal{L} \to \mathbb{F}_2[x])$ 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 where

import Logica
import Haskell4Maths (Vect(..),var,mindices,eval)
```

```
import F2

import Test.QuickCheck
```

La función encargada de hacer dicha traducción es la función `tr`., que equivale al mapeo P descrito 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 contraria (de polinomios a fórmulas) se usará la función $\Theta : \mathbb{F}_2[x] \to Form(\mathcal{L})$ 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 2.2.2. Sea f una fórmula proposicional cualquiera, $\Theta(P(f))$ es equivalente a f. La implementación de esta propiedad es:

```
-- |
-- >>> quickCheckWith (stdArgs {maxSize = 50}) 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 chequear 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 2.2.3. 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 $\mathbb{F}_2[x]$.

2.2.5. 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 mapeos entre el conjunto de interpretaciones o valoraciones y los pontos de $\mathbb{F}_2^n$, que definen biyecciones entre modelos de F y puntos de la variedad algebraica $\mathcal{V}(1 + P(F))$;

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

Por ejemplo, sea 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 $\mathcal{V}(1 + P(F)) = \mathcal{V}(x_1 + x_1x_2x_3)$.

```
-- |
-- >>> let [p1,p2,p3] = map Atom ["p1","p2","p3"]
-- >>> let 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
```

2.2.6. Proyeccion polinomial

Consideremos ahora la parte derecha de la figura 2.1. Para simplificar la relación entre la semántica de la lógica proposicional

Capítulo 3

Regla de independencia y prueba no clausal de teoremas

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 ([1]).

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

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

3.1. 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 4

Aplicaciones

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 ([1]).

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

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

4.1. 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 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 ([1]).

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. Alonso. Temas de programación funcional. Technical report, Univ. de Sevilla, 2015.

[2] 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.

Índice alfabético