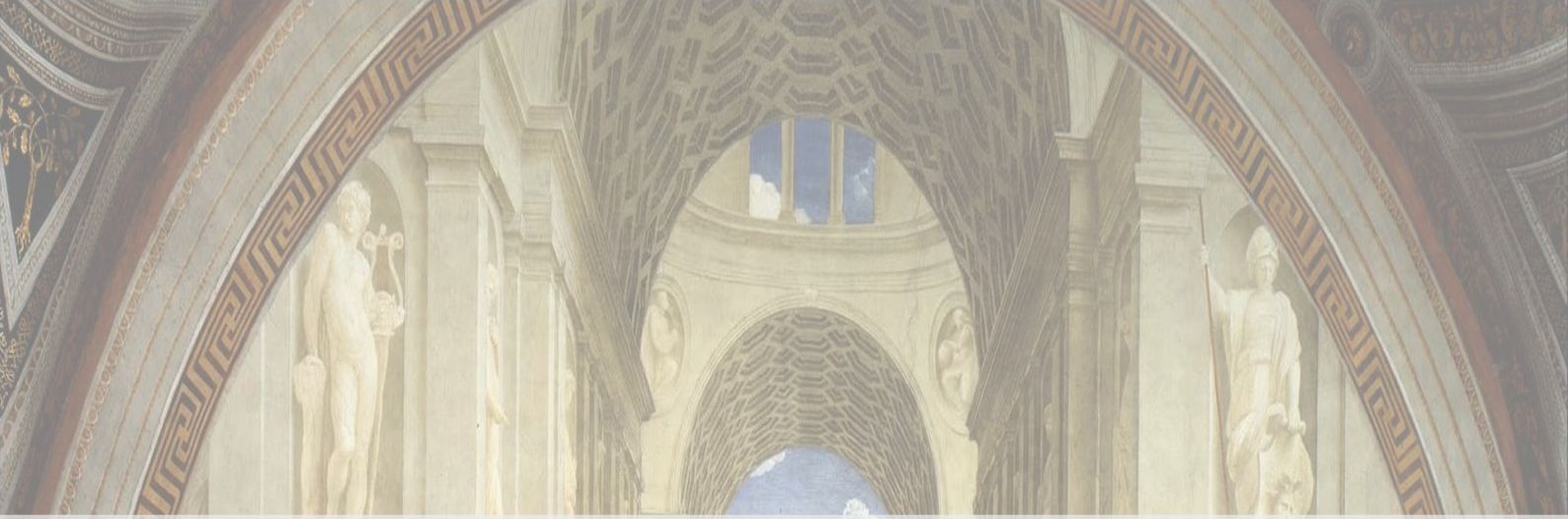

Desarrollo de la Lógica Proposicional y de Primer Orden bajo el paradigma funcional y la orientación Web.

Proyecto de Alumnía Interna 2019/20

Autor:
Ramos González, Víctor

Tutor:
Sancho Caparrini, Fernando
Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE SEVILLA

Desarrollo de la Lógica Proposicional y de Primer Orden bajo el paradigma funcional y la orientación Web.

Víctor Ramos González

Tutor: Fernando Sancho Caparrini
Ciencias de la Computación e Inteligencia Artificial
E.T.S. Ingeniería Informática
Universidad de Sevilla

Curso 2019/20

Resumen

El proyecto aborda los conceptos y algoritmos básicos de la Lógica Proposicional y la Lógica de primer, desde un punto de vista implementativo a través de un lenguaje encuadrado en el paradigma funcional (Elm).

El proyecto, basado en la asignatura de Lógica Informática, busca una doble finalidad, por un lado servir como una somera introducción a la programación declarativa al mismo que tiempo que proporcionar al alumnado herramientas intuitivas y de sencillo uso en la realización de los ejercicios que apoyen los contenidos teóricos que se desarrollan en dicha asignatura.

Índice general

Capítulo 1

Introducción. Objetivos y organización del proyecto

1.1. Introducción

El proyecto surge desde 2 inspiraciones: por un lado mi interés para con el desarrollo de la Teoría de la Lógica Matemática y por otro mi gusto y vocación en el ámbito docente, por lo que la idea de poder desarrollar una herramienta, enfocada al ámbito académico, para el trabajo con diversas lógicas, me despertó una gran motivación.

El presente proyecto, pretende llevar a cabo el desarrollo de una herramienta que permita dar un punto de vista más práctico de los contenidos de la Lógica Proposicional y la Lógica de Primer Orden, al mismo tiempo que sirva al alumno como complemento para la comprensión de los conceptos y la realización y razonamiento de los ejercicios relacionados con dichos conceptos.

El desarrollo del proyecto se llevará a cabo bajo el paradigma de la programación funcional (con el lenguaje Elm) y la posibilidad que este ofrece para su sencilla implementación web para permitir la elaboración de una herramienta interactiva, sencilla y accesible.

1.2. Antecedentes del proyecto

Tras haber cursado la asignatura de Lógica Informática con el profesor D. Fernando Sancho Caparrini, director de este proyecto, se despertó mi gusto por la Teoría de la Lógica Matemática y Computacional y tras comprobar que son excasas las herramientas prácticas que abordan estos contenidos desde un punto de vista académica se planteó la realización de un proyecto análogo con el uso de otras herramientas y otros lenguajes, aunque finalmente, el proyecto no pudo llevarse a cabo.

No obstante la idea del desarrollo de esta herramienta seguía en mente y tras cursar la asignatura de *Programación Declarativa* con el profesor D. Miguel Ángel Martínez del Amor y el lenguaje Haskell, planteé a Fernando la posibilidad de retomar el proyecto pero bajo el lenguaje *Haskell* y el paradigma funcional. Sin embargo, una herramienta similar había sido desarrollada por el profesor D. José Antonio Alonso Jiménez en su obra *Lógica en Haskell*, en la que realiza un amplio desarrollo de la Lógica Proposicional y completada por D. Eduardo Paluzo en su TFG, en el que aborda con una metodología análoga al anterior los conceptos de la Lógica de Primer Orden.

Fue entonces cuando Fernando me propuso realizar una 'traducción' (incorporando algunos aspectos complementarios) de dichas obras, dentro del mismo paradigma pero bajo otro lenguaje, orientado además al ámbito Web, el lenguaje *Elm*.

1.3. Objetivos del proyecto

Como hemos comentado anteriormente, el proyecto persigue un objetivo dual:

- Por un lado, desarrollar una librería completa (compuesta de una serie de módulos) en el lenguaje *Elm*, que nos permita llevar a cabo la definición de conjuntos de fórmulas proposicionales y de primer orden, y la aplicación de los algoritmos básicos para tratar de abordar la satisfactibilidad de los mismos, mediante la aplicación de algoritmos fundamentales.
- Por otro lado, llevar a cabo una implementación Web, de manera que el sistema sea usable por los alumnos de manera sencilla y visual, para que sirva como complemento de los conceptos y técnicas abordados en la asignatura de *Lógica Informática.*

Además podemos destacar una serie de objetivos u competencias complementarias que aporta la realización de este proyecto como es la introducción al manejo de herramientas de publicación y manejo de versiones (*git*) o el manejo de *Latex*.

1.4. Estructura del proyecto

El proyecto se estructará en 2 partes fundamentales:

1.4.1. Módulos funcionales

En primer lugar, se llevará a cabo desarrollo de una librería completa que nos permita trabajar, tanto con fórmulas del ámbito de la Lógica Proposicional como de la Lógica de Primer Orden mediante la implementación de distintos módulos funcionales, que se describen detalladamente a lo largo de este documento y que recogen las estructuras, algoritmos y funciones necesarias para abordar los conceptos y técnicas vistas en la asignatura de *Lógica Informática*.

1.4.2. Interfaz gráfica (GUI)

En segundo lugar, se llevará a cabo el diseño e implementación de una interfaz web, que permita el uso del sistema desde 2 ámbitos distintos, por una parte desde el punto de vista de la definición de fórmulas y la aplicación directa de las funciones y técnicas sobre dichas fórmulas, y en segundo lugar la posibilidad de trabajar con las funciones de una manera más cercana a la *Programación Declarativa*.

Para ello se combinará, además del uso de los módulos funcionales, comentados anteriormente, los lenguajes *Elm*, *Html*, *CSS* y *js* para llevar a cabo la implementación de dicha interfaz.

1.4.3. Propósito y Estructura de este documento

Este documento pretende cumplir un doble objetivo:

- Por una parte se pretene que este documento pueda servir como material didáctico, a lo largo de los capítulos integraremos el desarrollo de los conceptos teóricos con las implementaciones llevadas a cabo, de forma que dichos conceptos se vean reflejados de forma casi directa en los códigos presentados.
- Por otra parte, es objeto de este documento servir como documentación del proyecto y manual de uso de la herramienta. Para facilitar esto, al final de cada uno de los capítulos se recogen los códigos completos de los módulos que intervienen en dicho capítulo, así como varias tablas resumen, en el que se presentan los tipos y funciones disponibles en cada uno de los módulos, junto a una somera descripción de las mismas.

Así, el documento se estructura en distintos capítulos, que servirán de unidades didácticas, y un conjunto de anexos en el que se presentan un conjunto de módulos complementarios, desarrollados para el funcionamiento del sistema.

Capítulo 2

Sintaxis y Semántica

2.1. Descripción general del capítulo

En este capítulo se recogen, de forma detallada, los módulos implementados que abordan el ámbito de la sintaxis y semántica de la Lógica Proposicional y Primer Orden. Complementariamente, en el *Anexo A. Parsers* se encuentra el desarrollo de varios Parsers, que nos permiten acercar la escritura natural de las fórmulas a la definición en el sistema.

2.1.1. Estructura del capítulo

El capítulo se encuentra estructurado en distintas secciones, a través de las cuales se abordan los conceptos fundamentales del ámbito sintáctico-semántico de la LP y la LPO, presentados conjuntamente con los módulos que implementan dichos conceptos:

Módulos

- **Módulo SintaxSemanticsLP**. Recoge las implementaciones de los tipos fundamentales relacionados con los aspectos sintáctico-semánticos de la Lógica Proposicional.
- **Módulo SintaxSemanticsLPO**. Recoge las implementaciones de los tipos fundamentales relacionados con los aspectos sintáctico-semánticos de la Lógica de Primer Orden.

Módulos Complementarios (*A1. Parsers*)

- **Módulo LP_Parser**. Recoge la implementación de un Parser, que permite la escritura de fórmulas LP según la estructura natural, permitiendo el uso de operadores infijos (negación, conjunción, disyunción, implicación, equivalencia).
- **Módulo LPO_Parser**. Recoge la implementación de un Parser, que permite la escritura de fórmulas LPO según la estructura natural, permitiendo, además de los operadores LP (negación, conjunción, disyunción, implicación, equivalencia), el uso de operadores LPO (existencial y universal).
- **Módulo LP_toString**. Recoge algunas funciones destinadas a la presentación de las fórmulas en formato de cadena de texto y en formato *Latex*.

2.2. Conceptos básicos de la Lógica Proposicional

2.2.1. Caracetización básica de la Lógica Proposicional

La Lógica surge como método de modelado del siguiente problema:

> Dado un conjunto de asertos (afirmaciones), $\mathcal{BC}$ (*Base de conocimiento*), y una afirmación, $\mathcal{A}$, decidir si $\mathcal{A}$ ha de ser necesariamente cierta supuestas ciertas las fórmulas de $\mathcal{BC}$.

De manera que para abordar este problema desde el punto de vista lógico-proposicional, resultan necesarios los siguientes elementos:

- Un lenguaje que permita expresar de forma precisa las afirmaciones. (Sintáxis)
- Una definición clara de qué se entiende por *afirmación cierta* (Semántica)
- Mecanismos efectivos (y a poder ser eficientes) que garanticen la corrección (y preferentemente la completitud) en las deducciones. (Algoritmos de decisión)

A lo largo de los distintos capítulos abordaremos estos puntos para las dos representaciones más comunes, la Lógica Proposicional (*LP* o *PL*) y la Lógica de Primer Orden (*LPO* o *FOL*).

Por el momento vamos a comenzar este primer capítulo abordando los dos primeros puntos para la Lógica Proposicional y la Lógica de Primer Orden, dando una somera introducción al tercero para cada caso dejando el desarrollo de los algoritmos de decisión, que se irán abordando a lo largo del resto de capítulos.

Características fundamentales de la LP.

- Sus expresiones (denominadas f*órmulas proposicionales* o *proposiciones*) modelan afirmaciones que pueden considerarse *ciertas* o *falsas*.
- Las fórmulas proposicionales (en adelante fórmulas (si no existe ambigüedad)), se construyen mediante un conjunto de expresiones básicas (*fórmulas atómicas* o *átomos*) y conjunto de operadores (*conectivas lógicas*). Dichas conectivas permiten modelar los siguientes tipos de afirmaciones:
  - *Conjunción*: '... tal ... Y ... cual ...'
  - *Disyunción*: '... tal ... O ... cual ...'
  - *Implicación*: 'SI tal ... ENTONCES ... cual ...'
  - *Equivalencia*: '... tal ... SI Y SÓLO SI ... cual ...'
  - *Negación*: 'NO es cierto tal ...'

  Profundizaremos en este aspecto en la próxima sección, cuando veamos la *Sintaxis de la LP*.
- El lenguaje sólo permite modelar este tipo de afirmaciones, por lo que muchas veces puede ser difícil (o imposible) representar el problema en este tipo de Lógica, y es necesario recurrir a otras más ricas (*LPO, Lógicas Modales, Lógica Fuzzy, etc*). Especificaremos este apartado cuando tratemos las limitaciones de la LP e introduzcamos la LPO.
- Aunque esta Lógica puede resultar de una aparente sencillez, el problema SAT corresponde a la categoría de problemas NP-completos, esto es, no existe ningún algoritmo capaz de resolver el problema planteado en un tiempo polinomial de ejecución. Trataremos de nuevo este aspecto en la introducción a los algoritmos de decisión.

2.3. Sintáxis y Semántica de LP. Módulo SintaxSemanticsLP

En este primera sección vamos a abordar desde un punto de vista teórico-práctico, los elementos base que conforman la Lógica Proposicional, esto es la Síntaxis y la Semántica, mostrando unificadamente los desarrollos formales como las implementaciones llevadas a cabo para modelar cada uno de ellos.

2.3.1. Aspectos Sintácticos

El alfabeto proposicional

El concepto '*alfabeto proposicional*' referencia al conjunto de símbolos que forman parte de este lenguaje. Podemos distinguir las siguientes categorías:

- **Variables proposicionales o átomos**. Ya hemos señalado previamente que todo problema está representado por relaciones entre un conjunto finito de afirmaciones básicas, dichas afirmaciones se representan por símbolos proposicionales: $VP = \{p_0, p_1, \ldots, p, q, r\}$.

  En el lenguaje LIUS, corresponden a cadenas de caracteres:

```
type alias PSymb = String
```

Listing 2.1: Definición de Símbolo Proposicional como alias de String.

Aunque esta definición admite cualquier cadena como símbolo proposicional, vamos a adoptar algunas reglas de notación, que presentaremos a lo largo del capítulo y que vamos a implementar en el parser que nos permitirá pasar de cadenas que representan fórmulas en un lenguaje cercano al lenguaje formal de la lógica a fórmulas proposicionales reconocidas por el sistema. Adoptamos el siguiente criterio en relación a los símbolos proposicionales:

'*Los símbolos proposicionales deben comenzar por una letra minúscula, seguida (opcionalmente) de otros caractéres en minúscula o dígitos numéricos, exclusivamente.*'

[Ver la implementación del módulo *LP_Parser* para ver la implementación de dicho criterio. (*Ver A1. LP_Parser*)]

- **Conectivas Lógicas**. Modelan las relaciones entre las distintas afirmaciones básicas (si es que las hay). Podemos distinguir:
  - De aridad 1 o monoaria : *Negación* ($\neg$)
  - De aridad 2 o binarias: *Conjunción* ($\wedge$), *Disyunción* ($\vee$), *Condicional* ($\rightarrow$), *Bicondicional* ($\leftrightarrow$).

  En el lenguaje LIUS, corresponden a constructores, los veremos seguidamente cuando abordemos el concepto de fórmula.
- **Símbolos Auxiliares**: '*(*' y '*)*'. Permiten expresar relaciones de prioridad entre conectivas lógicas y evitar la ambigiüedad en la interpretación de las fórmulas.

  En el lenguaje LIUS (en la definición fundamental) no son necesarios, ya que la prioridad viene dada por el orden de uso de los constructores, pero sí serán necesarios en la definición del Parser, en el que se escribirán del mismo modo y cumplirán la función que se ha definido para los mismos.

Fórmula Proposicional

Una fórmula proposicional corresponde a una fórmula atómica (un símbolo proposicional) o a conjunto de símbolos proposicionales (con cardinalidad mayor o igual que 2), relacionados entre sí por alguna de las conectivas lógicas. Formalmente, El conjunto de las fórmulas proposicionales, $PROP$, es el menor conjunto de expresiones que verifica:

- $VP \subseteq PROP$
- Es cerrado bajo las conectivas lógicas, esto es:
  - Si una fórmula $F \in PROP$, entonces $\neg F \in PROP$
  - Si las fórmulas $F, G \in PROP$, entonces $(F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G) \in PROP$

De manera que se tiene una definición recursiva del concepto de *Fórmula*, tal que el caso base corresponde a una fórmula básica (*átomo*) y el caso recursivo corresponde a la aplicación de una conectiva sobre una o dos fórmulas (según la aridad de la conectiva).

En el lenguaje LIUS, hemos definido el concepto de Fórmula Proposicional, como un nuevo tipo, dando una definición recursiva análoga a la presentada anteriormente, mediante los constructores *Atom* (Átomo), *Conj* (Conjunción), *Disj* (Disyunción), *Impl* (Implicación o Condicional), *Equi* (Equivalencia o Bicondicional) y un tipo más que *Insat* que corresponde a la fórmula insatisfactible (la veremos cuando abordemos la *Semántica de LP*).

```
type Prop = Atom PSymb
          | Neg Prop
          | Conj Prop Prop
          | Disj Prop Prop
          | Impl Prop Prop
          | Equi Prop Prop
```

Listing 2.2: Definición del tipo Prop (Fórmula Proposicional).

Prioridad de las conectivas y Reducción de Paréntesis

1. Omitimos los paréntesis externos
2. Tomaremos como orden de precedencia de las conectivas (de mayor a menor): $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. Para la conectiva $\leftrightarrow$ se recomienda mantener los paréntesis en todos los casos.
3. Cuando una conectiva se usa repetidamente, se asocia por la derecha.

Si definimos las fórmulas a partir de los constructores, estas reglas no son necesarias, ya que no puede existir ambigüedad alguna en la definición fórmula, pero sí son relevantes en el caso de la declaración a partir del Parser.

[Ver la implementación del módulo *LP_Parser* para ver la implementación de dicho criterio. (*Ver A1. LP_Parser*)]

Esto nos permite definir todas las fórmulas proposicionales, presentamos la construcción de las siguientes 3 fórmulas a modo de ejemplo:

(a) $(p \wedge q) \vee (p \wedge r)$ (b) $(p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q$ (c) $(p \leftrightarrow q) \wedge (p \rightarrow \neg q) \wedge p$

```
a = Disj (Conj (Atom "p") (Atom "q")) (Conj (Atom "p") (Atom "r"))

b = Impl
        (Disj
            (Conj (Atom "p") (Atom "r"))
            (Conj (Neg (Atom "p")) (Atom "q"))
        )
        (Neg (Atom "q"))

c = Conj
        (Conj
            (Equi (Atom "p") (Atom "q"))
            (Impl (Atom "p") (Neg (Atom "q")))
        (Atom "p")
```

Listing 2.3: Ejemplos de definición de fórmulas proposicionales.

Como se puede apreciar, escribir las fórmulas de esta forma puede resultar una tarea ardua y propensa a errores, por eso, se ha desarrollado un parser que nos permite escribir de forma más cómoda, sintética y visual las fórmulas.

A partir de este momento todas las fórmulas de los ejemplos se definirán utilizando el Parser por lo que llegados a este punto se recomienda realizar un estudio del mismo, para tener claras las reglas sintácticas a seguir, aunque, son análogas a las planteadas por el lenguaje formal de la lógica proposicional. (*Ver Anexo A1. LP_Parser*).

Árboles de formación

Los árboles de formación corresponden a grafos de tipo árbol que muestran el desarrollo de formación de las fórmulas (siguiendo la definición recursiva de las mismas) por lo que el árbol de formación asociado a una fórmula es esencialmente único.

De hecho, en el lenguaje LIUS, podemos ver una correspondencia directa entre el árbol de formación y la definición de las fórmulas mediante el uso de los constructores. Si tomamos el ejemplo *(b)*, se puede apreciar claramente como el código de definición y el árbol de formación son claramente análogos:

```
b = Impl
        (Disj
            (Conj
                (Atom "p")
                (Atom "r"))
            (Conj
                (Neg
                        (Atom "p"))
                (Atom "q")))
        (Neg
                (Atom "q"))
```

Listing 2.4: Definición de $(p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q$.

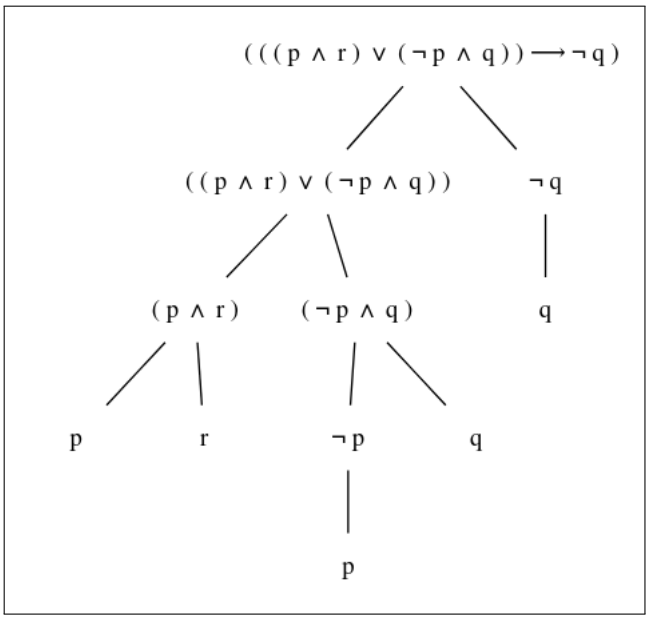

Figura 2.1: Árbol de formación de la fórmula $(p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q$

Además LIUS posee algunas funciones que nos permiten generar estos árboles de formación para poder visualizarlos.

Utilizando la función *formTree* definida en el módulo que estamos tratando, podemos generar un objeto *Graph* que corresponde al árbol de formación. Para poder visualizarlo, podemos pasar dicho objeto *Graph* a una *String* que representa el Grafo en formato *DOT*, con la función *formTree2DOT*.

PROVISIONAL

Para visualizar el árbol de formación podemos utilizar una herramienta externa como https://dreampuf.github.io/GraphvizOnline/, insertando el código DOT obtenido de *formTree2DOT*

A continuación se muestra el código de implementación de las funciones anteriores. La complejidad del código reside más en aspectos técnicos relacionados con el lenguaje *Elm* que en el propio algoritmo, por lo que no es necesario entender el código, es suficiente con entender el método y conocer la existencia de las funciones.

```
formTree : Prop -> Graph String ()
formTree x =
    case x of
        Atom psymb -> fromNodesAndEdges [Node 0 psymb] []
        Neg p ->
            let (nodes, edges) = formTreeAux p 1 in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::nodes) (Edge 0 1 ()::edges)
        Conj p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)

        Disj p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Impl p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Equi p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Insat -> fromNodesAndEdges [Node 0 (toStringProp x)] []


formTreeAux : Prop -> NodeId -> (List (Node String), List (Edge ()))
formTreeAux x nodeid=
    case x of
        Atom psymb -> ([Node nodeid psymb], [])
        Neg p ->
            let
                nextid = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
            in
                let
                    (nodes, edges) = formTreeAux p nextid
                in
                    (Node nodeid (toStringProp x)::nodes,
                                        Edge nodeid nextid ()::edges)
```

```
        Conj p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                           edges1 ++ edges2)
        Disj p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                           edges1 ++ edges2)
        Impl p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                           edges1 ++ edges2)
```

```
        Equi p q ->
           let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                              edges1 ++ edges2)
        Insat -> ([Node nodeid (toStringProp x)], [])

formTree2DOT : Graph String () -> String
formTree2DOT ft =
    let myStyles =
            { defaultStyles |
                node = "shape=plaintext, color=black"
                , edge = "dir=none"}
    in
        outputWithStyles myStyles (\x -> Just x) (\_ -> Nothing) ft
```

Listing 2.5: Definición de las funciones *formTree* y *formTree2DOT*

Principio de inducción sobre fórmulas

Gracias a la definición de $PROP$ (y su estructura recursiva), para probar que toda fórmula proposicional satisface una cierta propiedad (Ψ), podemos hacerlo aplicando el método de inducción sobre fórmulas.

De esta forma, probamos:

1. (*Caso base*). Probar que todos los elementos de VP tienen la propiedad Ψ.
2. (*Paso de inducción*).
   a) Si $F \in PROP$ tiene la propiedad Ψ, entonces $\neg F$ tiene la propiedad Ψ.
   b) Si $F, G \in PROP$, tienen la propiedad Ψ entonces $(F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G) \in PROP$ tienen la propiedad Ψ

Conjuntos de fórmulas

Definido $PROP$, los conjuntos de fórmulas no son más que subconjuntos de $PROP$, estoes, corresponden a agrupaciones de fórmulas proposicionales.

En el lengaje LIUS, los conjuntos proposicionales se definen como listas de fórmulas proposicionales. De esta forma:

```
type alias PropSet = List Prop
```

Listing 2.6: Definición de Conjunto de Fórmulas como Lista de Fórmulas Proposicionales

De forma que la definición de estos se realiza como listas de objetos *Prop*. Como ejemplo, el conjunto $\{(p \wedge q) \vee (p \wedge r), (p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q, (p \leftrightarrow q) \wedge (p \rightarrow \neg q) \wedge p\}$

```
a : Prop
a = Disj (Conj (Atom "p") (Atom "q")) (Conj (Atom "p") (Atom "r"))

b : Prop
b = Impl
        (Disj
            (Conj (Atom "p") (Atom "r"))
            (Conj (Neg (Atom "p")) (Atom "q"))
        )
        (Neg (Atom "q"))

c : Prop
c = Conj
        (Conj
            (Equi (Atom "p") (Atom "q"))
            (Impl (Atom "p") (Neg (Atom "q")))
        (Atom "p")

M: PropSet
M = [a, b, c]
```

Listing 2.7: Ejemplo de definición de conjuntos proposicionales.

Para facilitar su escritura, en el módulo *LP_Parser*, se incluye la función *parserPropSet*, que nos permite definirlos como lo hacemos con el Parser de funciones proposicionales, separando éstas por '*;*'. Recomendamos vea la definición, explicación y ejemplos en el (*Anexo A1. LP_Parser*).

2.3.2. Aspectos Semánticos

Interpretaciones y Valor de verdad de fórmulas proposicionales.

Una vez provista la sintaxis, pasamos a desarrollar la semántica de la Lógica Proposicional. Como ya comentamos, hemos de abordar la interpretación de certeza o veracidad de las fórmulas. Para esto es necesario conocer los conceptos de *valor de verdad* y *función de verdad*.

- **Valor de verdad**. Los elementos del conjunto $\{0, 1\}$ se denominan valores de verdad o valores booleanos. Representan si un hecho es cierto o no, de forma que el valor 1 se asocia a *verdadero* y el valor 0 a *falso*.
- **Funciones de verdad**. Corresponden a funciones que devuelven un valor de verdad según el valor de verdad de los argumentos. Así, el significado (valor de verdad asociado) de cada una de las conectivas lógicas viene dado por una función de verdad, de forma que:
  - $H_{\neg}(i) = \begin{cases} 1 & \text{si } i = 0 \\ 0 & \text{si } i = 1 \end{cases}$
  - $H_{\wedge}(i,j) = \begin{cases} 1 & \text{si } i = j = 1 \\ 0 & \text{e.o.c} \end{cases}$
  - $H_{\vee}(i,j) = \begin{cases} 0 & \text{si } i = j = 0 \\ 1 & \text{e.o.c} \end{cases}$
  - $H_{\rightarrow}(i,j) = \begin{cases} 0 & \text{si } i = 1, j = 0 \\ 1 & \text{e.o.c} \end{cases}$
  - $H_{\leftrightarrow}(i,j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{e.o.c} \end{cases}$

Visto esto, pasamos a estudiar el valor de verdad de las fórmulas proposicionales. Para ello debemos definir el valor de verdad de las variables proposicionales, (denominadas *valoraciones* o *interpretaciones*) y a partir de estas y las funciones de verdad de las conectivas,podemos extender cada valoración,v,de forma única,al conjunto de todas las fórmulas de manera que para cada fórmula F se verifique:

- $v(\neg F) = H_{\neg}(v(F))$
- $v((F \wedge G)) = H_{\wedge}(v(F), v(G))$
- $v((F \vee G)) = H_{\vee}(v(F), v(G))$
- $v((F \rightarrow G)) = H_{\rightarrow}(v(F), v(G))$
- $v((F \leftrightarrow G)) = H_{\leftrightarrow}(v(F), v(G))$

Se dice que $v(F)$ es el valor de verdad de F respecto de la valoración v.

De esta forma, es sencillo realizar el cálculo del valor de verdad de una fórmula respecto de una valoración, recurriendo al árbol de formación de la fórmula, evaluando las subfórmulas, desde las hojas (variables proposicionales) hasta el nodo raíz (la fórmula completa). Por ejemplo el cálculo de la valoración de $F \equiv \neg(\neg(p \vee q) \vee (\neg r \vee s))$ respecto de $v \equiv \{p = 1, q = 1, r = 0, s = 0\}$:

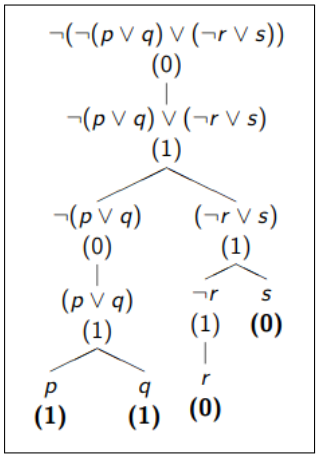

Figura 2.2: Ejmplo del cálculo de la valoración a través del árbol de formación

Para poder modelar la semántica de la LP, en el lenguaje LIUS, se han definido las estructuras necesarias que nos permitan representar las interpretaciones (o valoraciones). Para ello, hemos elegido una representación "dispersa"de manera que una interpretación corresponde a una lista de símbolos proposicionales (variables proposicionales) que son los que son considerados verdaderos, los términos que no aparecen en la lista serán considerados como falsos.

```
type alias Interpretation = List PSymb
```

Listing 2.8: Definición de Interpretación como Lista de Símbolos Proposicionales

De esta forma podemos definir, de forma sencilla el valor de verdad de una fórmula respecto de una valoración. Como hemos señalado antes, el árbol de formación es equivalente a la definición recursiva que hemos adoptado para el modelado de las fórmulas proposicionales, de forma que podemos realizar el cálculo de igual forma que la presentada con el árbol de formación, de forma que:

- (*Caso base*) Una fórmula atómica será verdadera si y sólo si el símbolo proposicional pertenece a la lista de interpretación.
- (Casos recursivos) Según la clase de fórmula:
  - La negación de una fórmula será verdadera respecto de una interpretación si y sólo si la evaluación de la fórmula es falsa.
  - La conjunción de dos fórmulas proposicionales será verdadera respecto de una interpretación si y sólo si la evaluación de ambas fórmulas respecto de dicha evaluación es verdadera.
  - La disyunción de dos fórmulas proposicionales será verdadera será verdadera respecto de una interpretación si y sólo si alguna de las evaluaciones de las dos fórmulas es evaluada verdadera respecto de dicha interpretación.
  - La implicación será verdadera respecto de una interpretación si y sólo si o la evaluación del antecedente es evaluado falso respecto de dicha interpretación o el consecuente es evaluado verdadero respecto de la misma.

- La equivalencia será verdadera respecto de una interpretación si y sólo si la evaluación del antecedente coincide con la evaluación del consecuente.

```
valuation : Prop -> Interpretation -> Bool
valuation pr i =
    case pr of
        Atom p -> List.member p i
        Neg p -> not (valuation p i)
        Conj p q -> valuation p i &&  valuation q i
        Disj p q ->   valuation p i ||  valuation q i
        Impl p q ->   not (valuation p i) ||  valuation q i
        Equi p q ->   valuation (Impl p q) i &&  valuation (Impl q p) i
```

Listing 2.9: Función de valoración de las fórmulas proposicionales

Interpretaciones. Modelos. Satisfactibilidad y Validez Lógica

Existen métodos alternativos (pero equivalentes) para el cálculo del valor de verdad de una fórmula respecto de una valoración. Dado que, dada una valoración,v, el valor de verdad de una fórmula F respecto de v está determinado por los valores de verdad de las subfórmulas de F, podemos construir una tabla que recorra los valores de sus subfórmulas:

p	q	r	$p \to q$	$(p \to q) \vee r$	$\neg((p \to q) \vee r)$
0	0	1	1	1	0

Figura 2.3: Ejmplo del cálculo de la valoración a través de tabla

Una tabla de verdad corresponde a una estructura similar a la anterior (nosotros sólo reflejaremos el valor de las variables proposicionales y el valor de verdad de la fórmula completa), en la que en cada fila se presenta la valoración y el valor de verdad de la fórmula respecto a la misma, para toda interpretación posible (que corresponda a las variables proposicionales presentes en la fórmula).

En el lenguaje LIUS, para poder construir la tabla de verdad, primero hemos de calcular todas las interpretaciones posibles, que, dada la definición dispersa que hemos adoptado, correspondería a todos los posibles subconjuntos (*powerset*) que podríamos construir con los símbolos proposicionales que aparecen en la fórmula. Así:

```
symbInProp : Prop -> Set PSymb

symbInProp f=
    case f of
        Atom p -> Set.singleton p
        Neg p -> symbInProp p
        Conj p q -> Set.union (symbInProp p ) (symbInProp q)
        Disj p q -> Set.union (symbInProp p ) (symbInProp q)
        Impl p q -> Set.union (symbInProp p ) (symbInProp q)
        Equi p q -> Set.union (symbInProp p ) (symbInProp q)
```

Listing 2.10: Función para extraer los símbolos proposicionales que intervienen en una fórmula

```
allInterpretations : Prop -> List Interpretation
allInterpretations x =  Aux.powerset <| List.sort <| Set.toList
                                                   <| symbInProp x
```

Listing 2.11: Función para extraer las posibles interpretaciones para una fórmula propsicional

De esta forma podemos expresar la tabla de verdad como una lista de tuplas en las que el primer elemento corresponde a la interpretación y el segundo corresponde a la evaluación de la fórmula respecto de dicha valoración:

```
truthTable : Prop -> List (Interpretation, Bool)
truthTable x = List.map (\xs ->  (xs,valuation x xs)) <| allInterpretations x
```

Listing 2.12: Función para la construcción de la tabla de verdad de una fórmula

Modelos. Validez y Satisfactibilidad

Una vez estudiado lo anterior, vamos dar unas cuantas definiciones:

- **Modelo.** Se dice que una fórmula F es válida en una valoración v o equivalentemente que v es **modelo** de F si $v(F) = 1$ y se denota por $v \models F$. En caso contrario, se dice que v es **contramodelo** de F y se denota $v \not\models F$.
- **Satisfactibilidad.** Una fórmula F se dice **satisfactible** (o consistente) si existe una valoración v que es modelo de F. En caso contrario se dice que F es **insatisfactible** (o inconsistente), y se representa por $\bot$.
- **Validez lógica o Tautología**. Una fórmula F se dice **tautología** (o (lógicamente) válida) si toda valoración es modelo de F y se denota $\models F$.

Relación entre Validez y Satisfactibilidad

LEMA: Para cada $F \in PROP$ se verifica:

1. Si F es tautología, entonces F es satisfactible.
2. F es tautología si y sólo si F es insatisfactible.

Como se ha expuesto, los modelos corresponden a las interpretaciones que son evaluadas verdaderas, esto es, de las posibles interpretaciones aquellas hacen la fórmula verdadera. Aquellas interpretaciones que hacen la fórmula falas se denominan contramodelos. Así:

```
models : Prop -> List Interpretation
models x = List.filter (\y -> valuation x y) (allInterpretations x)

countermodels : Prop -> List Interpretation
countermodels x = List.filter (\y -> not (valuation x y))
                                          (allInterpretations x)
```

Listing 2.13: Función para el cálculo de los modelos de una fórmula proposicional

Definidos los modelos, podemos así mismo definir (funcionalmente) los conceptos de satisfactibilidad y validez, de acuerdo a las definiciones expuestas:

```
satisfactibility : Prop -> Bool
satisfactibility x = List.any (\xs-> valuation x xs) (allInterpretations x)

validity : Prop -> Bool
validity x = List.all (\xs-> valuation x xs) (allInterpretations x)

insatisfactibility : Prop -> Bool
insatisfactibility x = not (satisfactibility x)
```

Listing 2.14: Funciones de Satisfactibilidad, Validez e Insatisfactibilidad

Conjuntos de Fórmulas. Modelos y Consistencia.

De forma análoga a la presentada para las fórmulas proposicionales podemos definir los conceptos anteriores, aplicados a conjuntos de fórmulas de forma que:

- **Modelo.** Se dice que una valoración v es **modelo** de un conjunto de fórmulas U si para toda fórmula $F \in U$ se tiene que $v(F) = 1$ y se denota por $v \models U$. En caso contrario, se dice **contramodelo**.
- **Consistencia** Un conjunto de fórmulas U se dice **consistente** si existe una valoración v que es modelo de U. En caso contrario se dice que U es **inconsistente**.

Funcionalmente la definición de los conceptos anteriores es análoga a la de las fórmulas. De forma que:

```
setSymbols : List Prop -> Set PSymb
setSymbols xs =
    List.foldr (\x acc -> Set.union acc (symbInProp x)) Set.empty xs

allSetInterpretations : List Prop -> List Interpretation
allSetInterpretations xs = Aux.powerset <| Set.toList <| setSymbols xs

isSetModel : List Prop -> Interpretation -> Bool
isSetModel xs i =  List.all (\x -> valuation x i) xs

allSetModels : List Prop -> List Interpretation
allSetModels xs = List.filter (isSetModel xs) (allSetInterpretations xs)

allSetCounterModels : List Prop -> List Interpretation
allSetCounterModels xs =
    List.filter (\x -> not(isSetModel xs x)) <| allSetInterpretations xs
```

Listing 2.15: Modelos y contramodelos en conjuntos de fórmulas proposicionales

De forma que ahora, es sencillo, comprobar la consistencia de un conjunto a partir de la definición:

```
isConsistent : List Prop -> Bool
isConsistent xs =
    List.any (\x -> isSetModel xs x) <| allSetInterpretations xs

isInconsistent: List Prop -> Bool
isInconsistent xs = not(isConsistent xs)
```

Listing 2.16: Consistencia e Inconsistencia en Conjuntos Proposicionales

Consecuencia Lógica.

Por último nos queda definir el concepto de **consecuencia lógica**. Una fórmula F es consecuencia lógica (o se sigue) de un conjunto de fórmulas U, y se denota por $U \models F$, si toda valoración que es modelo de U es también modelo de F.

Es precisamente este concepto el que permite formular el problema básico en el marco de la lógica proposicional, que planteamos como objetivo de la LP.

Relación entre consecuencia lógica, consistencia y validez

PROPOSICIÓN: Sea $\{F_1, F_2, \ldots, F_n\} \in PROP$ y $F \in PROP$ son equivalentes:

1. $\{F_1, F_2, \ldots, F_n\} \models F$
2. $(F_1 \wedge F_2 \wedge \ldots \wedge F_n \rightarrow F) \in TAUT$
3. $\{F_1, F_2, \ldots, F_n, \neg F\} \equiv \perp$

Funcionalmente, podemos plantear dos desarrollos alternativos para el concepto de Consecuencia Lógica:

1. Acudiendo a la propia definición.
2. Acudiendo al tercer punto de la proposición anterior.

```
isConsecuence : List Prop -> Prop -> Bool
isConsecuence xs x = List.all (\y -> valuation x y) <| allSetModels xs

isConsecuence : List Prop -> Prop -> Bool
isConsecuence xs x = isInconsistent (xs ++ [Neg x])
```

Listing 2.17: Consecuencia Lógica

2.4. Algoritmos de decisión en LP

Como hemos señalado vamos a presentar de forma somera los algoritmos de decisión en LP, e iremos desarrollando algunos de los algoritmos más importantes a lo largo de los distintos capítulos.

Dado un conjunto de fórmulas U, un **algoritmo de decisión** para U es aquél que dada una fórmula $A \in PROP$, devuelve `SI` cuando $A \in U$ y `NO` cuando $A \notin U$

Esto da pie a la definición de algunos problemas con un especial interés:

- $SAT = \{A \in PROP : A \textit{ es satisfactible}\}$.
- $TAUT = \{A \in PROP : A \textit{ es tautología}\}$.
- Fijado $U \subseteq PROP$, la **Teoría de** U corresponde a: $\mathcal{T}(U) = \{A \in PROP : U \models A\}$.

Precisamente, un algoritmo de decisión para $\mathcal{T}(U)$ proporciona una respuesta al Problema Básico que planteamos al comienzo del capítulo. Por tanto, podemos reducir dicho problema a uno nuevo: Obtener un algoritmo que, dado un conjunto finito de fórmulas proposicionales, U, y una fórmula, F, decida si $U \models F$.

Y este a su vez se reduce a comprobar la satisfactibilidad de una cierta fórmula (o bien la validez de otra), hemos aquí el problema conocido como *Problema SAT*.

Notemos que ya hemos visto un algoritmo, el de las Tablas de Verdad, que resuelve el problema, pero la complejidad de dicho algoritmo es exponencial en el número de símbolos proposicionales, lo que lo hace inabordable para fórmulas de cierta complejidad , incluso computacionalmente.

Hemos de señalar que existen otros algoritmos de decisión del problema SAT, algunos de los cuales abordaremos a lo largo de los siguientes capítulos, pero aún no se ha encontrado ninguno capaz de resolver el problema eficientemente (complejidad polinomial), y, de hecho, se duda de la existencia del mismo. De hecho, determinar la satisfactibilidad de una fórmula proposicional se trata de un problema NP-completo.

2.5. Origen y Conceptos Básicos de Lógica de Primer Orden

2.5.1. Limitaciones de la Lógica Proposicional

Aunque a lógica proposicional posee un semántica sencilla y existen algoritmos de decisión (poco eficientes) para sus problemas básicos, como SAT o la consecuencia lógica, la expresividad de LP es bastante limitada, esto hace que muchos problemas no sean modelables en LP, bien porque requieren un gran número de fórmulas o fórmulas de gran tamaño, o bien porque no puedan ni siquiera expresarse en este lenguaje.

El siguiente ejemplo presenta un razonamiento que es válido, sin embargo no es expresable en LP:

1. Todo hombre es mortal. 2. Sócrates es hombre. 3. Por tanto, Sócrates es mortal.

Es aquí precisamente donde comienza el ámbito de la Lógica de Primer Orden.

2.5.2. Caracterización de la Lógica de Primer Orden

La **Lógica de Primer Orden** o **Lógica de Predicados** es un sistema formal diseñado para estudiar los métodos inferenciales en los leguajes de primer orden. Un **lenguaje de primer orden** corresponde a un lenguaje formal que consta de:

- Símbolos lógicos (comunes a todos los lenguajes): En los que se engloban:
  - Un conjunto de *Variables:* $V = \{x, x_0, x_1, \ldots, y, y_0, \ldots\}$
  - *Conectivas lógicas :* $\neg$ (negación), $\wedge$ (conjunción), $\vee$ (disyunción), $\rightarrow$ (implicación), $\leftrightarrow$ (equivalencia).
  - *Cuantificadores:* $\exists$ (existencial), $\forall$ (universal).
  - *Símbolos auxiliares:* '$($' y '$)$'
- Símbolos no lógicos (propios de cada lenguaje): En los que se engloban:
  - Un conjunto de *Constantes:* $L_C = \{a, b, \ldots, a_0, a_1, \ldots\}$
  - Un conjunto de *símbolos de función:* $L_F = \{f_0, f_1, \ldots\}$, cada uno con su aridad correspondiente.
  - Un conjunto de **símbolos de predicado:** $L_P = \{P_0, P_1, \ldots, Q, Q_0, \ldots\}$, cada uno con su aridad correspondiente.
    Dos notas:
    - Los símpolos de predicado de aridad 0 actúan como símbolos proposicionales.
    - El símbolo de igualdad ('=') no es un predicado común a todos los lenguajes de primer orden, pero si es corriente su aparición. La familia de lenguajes que incluyen este predicado es denominada Lenguajes de Primer Orden con igualdad.

2.6. Sintáxis y Semántica en LPO Módulo SintaxSemanticLPO

TO DO

2.7. Código y funciones de los módulos presentados

2.7.1. SintaxSemanticsLP

Código del módulo

```
module Modules.SintaxSemanticsLP exposing (
    PSymb, Prop(..), Interpretation,PropSet,
    valuation, truthTable, models, countermodels, satisfactibility, validity,
    insatisfactibility, isSetModel, allSetModels, allSetCounterModels,
    isConsistent, isInconsistent, isConsecuence, setSymbols, formTree,
    formTree2DOT)

import List
import Set
import Modules.AuxiliarFunctions as Aux
```

```
import Graph exposing (Graph(..), Node, Edge, NodeId, fromNodesAndEdges)
import Graph.DOT exposing (outputWithStyles, defaultStyles)
import Maybe exposing (Maybe(..))

-----------
-- TYPES --
-----------
type alias PSymb = String

type Prop = Atom PSymb
          | Neg Prop
          | Conj Prop Prop
          | Disj Prop Prop
          | Impl Prop Prop
          | Equi Prop Prop
          | Insat

type alias Interpretation = List PSymb
type alias PropSet = List Prop

-------------
-- METHODS --
-------------

formTree : Prop -> Graph String ()
formTree x =
    case x of
        Atom psymb -> fromNodesAndEdges [Node 0 psymb] []
        Neg p ->
            let (nodes, edges) = formTreeAux p 1 in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::nodes) (Edge 0 1 ()::edges)
        Conj p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)

        Disj p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Impl p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Equi p q ->
            let
                (nodes1, edges1) = formTreeAux p 1
                (nodes2, edges2) = formTreeAux q 2
```

```
            in
                fromNodesAndEdges
                    (Node 0 (toStringProp x)::(nodes1 ++ nodes2))
                    ([Edge 0 1 (), Edge 0 2 ()] ++ edges1 ++ edges2)
        Insat -> fromNodesAndEdges [Node 0 (toStringProp x)] []


formTreeAux : Prop -> NodeId -> (List (Node String), List (Edge ()))
formTreeAux x nodeid=
    case x of
        Atom psymb -> ([Node nodeid psymb], [])
        Neg p ->
            let
                nextid = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
            in
                let
                    (nodes, edges) = formTreeAux p nextid
                in
                    (Node nodeid (toStringProp x)::nodes,
                                        Edge nodeid nextid ()::edges)

        Conj p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                           edges1 ++ edges2)
        Disj p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                           edges1 ++ edges2)
        Impl p q ->
            let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
```

```
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                              edges1 ++ edges2)

        Equi p q ->
           let
                nextid1 =  Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "1"
                nextid2 = Maybe.withDefault 0
                                <| String.toInt
                                    <| String.fromInt nodeid ++ "2"
            in
                let
                    (nodes1, edges1) = formTreeAux p nextid1
                    (nodes2, edges2) = formTreeAux q nextid2
                in
                   ( Node nodeid (toStringProp x)::(nodes1 ++ nodes2),
                       [Edge nodeid nextid1 (), Edge nodeid nextid2 ()] ++
                                                              edges1 ++ edges2)
        Insat -> ([Node nodeid (toStringProp x)], [])

formTree2DOT : Graph String () -> String
formTree2DOT ft =
    let myStyles =
            { defaultStyles |
                node = "shape=plaintext, color=black"
                , edge = "dir=none"}
    in
        outputWithStyles myStyles (\x -> Just x) (\_ -> Nothing) ft


valuation : Prop -> Interpretation -> Bool
valuation pr i =
    case pr of
        Atom p -> List.member p i
        Neg p -> not (valuation p i)
        Conj p q -> valuation p i &&  valuation q i
        Disj p q ->   valuation p i ||  valuation q i
        Impl p q ->   not (valuation p i) ||  valuation q i
        Equi p q ->   valuation (Impl p q) i &&  valuation (Impl q p) i
        Insat -> Basics.False

symbInProp : Prop -> Set.Set PSymb

symbInProp f=
    case f of
        Atom p -> Set.singleton p
        Neg p -> symbInProp p
        Conj p q -> Set.union (symbInProp p ) (symbInProp q)
        Disj p q -> Set.union (symbInProp p ) (symbInProp q)
        Impl p q -> Set.union (symbInProp p ) (symbInProp q)
        Equi p q -> Set.union (symbInProp p ) (symbInProp q)
        Insat -> Set.empty
```

```
allInterpretations : Prop -> List Interpretation
allInterpretations x =  Aux.powerset <| List.sort <| Set.toList
                                                    <| symbInProp x

truthTable : Prop -> List (Interpretation, Bool)
truthTable x = List.map (\xs ->  (xs,valuation x xs)) <| allInterpretations x

models : Prop -> List Interpretation
models x = List.filter (\y -> valuation x y) (allInterpretations x)

countermodels : Prop -> List Interpretation
countermodels x = List.filter (\y -> not(valuation x y))
                              (allInterpretations x)

satisfactibility : Prop -> Bool
satisfactibility x = List.any (\xs-> valuation x xs) (allInterpretations x)

validity : Prop -> Bool
validity x = models x== allInterpretations x

insatisfactibility : Prop -> Bool
insatisfactibility x = List.isEmpty (models x)

setSymbols : List Prop -> Set.Set PSymb
setSymbols xs = List.foldr (\x acc -> Set.union acc (symbInProp x))
                           Set.empty
                           xs

allSetInterpretations : List Prop -> List Interpretation
allSetInterpretations xs = Aux.powerset <| Set.toList <| setSymbols xs

isSetModel : List Prop -> Interpretation -> Bool
isSetModel xs i =  List.all (\x -> valuation x i) xs

allSetModels : List Prop -> List Interpretation
allSetModels xs = List.filter (isSetModel xs) (allSetInterpretations xs)

allSetCounterModels : List Prop -> List Interpretation
allSetCounterModels xs = List.filter (\x -> not(isSetModel xs x))
                                     (allSetInterpretations xs)

isConsistent : List Prop -> Bool
isConsistent xs = List.any (\x -> isSetModel xs x)
                           (allSetInterpretations xs)

isInconsistent: List Prop -> Bool
isInconsistent xs = not(isConsistent xs)

isConsecuence : List Prop -> Prop -> Bool
--isConsecuence xs x = List.all (\y -> valuation x y) <| allSetModels xs
isConsecuence xs x = isInconsistent (xs ++ [Neg x])
```

Listing 2.18: Módulo SintaxSemanticsLP

Funciones disponibles

Tipo	Descripción
PSymb	Alias de *String*. Representa los símbolos proposicionales.
Prop	Representa a las proposiciones o fórmulas proposicionales. Posee varios constructores según el tipo de fórmula proposicional: (*Atom, Neg, Conj, Disj, Impl, Equi*).
Interpretation	Alias de *List PSymb*. Representa una interpretación, de forma que se consideran verdaderos los símbolos proposicionales que aparecen en la lista, y falsos aquellos que no aparecen.
PropSet	Alias de *List Prop*. Representa conjuntos de fórmulas.

Tabla 2.1: Módulo SintaxSemanticsLP I. Tipos

Método	Descripción
formTree	*formTree: Prop ->Graph String ()* Genera un grafo con el árbol de formación de la fórmula dada.
formTree2DOT	*formTree2DOT: Graph String () ->String* Genera el una cadena con el código en formato DOT del árbol de formación recibido.
valuation	*valuation: Prop ->Interpretation ->Prop* Calcula el valor de verdad de una proposición según la interpretación dada.
truthTable	*truthTable: Prop ->List (Interpretation, Bool)* Calcula la tabla de verdad asociada a una fórmula proposicional, devolviéndola como una lista de pares (Interpretación, Valoración).
models	*models: Prop ->List Interpretation* Calcula los modelos de una fórmula proposicional.
countermodels	*countermodels: Prop ->List Interpretation* Calcula los contramodelos de una fórmula proposicional.
satisfactibility	*satisfactibility: Prop ->Bool* Decide si una fórmula proposicional es satisfactible o no.
validity	*validity: Prop ->Bool* Decide si una fórmula proposicional es tautología o no.
insatisfactibility	*insatisfactibility: Prop ->Bool* Decide si una fórmula proposicional es insatisfactible o no.
isSetModel	*isSetModel: List Prop ->Interpretation ->Bool* Decide si una interpretación es modelo de un conjunto de fórmulas proposicionales o no.
allSetModels	*allSetModels: List Prop ->List Interpretation* Calcula los modelos asociados a un conjunto de fórmulas proposicionales.
allSetCounterModels	*allSetCounterModels: List Prop ->List Interpretation* Calcula los contramodelos asociados a un conjunto de fórmulas proposicionales.
isConsistent	*isConsistent: List Prop ->Bool* Decide si un conjunto de fórmulas proposicionales es consistente o no.
isInconsistent	*isInconsistent: List Prop ->Bool* Decide si un conjunto de fórmulas proposicionales es inconsistente o no.
isConsecuence	*isConsecuence: List Prop ->Prop ->Bool* Decide si una fórmula es consecuencia lógica de un conjunto de fórmulas proposicionales.

Tabla 2.2: Módulo SintaxSemanticsLP II. Funciones

2.7.2. SintaxSemanticsLPO

TO DO

2.8. Ejercicios Propuestos

TO DO

2.9. Resolución de los ejercicios propuestos utilizando LIUS

TO DO

Anexos

Anexo A

Parsers

A.1. Módulo LP_Parser

TO DO

A.2. Módulo LPO_Parser

Este módulo pretende facilitar la tarea de definición de las fórmulas, acercando a la forma natural de escritura de las fórmulas de primer orden. Para ello vamos a basarnos en las reglas de sintaxis expuestas en el Módulo SintaxSemanticsLPO. Vamos a presentarlas junto a algunos casos que ejemplifiquen su escritura:

Reglas sintácticas

$\longrightarrow$ *Términos*

- Las **variables** se escriben con una letra en minúscula seguidas de caracteres alphanuméricos o el símbolo '_' (*ejemplos:* 'x', 'x1', 'x_1', 'x_ab'). *NOTA:* Son palabras reservadas las cadenas 'forall', 'exists'.

```
parseVar : Parser Term
parseVar = succeed Var
            |= variable
                { start = Char.isLower
                , inner = \c -> Char.isAlphaNum c || c == '_'
                , reserved = Set.fromList ["exists", "forall"]
                }
```

Listing A.1: ParserLPO. Definición de Variables.

- Las **funciones** se definen siguiendo la notacion prefija, según el siguiente patrón:

  [símbolo de función] [parámetros]

  donde:

  - El *sómbolo de función* debe comenzar por el carácter '_', seguido de una serie de caracteres, entre estos se admiten caracteres alfanuméricos y también símbolos (exceptuando '[', ']', '(', ')', ';').
  - En los *parámetros* se pueden dar 2 casos, o bien la función es una **constante** (no tiene parámetros), con lo cual bastaría escribir el nombre (*ejemplos:* '_a', '_PEDRO', '_1') o bien es una **función** (dependiente de, al menos, unos argumentos), en tal caso los argumentos se dan en una lista acotada por corchetes ('[', ']') y tras cada argumento ha de ir un ';' (*ejemplos:* '_f[x;]', '_g[x;y;]', '_+[x;_1;_·[y;_2;]]').

```
parseTerm: Parser Term
parseTerm =
    oneOf [
        succeed Func
            |. symbol "_"
            |= variable
                { start =  \c -> c /= '[' && c /= ']' && c /= ';' && c/= '('
                    && c/= ')' && not (isSpace c)
                , inner = \c -> c /= '[' && c /= ']' && c /= ';' && c/= '('
                    && c/= ')' && not (isSpace c)
                , reserved = Set.fromList []
                }
            |= parseParams
        , parseVar
    ]
```

```
parseParams : Parser (List Term)
parseParams =
    oneOf [
        succeed identity
            |. symbol "["
            |= loop [] listTerm
            |. symbol "]"
        , succeed []
        ]


listTerm : List Term -> Parser (Step (List Term) (List Term))
listTerm revTerms =
  oneOf
    [ succeed (\term-> Loop (term :: revTerms))
        |= parseTerm
        |. spaces
        |. symbol ";"
        |. spaces
    , succeed ()
        |> map (\_ -> Done (List.reverse revTerms))
    ]
```

Listing A.2: ParserLPO. Definición de Términos.

$\longrightarrow$ *Fórmulas*

Aunque recordemos que las fórmulas podíamos dividirlas en **atómicas** y **compuestas**, vamos a hacer algunas dentro de estas.

- Las fórmulas atómicas están constituidas por los **predicados**. Para todos ellos se debe usar la notación prefija, según el patrón:

  [símbolo de predicado] [parámetros]

  donde:

  - El *símbolo de predicado* debe comenzar por un carácter en mayúscula o un símbolo (exceptuando '_', '!', '(', ')', '[', ']', '¬'. También se admiten dígitos numéricos pero no se recomienda), seguido de una serie de caracteres, entre estos se admiten caracteres alfanuméricos y también símbolos (exceptuando '[', ']', '(', ')').
  - En los *parámetros* se pueden dar 2 casos, o bien la función es un **predicado proposicional** (no tiene parámetros), con lo cual bastaría escribir el símbolo de predicado (*ejemplos:* 'P', 'Q_1') o bien es un **predicado n-ario** (dependiente de, al menos, un argumento), en tal caso los argumentos se dan en una lista acotada por corchetes ('[', ']') y tras cada argumento ha de ir un ';' (*ejemplos:* 'P[x;]', '>=[x;y;]', 'MAX[x;_1;_·[y;_2;]]').

```
succeed Pred
        |= variable
            { start = \c -> not (Char.isLower c || c == '_' || c == '!'
                || c == '(' || c == ')' || c == '[' || c == ']'
                || c=='¬'|| i sSpace c)
            , inner = \c -> Char.isAlphaNum c || not (c == '('
                || c == ')' || c == '[' || c == ']' || isSpace c)
            , reserved = Set.fromList []
            }
        |= parseParams
        |.spaces
```

Listing A.3: ParserLPO. Definición de Predicados.

Hay una excepción, el **predicado de igualdad (binaria)** se utilizará de forma infija con el símbolo '='.)(*ejemplos:* 'x=y' , 'x_4=_3', '_f[x;]=_f[y;]'), siguiendo el patrón:

[término] = [término]

```
, succeed Equal
    |= parseTerm
    |. spaces
    |. symbol "="
    |. spaces
    |= parseTerm
    |. spaces
```

Listing A.4: ParserLPO. Definición de Predicado de Igualdad.

- Las fórmulas compuestán formadas por fórmulas y **conectivas** (teniendo en cuenta las prioridades de las mismas '¬', '∧', '∨', '→', '↔' (con esta última preferentemente los paréntesis), y **cuantificadores** ('∃', '∀'). Estos últimos tienen prioridad sobre las conectivas (se recomienda mantener los paréntesis para evitar posibles fallos en la interpretación de la notación). Con el uso de cuantificadores, la notación sigue el patrón:

  [símbolo de cuantificador] {[variable]} [fórmula]

  donde:

  - El *símbolo de cuantificador* corresponde o bien al existencial (∃), representado por la cadena 'exists' o al universal (∀), representado por la cadena 'forall'.
  - La *variable* corresponde a una variable, siguiendo las reglas expuestas anteriormente. (Nótese que ha de estar escrita entre llaves).
  - La *fórmula* corresponde a una fórmula escrita según las reglas expuestas.

```
succeed Exists
    |.symbol "exists"
    |.spaces
    |.symbol "{"
    |.spaces
    |= parseVar
    |.spaces
    |.symbol "}"
    |.spaces
    |= lazy(\_ -> lpoParser)
    |.spaces

  , succeed Forall
    |.symbol "forall"
    |.spaces
    |.symbol "{"
    |.spaces
    |= parseVar
    |.spaces
    |.symbol "}"
    |.spaces
    |= lazy(\_ -> lpoParser)
    |.spaces
```

Listing A.5: ParserLPO. Definición de Fórmulas con cuantificadores.

De forma que ya podemos escribir todas las fórmulas posibles en Lenguajes de Primer Orden (puedes consultar el código del módulo y un resumen de los requisitos sintácticos al final de esta sección).

Ejemplos de definición de fórmulas LPO con LPO_Parser.

- Prop. Antisimétrica: $\forall x \forall y(R(x,y) \land R(y,x) \rightarrow x = y$

```
antisimetricProp : FormulaLPO
antisimetricProp =
    Maybe.withDefault Insat <| Tuple.first <| parserFormula
        <| "forall{x} forall{y} (R[x;y;] & R[y;x;] -> x = y)"
```

Listing A.6: ParserLPO. Definición de la de la Prop. Antisimétrica.

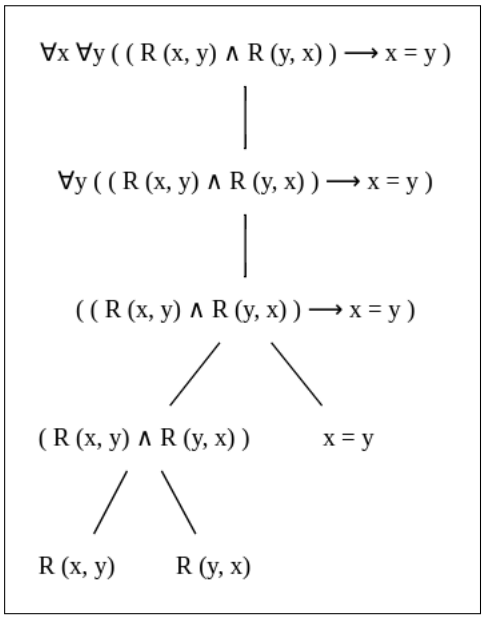

Figura A.1: Árbol de formación de la Prop. Antisimétrica

- Prop Biyectiva. Para expresarla consideremos el siguiente lenguaje: $L = \{O, I, f\}$, tal que:
  - O es un predicado de aridad 1 que expresa si el objeto pertenece al conjunto origen.
  - I es un predicado de aridad 1 que expresa si el objeto pertenece al conjunto imagen.
  - f es una función que dado un elemento del conjunto origen obtiene el correspondiente elemento del conjunto imagen y dado un elemento del conjunto imagen obtiene el correspondiente del conjunto origen.

  La propiedad biyectiva se formula como: *'Una relación es biyectiva si para todo elemento del conjunto origen existe un único elemento del conjunto imagen con el que está relacionado y todo elemento del conjunto imagen está relacionado con algún elemento del origen'*. Esto es:

$$\forall x, y \in O\,(x \neq y \rightarrow f(x) \neq f(y)) \land \forall x \in I\,\exists y \in O\,(f(x) = y)$$

```
antisimetricProp : FormulaLPO
antisimetricProp =
    Maybe.withDefault Insat <| Tuple.first <| parserFormula
        <| "forall{x}forall{y}(O[x;] & O[y;] & ¬(x=y) -> ¬(_f[x;]=_f[y;]))
            & forall{x}(I[x;]->exists{y}(O[y;]&_f[y;]=x))"
```

Listing A.7: ParserLPO. Definición de la de la Rel. Biyectiva.

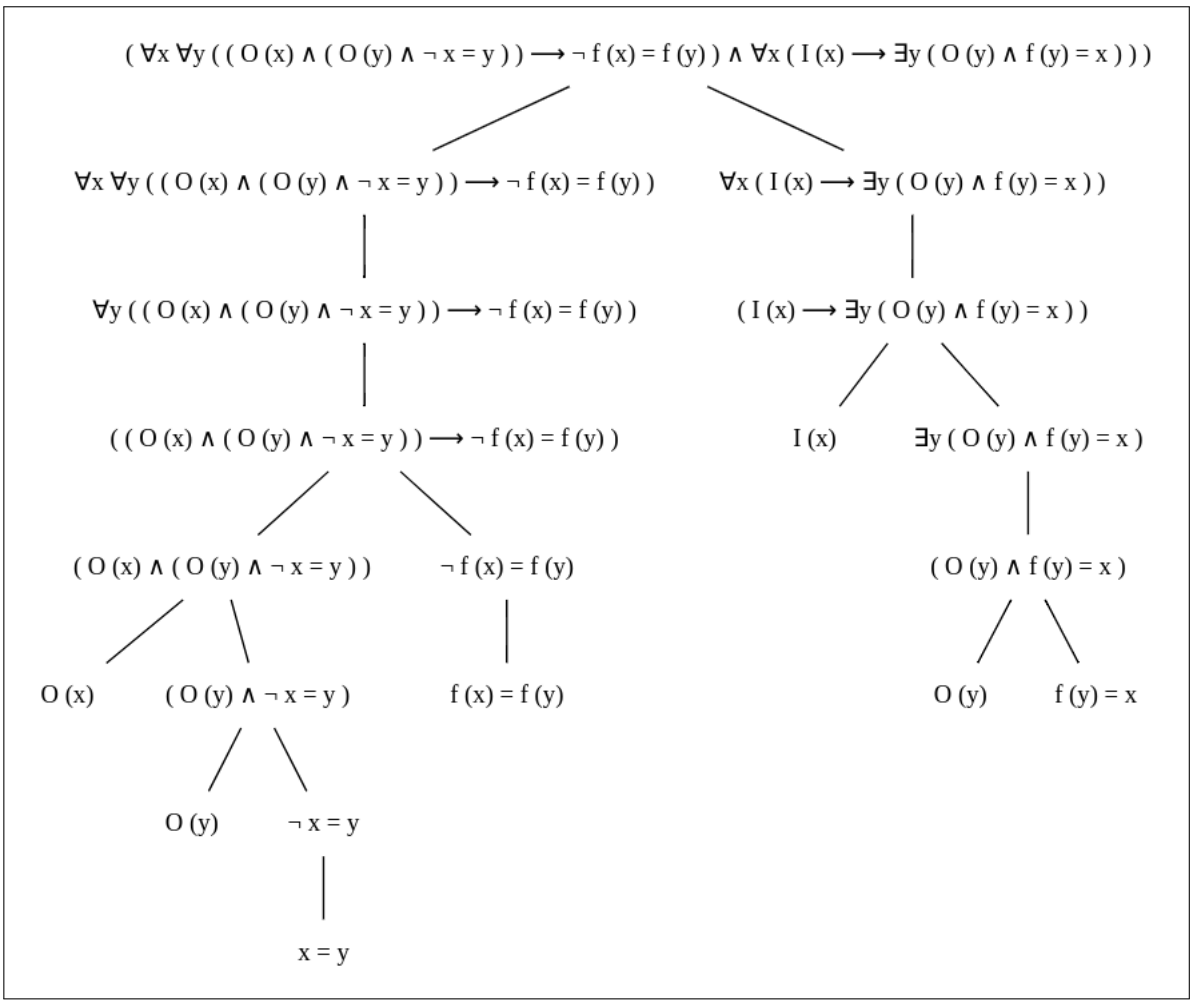

Figura A.2: Árbol de formación de la Rel. Biyectiva.