LogicUS: Tratmiento Funcional con Orientación Web de las Lógicas Proposicional y de Primer Orden

Autor: Ramos González, Víctor

TRABAJO FIN DE GRADO - G.I.I. Tecnologías Informáticas

Dirigido por el Prof. D. Fernando Sancho Caparrini
Dpto. Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE SEVILLA

LogicUS

Agradecimientos

En primer lugar me gustaría agradecer de manera especial su colaboración al profesor D. Fernando Sancho Caparrini. No sólo por las labores de dirección, la resolución de dudas y la formación que me ha permitido adquirir, sino por apoyar, desde que lo conocí mi desarrollo a lo largo de mi trayectoria académica e investigadora.

En segundo lugar, me gustaría agradecer a mi familia y amigos, de manera especial a mis padres y mi hermano, por los sacrificios y el apoyo que me han brindado, animándome siempre a llevar a cabo mis proyectos personales y profesionales, así como a aprovechar las oportunidades y conseguir las metas que se han planteado en mi trayectoria académica y personal.

He de agradecer, de forma destacada, también su empeño y dedicación a todo el grupo que me ha acogido para realizar mi Beca de Colaboración permitiéndome también adquirir muchos conocimientos y experiencias complementarias a la mera formación académica; y en especial a su director Juan Antonio Álvarez y a cada uno de sus miembros: Luis Miguel Soria (LuisMi), José Luis Salazar, Marina Perea, Enrique López, Miguel Ángel Martínez y Pedro Almagro, porque no puedo dejar de agradecer vuestros consejos, vuestro apoyo y vuestra amabilidad en todas aquellas sesiones a las que siempre tuve el placer de asistir, en gran parte por la simpatía con la que todos me habéis acogido.

Finalmente dar mis agradecimientos a las labores docentes a mis profesores del grado, al Dpto de Ciencias de la Computación e Inteligencia Artificial por permitirme la posibilidad de realizar con ellos este trabajo, así como a la E.T.S. de Ingeniería Informática y a la Universidad de Sevilla por ofrecerme la posibilidad de desarrollar en ella mi formación en esta etapa de mi trayectoria académica.

Universidad de Sevilla
Escuela Técnica Superior de Ingeniería Informática.

G.I.I. Tecnologías Informáticas.

TRABAJO FIN DE GRADO

LogicUS: Tratamiento Funcional con Orientación Web de las Lógicas Proposicional y de Primer Orden

Realizado por:
Ramos González, Víctor

Dirigido por:
Fernando Sancho Caparrini

Departamento
Ciencias de la Computación e Inteligencia Artificial

Sevilla, Julio de 2021

Resumen

El proyecto realiza un estudio de las distintas herramientas existentes para el trabajo con la Lógica Proposicional y la Lógica de Primer Orden, de forma que aborden de una forma clara, sencilla y visual muchos de los principales algoritmos para el tratamiento de dichas lógicas.

En estos términos, y ante la imposibilidad de encontrar una que cubra los requisitos expuestos, este proyecto pretende aportar una herramienta para el tratamiento de la Lógica Computacional en un ámbito académico, posibilitando el trabajo con conjuntos de fórmulas de proposicionales y de lenguajes de primer orden (incluyendo lenguajes con igualdad), con el desarrollo de distintos algoritmos bajo el paradigma de la programación declarativa, de manera que proporcione una herramienta capaz de servir como apoyo docente tanto al profesorado como al alumnado, y a estos efectos aglutina en sus distintos componentes y modalidades la capacidad de trabajo desde un enfoque más cercano al paradigma declarativo y la notación funcional al mismo tiempo que, a través del manejo de la interfaz web, proporciona la capacidad de un uso completo y sencillo de las funcionalidades provistas.

El proyecto se centra principalmente en el desarrollo de módulos del lenguaje Elm, que permite además el trabajo multiplataforma, a través del uso de la propia consola (*elm-repl*), o a través de la compilación de los propios módulos al lenguaje web *javascript*, permitiendo la integración con sistemas para la creación de documentos *.html* y *.md* (integración con *gicentre/litvis*).

Índice de contenidos

Contenidos **Página**

1 Introducción

1.1. Marco de desarrollo del proyecto

1.1.1. Inicios del proyecto

El proyecto surge a raíz de una propuesta en el curso 2018/2019 por parte del profesor Fernando Sancho Caparrini, consistente en la realización de una extensión para Visual Studio Code de una adaptación del programa TouIST (que luego trataremos), como parte del programa de Alumnía Interna de tres alumnos (entre los que me incluía). Sin embargo, y por distintas causas, el proyecto no fue satisfactorio, por lo que parecía que todo había quedado en meras intenciones.

Sin embargo, la semilla estaba sembrada y mi motivación hacia el mundo docente haría resurgir la idea un año más tarde. Tras cursar la asignatura de Programación Declarativa con el profesor D. Miguel Ángel Martínez del Amor y ante la situación pandémica vivida en el curso 2019/2020, que obligó a reorientar la docencia hacia la difusión telemática y la digitalización del desarrollo de las sesiones y los contenidos, se realizó una nueva propuesta para la realización del proyecto, esta vez de manera individual, y bajo el paradigma de la programación declarativa y con una acentuada orientación Web.

De forma que, en colaboración con el Dpto. de Ciencias de la Computación e Inteligencia Artificial, y los profesores de la asignatura Lógica Informática (D. Fernando Sancho y D. Felix Lara) se acordó la realización de un proyecto para el desarrollo de un software orientado al tratamiento de la Lógica Proposicional y Lógica de Primer Orden que permitiera complementar los contenidos docentes de la asignatura con contenidos prácticos y la generación de documentación, favoreciendo además la digitalización docente y el desarrollo de la capacidad de aprendizaje de los alumnos.

1.1.2. Justificación del proyecto

Si realizamos una somera visión de la trayectoria y aplicación de la Lógica Informática, entonces es claro que la misma constituye un papel básico en el desarrollo de toda la disciplina informática, interviniendo muy distintas áreas desde aspectos más orientados al hardware y el diseño de circuitos (en los que la lógica de Boole constituye la base) hasta otros más cercanos al desrrollo de software como la programación, el manejo de bases de datos u otros referentes al campo de la computación en ámbitos como la Teoría de la Computabilidad y la Complejidad Computacional y una de las más importantes disciplinas hoy en día, la Inteligencia Artificial.

En particular, las teorías lógicas en IA son independientes de las implementaciones. Se pueden utilizar para proporcionar información sobre el problema de razonamiento sin informar directamente la implementación. Las implementaciones directas de ideas a partir de la lógica (técnicas de demostración de teoremas y construcción de modelos) se utilizan en la IA, pero los teóricos de la IA que se basan en la lógica para modelar sus áreas problemáticas también pueden utilizar otras técnicas de implementación. Así, Robert C. Moore en el primer capítulo de su libro *Lógic and representation*[1] distingue tres usos de la lógica en la IA: como herramienta de análisis, como base para la representación del conocimiento, y como lenguaje de programación.

Esto pone de relevancia la necesidad de una formación sólida de todos los informáticos en el ámbito de la Lógica, al menos a un nivel básico, aún más para aquellos cuya área de aplicación pretenda ser el ámbito de la Computación e IA. Para ello son necesarias, además de las teorías matemático-computacionales desarrolladas sobre la lógica a lo largo de la historia, herramientas software que permitan a los alumnos tratar de forma aplicada (práctica) los contenidos, técnicas y procedimientos que se sumergen en dichas teorías. De hecho, este aspecto es de tal relevancia que se organizan congresos internacionales en los que se presentan nuevas técnicas y herramientas desarrolladas con fines docentes (como el *International Congress on Tools for Teaching Logic*, Salamanca (2004, 2008, 2011), Rennes (2015)).

Centrándonos ahora en las asignaturas de Lógica Informática de distintos grados analizados (a nivel nacional e internacional), los contenidos típicos de las mismas suelen incluir el tratamiento de las dos lógicas más básicas, la Lógica Proposicional y la Lógica de Primer Orden, incluyendo en los contenidos los tres aspectos básicos que conforman cada una de las lógicas: sintáxis, semántica y algoritmos de decisión.

Al realizar una revisión de las distintas herramientas software disponibles para el tratamiento de dichos contenidos se encuentra cierta variedad de las mismas, entre las que destacan:

- *Prover9 y Mace4*: Prover9 es un demostrador automático de teoremas para lógicas de primer orden (también puede ser usado para proposicional), y Mace4 busca modelos finitos y contraejemplos. Sin embargo, no dispone de una gran capacidad gráfica ni explicativa, y carece de mantenimiento desde el 2007.
- *TouIST* (Toulouse Integrated Satisfiability Tool)[1]: Desarrollado por el IRIT (Instituto de Investigación en Ciencias de la Computación de Toulouse), es un programa utilizado para la definición y resolución de problemas de satisfacción de restricciones (CSP) modelados como fórmulas de la lógica proposicional (Popositional Logic (PL)) y lógica de primer orden (First Order Logic (FOL)). Este pequeño lenguaje de programación permite expresar fórmulas proposicionales (paramétricas) de una forma cómoda y natural. Sin embargo, no dispone de una explicación del desarrollo ni de la posibilidad de incluir comentarios, etc.

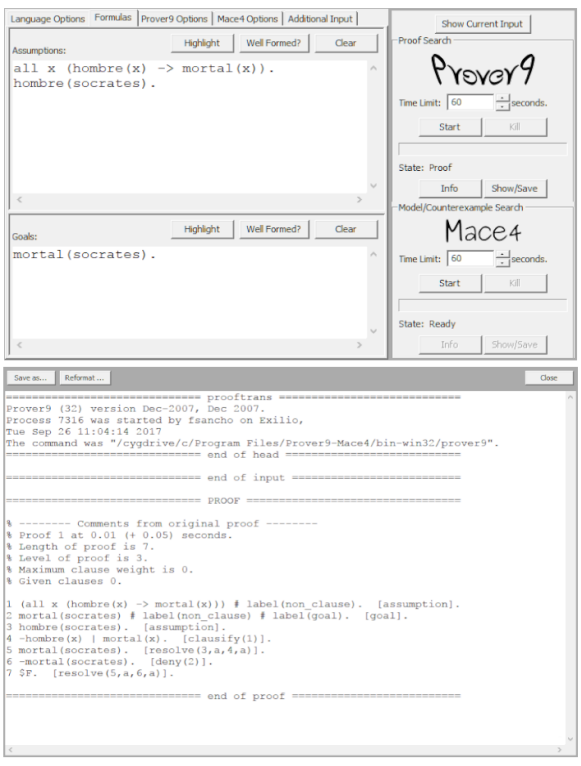


Figura 1.1: Ejemplo de Prover9Mace4
Fuente: https://www.cs.unm.edu/~mccune/prover9/

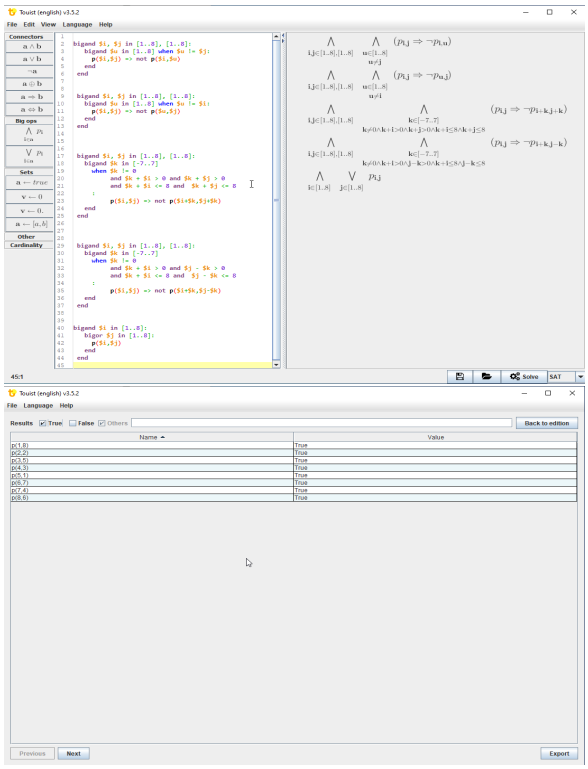


Figura 1.2: Ejemplo de TouIST
Fuente: https://www.irit.fr/TouIST/

- *Gateway to Logic*: Servicio web que permite realizar operaciones varias sobre fórmulas proposicionales (formas normales, tablas de verdad, árboles de formación, etc.).
- *ProofTools*: Una aplicación web para la generación automática y gráfica de Tableros Semánticos. Funciona, entre otras, para la LP, *LPO* y *LPO* con igualdad.

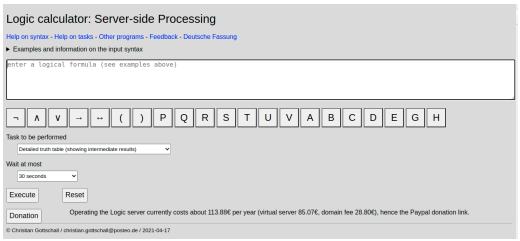


Figura 1.3: Ejemplo de Gateway to Logic
Fuente: https://www.erpelstolz.at/gateway

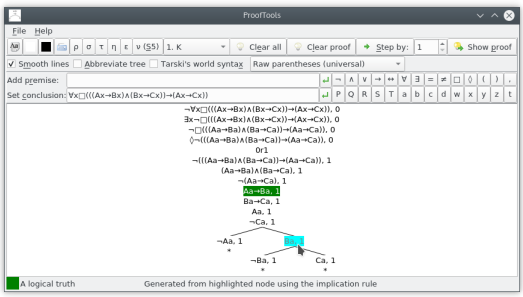


Figura 1.4: Ejemplo de Proof Tools
Fuente: https://creativeandcritical.net/prooftools

- *Tree Proof Generator*: Servicio web que genera Tableros Semánticos en *LP* y *LPO*.
- *LogEx*: Servicio web que convierte a formas normales y prueba equivalencias en *LP*. Se puede usar como herramienta de ejercicios porque permite la verificación de los pasos dados por el usuario, y también muestra la solución paso a paso.

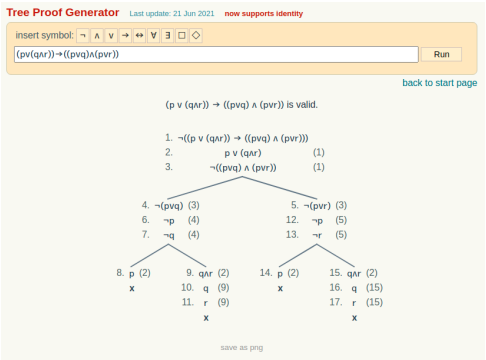


Figura 1.5: Ejemplo de Proof Generator
Fuente: https://www.umsu.de/trees/

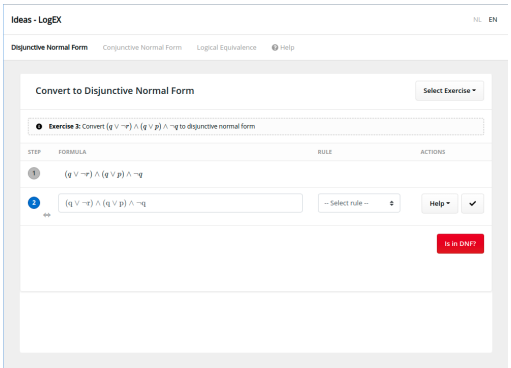


Figura 1.6: Ejemplo de LogEx
Fuente: https://ideas.science.uu.nl/logex/

- *Logica*: Herramienta Web desarrollada por la Universidad de Standford que permite aplicar algunos algoritmos de LP y LPO.

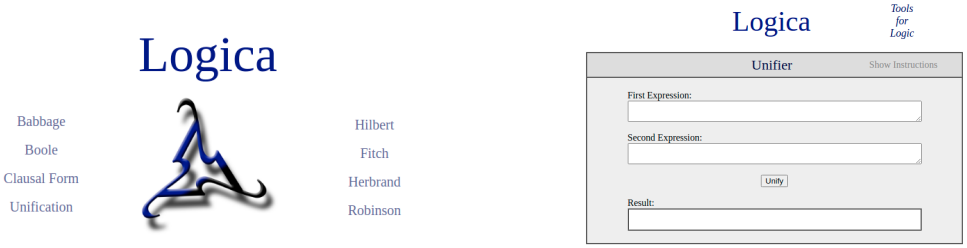


Figura 1.7: Ejemplo de lógica
Fuente: http://logica.stanford.edu/logica/php/index.php

- Otras: *BoolTool*, *SATRennesPA*, *Logictools*, ...

Según se ha podido observar, existe una cantidad considerable de herramientas, pero prácticamente todas cubren propósitos específicos y muchas no son muy explicativas y/o carecen de representaciones gráficas para los algoritmos. Por ello, el principal objetivo de este proyecto es la creación de un sistema que permita utilizar todas las herramientas y algoritmos dedicados a la Lógica Proposicional y a la Lógica de Primer Orden bajo un mismo entorno que incluya la posibilidad de mostrar de forma gráfica los procesos realizados y la generación de documentos con dichos procesos.

1.2. Objetivos y requerimientos del proyecto.

En este punto comentaremos de forma detallada los objetivos específicos del proyecto y los requerimientos del mismo.

Objetivos y alcance del proyecto

Dentro del marco de desarrollo del proyecto se pretenden lograr los siguientes puntos:

- Disponer de una herramienta que permita, en un mismo entorno, trabajar con las fórmulas de LP y LPO.
- Disponer de representaciones gráficas para las fórmulas (utilizando subíndices, superíndices, símbolos lógicos,...), en todos los modos de visualización, sobre todo, en el documental.
- Disponer de desarrollos explicativos para los algoritmos, adoptando representaciones en forma de listas, árboles de representación, etc.
- Disponer de una herramienta capaz de integrar la documentación en forma de texto, imágenes, tablas, etcétera, con las funciones ejecutables de forma que se pueda mostrar conjuntamente y de forma ordenada tanto el texto como los resultados, integrados en un mismo documento (preferiblemente en formato *.md*, *.html* o *.pdf*)

Además de dichos objetivos, se considera también favorable:

- Un diseño modular de la herramienta, no cerrado, de forma que pueda ser fácilmente ampliable con otros contenidos complementarios (nuevos algoritmos y estrategias para los algoritmos existentes, por ejemplo).
- Disponer de documentación online de las distintas funciones mostrando, si procede, algunos ejemplos de uso.
- Permitir el uso de la herramienta en la mayoría de sistemas operativos, con una instalación ligera, fácil y que conste de pocos componentes.

Requerimientos del proyecto

- El sistema debe permitir la definición de fórmulas proposicionales y de primer orden, de forma equivalente a la definición formal de la lógica, representando los elementos de forma análoga a como se representan en el formalismo matemático, definiendo el uso de símbolos lógicos, conectivas lógicas, cuantificadores, predicados y funciones y algunos símbolos auxiliares. Así mismo se debe permitir la definición de conjuntos de fórmulas.
- La sintaxis de las fórmulas ha de ser clara y adecuarse con similitud a la sintáxis formal de la lógica proposicional y de primer orden, con una simbología sencilla y análoga a los correspondientes símbolos matemáticos ($\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \forall, \exists$).
- El sistema debe proveer la semántica de las fórmulas, permitiendo la definición de interpretaciones, la evaluación de las fórmulas respecto a estos, las comprobaciones de satisfactibilidad, validez y consecuencia lógica, así como los algoritmos tratados en la asignatura para abordar el problema SAT:

- Para la Lógica Proposicional (*LP*):
  - *Tableros Semánticos*. Se deberá implementar el algoritmo de tableros semánticos, permitiendo su visualización en forma de árbol, así como la obtención de modelos y/o contramodelos.

  - *Formas Normales y Cláusulas*. Se deben implementar los algoritmos que permitan expresar las fórmulas en las distintas formas normales (*FNN*, *FNC* y *FND*) y especialmente en Forma Clausal (*FC*). Además, se debe poder identificar tautologías e inconsistencias a través de estas formas.

  - *DPLL (Davis-Putnam-Logemann-Loveland)*. Se deberá implementar el algoritmo *DPLL*, permitiendo su visualización en forma de árbol, así como la obtención de modelos y/o contramodelos.

  - *Sistemas Deductivos y Cláusulas de Horn*. El sistema debe permitir el trabajo con los sistemas deductivos proposicionales (al menos en un nivel básico), especialmente deben implementarse los mecanismos de representación y sistemas de razonamiento con *Cláusulas de Horn*.

  - *Resolución Proposicional y Estrategias*. El sistema debe implementar el mecanismo de Resolución Proposicional junto con las principales estrategias básicas de resolución.

  - *Modelado y resolución de CSP*. El sistema debe implementar los mecanismos necesarios para el trabajo con Problemas de Satisfacción de Restricciones en LP, con el uso de fórmulas grandes para la definición de los problemas, y suministrando herramientas para la resolución de los mismos (a nivel académico).

- Para la Lógica de Primer Orden (*LPO*):
  - *Tableros Semánticos*. Se deberá implementar el algoritmo de tableros semánticos, restringido en profundidad (debido al carácter no decidible de la LPO) permitiendo su visualización en forma de árbol.

  - *Formas Normales y Cláusulas*. Se deben implementar los algoritmos que permitan obtener formas *Prenex*, *Skolem*, *FNC*, *FND*, *FC* para fórmulas *LPO*.

  - *Reducción de LPO a LP: Herbrand*. Se deben implementar los mecanismos de reducción (parcial) de *LPO* a *LP* a través de los trabajos de Herbrand (extensión y generación de modelos).

  - *Resolución en Primer Orden*. Se deben aportar los mecanismos de Resolución en Primer Orden (con búsqueda restringida), recogiendo su desarrollo en forma de grafo de búsqueda, utilizando la Unificación de Máxima Genralidad y los métodos heurísticos en la búsqueda.

  - *Lenguajes con Igualdad ($LPO_=$)*. Si la duración del proyecto lo permite, se deberán proveer los mecanismos de trabajo de los puntos anteriores para el trabajo con $LPO_=$.

Requerimientos de los módulos de representación y documentación

El proyecto debe desarrollar los módulos necesarios para la representación gráfica y textual de las fórmulas y algoritmos implementados, proveyendo funciones de representación que sean integrables con otras herramientas tales como *HTML*.

El proyecto debe establecer las herramientas necesarias que permitan generar documentos con contenidos de texto, imágenes, código, e incluso bloques ejecutables y/o que muestren dichos resultados de forma comprensible. Para este fin se podrá hacer uso de otras herramientas y/o plataformas disponibles.

1.3. Estructura del trabajo

El trabajo se estructura en 5 capítulos, además de la introducción, en los que podemos resumir el contenido siguiente:

- *Capítulo 2.* Se abordan los fundamentos teórico-formales de la Lógica Proposicional y la Lógica de Primer Orden, exponiendo las cuestiones sintácticas y semánticas de ambas lógicas así como los algoritmos de decisión y los problemas y/o limitaciones de ambas lógicas.
- *Capítulo 3.* Se exponen las principales tecnologías utilizadas en el desarrollo del proyecto, justificando su uso y destacando las principales características de cada una de ellas.
- *Capítulo 4.* Se muestra de forma detallada la implementación de las principales funciones, justificando las representaciones adoptadas, las estructuras definidas y los procedimientos provistos en cada uno de los módulos de la herramienta. Complementariamente, se muestran algunos ejemplos de uso para cada uno de dichos módulos, utilizando la herramienta desarrollada.
- *Capítulo 5.* Se exponen de forma breve los componentes, instalación y uso de las distintas funcionalidades provistas dentro de la herramienta.
- *Capítulo 6.* Se exponen las conclusiones derivadas del desarrollo del proyecto, la valoración de satisfactibilidad del mismo y las perspectivas de futuro, comentando los principales aspectos a trabajar en el futuro y la capacidad de crecimiento y actualización de la herramienta desarrollada en este trabajo.

1.4. Material complementario

Para el desarrollo y distribución de los módulos incluidos en el trabajo, así como otros aspectos futuros, se dispone del repositorio de GitHub https://github.com/vicramgon/logicus, en el que se incluyen:

- Los módulos que forman parte de la librería LogicUS, de acceso libre y debidamente comentados.
- Distintos *notebooks* con instrucciones de uso del sistema y distintos documentos generados a modo de manual de usuario de cada uno de los módulos.
- Las instrucciones de instalación del sistema para los principales sistemas operativos.
- La interfaz gráfica (web) para el uso de LogicUS (en desarrollo).
- Este documento.

2 | Fundamentos Teórico-Formales

En este capítulo daremos una somera introducción al formalismo de la Lógica Proposicional (LP) y de Primer Orden (LPO) centrándonos principalmente en el nivel sintáctico y semántico de las fórmulas y el problema de decisión asociado a la verificación de la satisfactibilidad de las mismas, aportando una perspectiva de las técnicas clásicas utilizadas para abordar el problema.

Conjuntamente, se presentará la representación adoptada para los distintos elementos que conforman las lógicas, de forma que se pueda ver una relación directa entre la estructura formal de las fórmulas y la traslación directa de esta estructura, la sintaxis y la semántica en la representación adoptada en el sistema.

2.1. Un perspectiva histórica

La lógica es tan antigua como la filosofía misma. Si acudimos a la etimología de la palabra, encontramos que proviene del vocablo griego *'logos'*, que adoptaba el significado de *razón, ley y palabra* y ha sido considerada desde la antigüedad clásica como una de las ramas principales de conocimiento, aunque el concepción de la disciplina haya tenido distintas interpretaciones a lo largo de los años.

Aristóteles ya planteó en algunos de sus escritos (recogidos en la obra *Organon*) la posibilidad de poder demostrar la veracidad del conocimiento, dando lugar al concepto de la *validez universal*. Encontró el fundamento para la demostración en el proceso deductivo, dando lugar a lo que se conoce como *silogismos*, en el que cierto conocimiento es deducible dada la veracidad de algunas premisas, y que se convirtieron en pieza central de su línea de estudio en este ámbito.

Los estudios aristótelicos sirvieron como inspiración a autores posteriores, desde Pedro Abelardo (s. XI-XII) hasta Kant, quien consideraba que Aristóteles había alcanzado la plenitud en el desarrollo de la lógica. Esto, sin embargo, cambió a partir del siglo XIX, en el que la disciplina se aparta de la filosofía y vira hacia la formalidad matemática con autores como George Boole, Augustus De Morgan, Charles Ludwig Dodgeson (Lewis Carrol), Hugh MacColl o Ernst Schröeder, quien advirtió la importancia del desarrollo de algoritmos rápidos para varios problemas del ámbito de la lógica y de la matemática que aún hoy constituyen lineas de investigación abiertas.

Ese acercamiento entre la lógica y las matemáticas constituyó, incluso, un cambio en el proceso de demostración de la validez que es sometida a verificación formal simbólica a través de un lenguaje similar al empleado en los enunciados de las matemáticas.

Aunque ya Leibnitz había planteado muchas de las cuestiones, fueron autores como Gottlob Frege, David Hilbert, Bertrand Russell y Alfred North Whitehead los que asentaron las bases para el desarrollo de lo que se conoce como *Lógica Matemática*.

Aunque el formalismo presentado llevó a la demostración formal de muchos teoremas y a la resolución de una gran cantidad de problemas vigentes hasta ese momento, casi a mediados del siglo XX una serie de resultados presentados por autores como Kurt Göedel, Alonzo Church o Alan Turing truncaron la posibilidad de considerar el formalismo lógico como método universal para la verificación matemática, pero sí continúo su desarrollo como una rama fundamental de la disciplina.

De hecho, esta rama es una de las responsables de muchísimos de los avances tecnológicos que se han producido, encontrando en el campo de la *Computación* un ámbito de aplicación y desarrollo, convirtiéndose en su pilar fundamental e interviniendo en resultados como el desarrollo de circuitos booleanos y puertas lógicas, el estudio de la explosión combinatoria y la teoría de la *NP-Completitud*, la aparición de lenguajes basados en el ámbito de la lógica, algunos aparentemente más alejados de esta disciplina como los lenguajes de consulta en bases de datos (*SQL*) y otros más dedicados como *Prolog*; el estudio de la semántica formal, que permite asegurar que diferentes implementaciones de un lenguaje de programación produzcan los mismos resultados; la validación del diseño y verificación formal; la Inteligencia Artificial junto con el razonamiento automático y los sistemas expertos; etc.

Así, se distinguen multitud de variantes en el estudio de la lógica, distinguiéndose las lógicas clásicas, en la que se centrará este trabajo, lógicas no clásicas, tales como las lógicas plurivariantes, la lógica difusa, la lógica intuicionista, o la lógica cuántica, lógicas modales, entre las que se incluyen la lógica modal, la deóntica o la temporal, la lógica informal y el razonamiento con inconsistencia, o la metalógica.

Como hemos señalado, en el proyecto se tratarán las lógicas clásicas, que tradicionalmente son clasificadas en dos grandes grupos:

- **La lógica proposicional (LP) o lógica de enunciados**.
- **La lógica de predicados o lógica de clases**. Nosotros nos centraremos en la más básica, correspondiente a la **Lógica de Primer Orden (LPO)**.

Aunque existen diferencias fundamentales entre ellas, todas cumplen tres principios básicos que han permitido desarrollar algunos de los métodos de trabajo comunes con estas lógicas, como son los tableros semánticos o las técnicas de resolución, que veremos someramente a lo largo del capítulo. Todas ellas cumplen tres principios básicos:

- *Principio de no contradicción*: Dos afirmaciones contradictorias no pueden ser ciertas al mismo tiempo.
- *Principio del tercero excluso*: En dos afirmaciones contrarias una de ellas ha de ser necesariamente verdadera.
- *Principio de identidad*: La identidad de una afirmación consigo misma es siempre verdadera.

De forma que, según lo visto hasta ahora, podemos observar cómo a través del tiempo han aparecido multitud de aproximaciones cuyo objetivo se centra tanto en la formalización del conocimiento como en el desarrollo de métodos que permitan obtener nuevas verdades de forma deductiva. En este sentido, podemos dar una visión clara del punto en el que poner el foco a través de lo que se puede considerar *El Problema Central de la Deducción*, y que se puede formular como:

Problema Central de la Deducción: *Dado un conjunto de asertos (que conforman el conocimiento conocido en algún ámbito de aplicación) y que denominaremos Base de Conocimiento ($\mathcal{BC}$) y una afirmación ($\mathcal{A}$), el problema reside en decidir si $\mathcal{A}$ ha de ser necesariamente verdadera supuestas ciertas las afirmaciones de $\mathcal{BC}$.*

Para asegurar la validez del resultado, se ha de construir un sistema que permita expresar, de forma simbólica, el contenido de las afirmaciones (*Sintaxis*) y distinguir la veracidad o falsedad de las mismas (*Semántica*); y un conjunto de mecanismos que permitan determinar la veracidad de las deducciones, buscando la efectividad, eficiencia y completitud de los mismos (*Algoritmos de decisión*).

Será, precisamente, en estos tres elementos en los que centraremos nuestra atención a lo largo de esta sección, centrándonos en las lógicas más básicas, más comunes y que sirven de fundamento, en cierto modo, del resto de modalidades.

2.2. El formalismo de la Lógica Proposicional

Vista una breve sinopsis del desarrollo histórico del ámbito de la lógica matemática e informática, y de una brevísima fundamentación de las lógicas con las que vamos a tratar, nos centramos ahora en la lógica proposicional y en los elementos que la dan forma, tratando, como complemento, los problemas y limitaciones asociados a la misma.

La lógica de enunciados o lógica proposicional (LP) es la lógica más simple, por lo que ofrece elementos más rígidos que otras y una menor capacidad expresiva. A pesar de ello, sí ha sido aplicable en una gran cantidad problemas y su 'simplicidad' hace además que pueda ser implementada computacionalmente de forma directa y con relativa sencillez, conservando incluso (a nivel computacional) las estructuras que se utilizan formalmente para la definición de las proposiciones y las operaciones entre ellas.

Antes de pasar a estudiar en detalle los elementos que conforman la lógica proposicional, veamos una panorámica de sus características más importantes:

- Sus expresiones (denominadas fórmulas proposicionales o proposiciones) modelan afirmaciones que pueden considerarse ciertas o falsas. No hay posibles estados intermedios, ni distintos, ni solapables, cada afirmación del lenguaje tiene una, y sólo una, de esas dos opciones (tal y como se puede deducir de los principios comentados anteriormente).
- Dichas proposiciones (en adelante fórmulas, si no existe ambigüedad) se construyen mediante un conjunto de expresiones básicas (fórmulas atómicas o átomos) y un conjunto de operadores (conectivas lógicas). Las conectivas disponibles 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 ...'
- El lenguaje sólo permite modelar este tipo de afirmaciones, por lo que muchas veces puede ser difícil (o imposible) representar algunos problemas en LP, y será necesario recurrir a otras opciones más expresivas.
- Aunque esta Lógica puede resultar de una aparente sencillez, veremos que en ella ya aparecen ciertos retos que (como el problema SAT) caen dentro de la categoría de problemas NP-completos, esto es, problemas complejos para los que no se conocen algoritmos eficientes para resolverlos. Trataremos de nuevo este aspecto cuando introduzcamos formalmente los algoritmos de decisión.

A lo largo de los siguientes apartados trataremos los elementos que forman el sistema, dando los conceptos y herramientas necesarias que permitan comprender los desarrollos de implementación que se han llevado a cabo en el proyecto.

2.2.1. Sintaxis de la Lógica Proposicional

Entre las acepciones del diccionario de la Real Academia de la Lengua de España (RAE) encontramos la siguiente definición para *sintaxis*:

> *'Conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje'*

Esto es justo lo que vamos definir, los elementos que forman el 'lenguaje' de la lógica proposicional y las asociaciones consideradas correctas o coherentes entre ellas y que juntos formarán lo que denominaremos *fórmulas proposicionales*.

Vamos a ver unas cuantas definiciones que nos permitan dar forma a la sintaxis LP:

Definición 2.2.1 (Alfabeto) *Corresponde al conjunto de símbolos que forman parte de un lenguaje y que se pueden combinar para construir las distintas expresiones permitidas.*

En concreto, en el alfabeto proposicional podemos distinguir tres elementos que lo constituyen:

- *Variables proposicionales.* Representan los asertos del ámbito de estudio y se denotan por letras minúsculas acompañadas, opcionalmente, de subíndices. Ejemplo de ellas son $\{p, q, r, s, \ldots, p_0, p_1, \ldots, q_0, q_1, \ldots\}$.
- *Conectivas lógicas.* Modelan las relaciones válidas entre las afirmaciones (variables) y corresponden (según su aridad):
  - De aridad 1: *negación* ($\neg$).
  - De aridad 2: *conjunción*($\wedge$), *disyunción*($\vee$), *implicación*($\rightarrow$), *equivalencia*($\leftrightarrow$).
- *Símbolos auxiliares*: que ayudan a establecer relaciones de prioridad entre conectivas lógicas y evitar ambigüedad en la interpretación de las fórmulas. Se consideran como símbolos auxiliares únicamente los paréntesis ('(' , ')').

Definición 2.2.2 (Expresiones y fórmulas) *Una expresión de un lenguaje corresponde a una sucesión finita (y no vacía) de símbolos del mismo. Las fórmulas, por su parte, corresponderían a las expresiones consideradas como 'bien formadas'.*

Más formalmente, el conjunto de las fórmulas proposicionales, PROP, es el menor conjunto de expresiones construidas usando el alfabeto LP tal que:

- *Todas las variables proposicionales aisladas son expresiones del lenguaje ($VP \subseteq PROP$).*
- *Es cerrado bajo las conectivas lógicas, esto es:*
  - *La negación de una fórmula es también una fórmula ($F \in PROP$ entonces $\neg F \in PROP$).*
  - *La aplicación de una conectiva sobre dos fórmulas es también una fórmula (y se enmarca entre paréntesis) ($F, G \in PROP$ entonces $(F \wedge G), (F \vee G), (F \rightarrow G), (F \leftrightarrow G) \in PROP$).*

Aunque esta definición formal puede resultar algo críptica corresponde en realidad a una definición recursiva, ideal para ser trasladada a lenguajes de carácter declarativo, como es nuestro caso. Veremos más tarde este aspecto de forma más concreta.

De hecho, esta definición recursiva nos permite establecer una estructura de representación en forma de árbol que indica el proceso de formación de la fórmula considerada, yendo desde las más simples (variables) en las hojas, hasta la fórmula completa considerada, en la raíz.

En la *figura 2.1* se muestra el árbol de formación para la fórmula $\neg(\neg(p \vee q) \rightarrow (\neg r \wedge s))$. De abajo a arriba, el árbol nos muestra, paso a paso, cómo obtener la fórmula que hay en su raíz partiendo de las variables proposicionales p, q, r y s que aparecen en sus hojas y las correspondientes conectivas lógicas. De arriba a abajo, el árbol nos muestra cómo descomponer la fórmula en otras más simples, a partir de las cuales ha sido construida mediante las conectivas. Las distintas fórmulas que aparecen en los nodos del árbol de formación de una fórmula F se denominan *subfórmulas* de F.

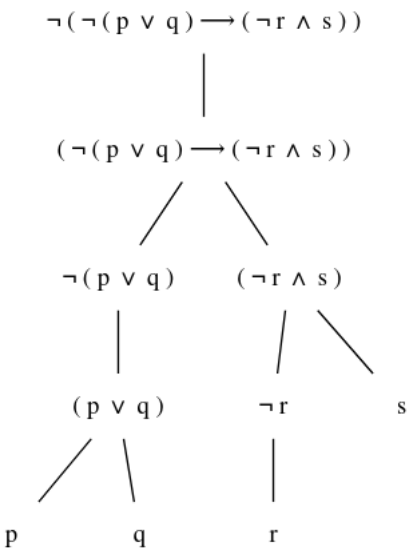


Figura 2.1: Árbol de formación

Finalmente, para facilitar la escritura y lectura de las fórmulas se han establecido una serie de convenios, aunque en realidad las fórmulas deben escribirse según la definición dada. Se admiten tres simplificaciones básicas:

- Se omiten los paréntesis externos de las fórmulas. por ejemplo $F \to G$ correspondería a $(F \to G)$.
- Se asigna a las conectivas un orden de prioridad, de mayor a menor correspondería a: $\neg, \wedge, \vee, \to, \leftrightarrow$. Así $F \wedge G \to H$ correspondería en realidad a $((F \wedge G) \to H)$.
- Cuando una conectiva se usa repetidamente, se asocia por la derecha. Por ejemplo, $F_1 \wedge F_2 \wedge F3$ es una representación alternativa de $(F_1 \wedge (F_2 \wedge F3))$.

2.2.2. Semántica de la Lógica Proposicional

Una vez conocemos un forma válida para escribir las afirmaciones asociadas a un cierto ámbito hemos de definir que se entiende por fórmula verdadera y falsa, esto es, cuál es la *interpretación de una fórmula*. Para poder establecer dichos conceptos formalmente vamos a apoyarnos en un par de definiciones.

Definición 2.2.3 (Valor de verdad) *Corresponden a los elementos del conjunto* $\{0,1\}$ *(valores booleanos) y representan si un hecho (variable) es verdadero (*1*) o falso (*0*).*

Aunque en las lógicas con las que vamos a trabajar se consideran únicamente esos grados de verdad, en otras lógicas se pueden tener más grados de verdad discretos e incluso que el grado de verdad varíe de forma continua en un cierto rango.

Definición 2.2.4 (Función de verdad) *Formalmente es una función n-aria que toma valores de verdad y devuelve un solo valor de verdad, es decir,* $f : \{0,1\}^n \mapsto \{0,1\}$.

Haremos uso de ellas para definir la semántica o interpretación de las conectivas lógicas, concretamente:

$$H_\neg(i) = \begin{cases} 1 & \text{si } i = 0 \\ 0 & \text{si } i = 1 \end{cases}$$

$$H_\wedge(i,j) = \begin{cases} 1 & \text{si } i = j = 1 \\ 0 & e.o.c \end{cases} \qquad H_\vee(i,j) = \begin{cases} 0 & \text{si } i = j = 0 \\ 1 & e.o.c \end{cases}$$

$$H_\to(i,j) = \begin{cases} 0 & \text{si } i = 1,\ j = 0 \\ 1 & e.o.c \end{cases} \qquad H_\leftrightarrow(i,j) = \begin{cases} 1 & \text{si } i = j \\ 0 & e.o.c \end{cases}$$

Los dos elementos definidos anteriormente nos permiten establecer una *interpretación* o *valoración* de los símbolos (variables) de una fórmula indicando para cada uno de ellos el valor de verdad correspondiente. De forma que para la fórmula $\neg(\neg(p \vee q) \to (\neg r \wedge s))$ una posible valoración podría ser $v = \{p = 0, q = 1, r = 0, s = 1\}$.

Una vez se ha dado una valoración, podemos *evaluar* F respecto a una valoración v de forma que $(v(F))$ el valor de verdad de la fórmula respecto de dicha valoración. Dicha función de evaluación para cada fórmula F puede definirse como $ev_F : \{0,1\}^n \in \{0,1\}$, esto es una función que a cada vector n-dimensional le asigna un valor binario que corresponde al valor de verdad de la fórmula respecto a esa valoración. Dicha función puede ser calculada fácilmente sobre el árbol de formación sin más que ir aplicando sucesivamente las funciones de verdad sobre los valores de verdad parciales de las subfórmulas que las componen y las conectivas que las relacionan, llevando a cabo dicha evaluación desde las hojas (variables proposicionales) hasta la raíz (fórmula completa), tal y como se muestra en la *figura 2.2*.

A continuación daremos algunas definiciones que completarán la semántica definida hasta el momento para el trabajo con fórmulas y conjuntos de fórmulas (de los que no hemos hablado hasta el momento).

Definición 2.2.5 (Modelo y Contramodelo) *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 $v \models F$. En caso contrario, se dice que v es contramodelo de F y se denota por $v \not\models F$.*

De forma análoga se dice que una valoración v es modelo de un conjunto de fórmulas $U = \{F_1, F_2, \dots\}$ si es modelo de toda fórmula del conjunto y se denota por $v \models U$.

Definición 2.2.6 (Satisfactibilidad) *Cierta fórmula F se dice satisfactible si existe una valoración v que es modelo suyo ($v \models F$), esto es, hay un mundo posible en el que la fórmula es verdadera. En caso contrario se dice que la fórmula es insatisfactible y se denota por $\perp$.*

Análogamente, un conjunto se dice satisfactible o consistente si existe una valoración v que es modelo suyo ($v \models U$). En caso opuesto se dice inconsistente o insatisfactible y se denota por $\perp$.

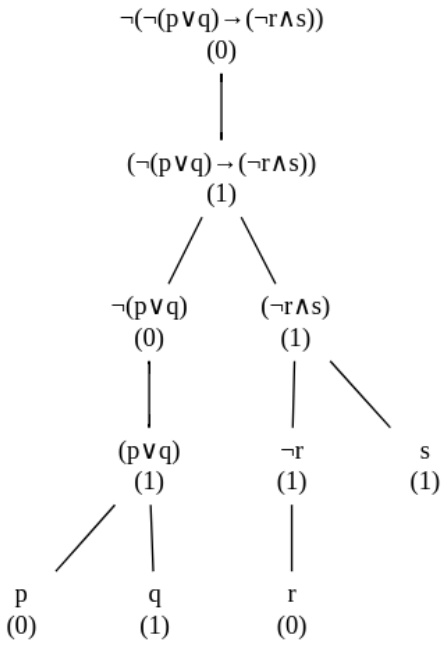


Figura 2.2: Evaluación en árbol de formación

Definición 2.2.7 (Validez lógica) *Una fórmula F se dice lógicamente válida o tautología si para toda valoración v, se tiene que v que es modelo suyo F, esto es, en todo mundo posible la fórmula es verdadera.*

Definición 2.2.8 (Conssecuencia lógica) *Una fórmula F se dice que es consecuencia lógica de un conjunto U si tomo modelo de U es también modelo de F, y se denota por $U \models F$. Esto es, si en un mundo se hace cierto U entonces necesariamente ha de ser cierto F.*

2.2.3. Algoritmos de decisión y el Problema SAT

Establecida la sintáxis y semántica de la lógica proposicional ya podemos expresar los hechos del mundo considerado e interpretar su veracidad. Ahora nos falta establecer métodos que permitan deducir información de los hechos planteados o permitan la demostración o refutación de hipótesis. Pasemos a ve entonces los conocidos como algoritmos de decisión, describiendo de forma general los más importantes y que han sido implementados en este proyecto.

Definición 2.2.9 (Algoritmo de decisión) *Un algoritmo de decisión es aquél que dado un conjunto de fórmulas Φ y una fórmula $A \in PROP$ devuelve sí cuando $A \in \Phi$ y NO en caso contrario.*

A priori pueda parecer un tema trivial, bastaría recorrer todos los elementos de Φ y verificar la pertenencia o no, pero el problema cobra relevancia cuando Φ es definido por un conjunto de características en vez de sus elementos, creándose conjuntos de extensión no finita y de elementos desconocidos. Algunos problemas especialmente importantes corresponden a:

- Problema SAT: Sea $\Phi = SAT = \{A \in PROP : A \text{ es satisfactible}\}$ (el conjunto de fórmulas satisfactibles).
- Problema de Validez: Sea $\Phi = TAUT = \{A \in PROP : A \text{ es tautología}\}$ (el conjunto de fórmulas satisfactibles).
- Problema de la Consecuencia: Sea $U \subseteq PROP$, y sea $\Phi = \mathcal{T}(U) = \{A \in PROP : U \models A\}$ (el conjunto de fórmulas que se siguen de U, esto es la *teoría* de U).

Este último problema modela justamente el *Problema Central de la Deducción*, que veíamos al comienzo del capítulo, por tanto podemos reducir nuestro objetivo primario (obtener un algoritmo eficaz, eficiente y completo que resuelva el Problema Central de la Deducción) a uno análogo: Obtener un algoritmo que dado un conjunto de fórmulas proposicionales U (la $\mathcal{BC}$) y otra fórmula F (*hipótesis*) decida si $U \models F$.

Hemos de destacar que en realidad los tres problemas anteriores resultan en realidad equivalentes, dado el *Teorema de la Consecuencia*, que establece que resultan análogos los siguientes supuestos:

- $U = \{F_1, F_2, \ldots, F_n\} \models F$ (F es consecuencia lógica de U).
- $F_1 \wedge F_2 \wedge \cdots \wedge F_n \rightarrow F \in TAUT$ (la veracidad de todos los supuestos de U implica necesariamente la veracidad de F).
- $\{F_1, F_2, \ldots, F_n, \neg F\}$ es inconsistente (no puede darse la veracidad de todos los elementos de U y la falsedad de F simultáneamente).

Aunque esta última propiedad no aparenta estar relacionada expresamente con los problemas anteriormente planteados, es equivalente al problema SAT ($F_1 \wedge F_2 \wedge \cdots \wedge F_n \wedge \neg F \notin SAT$) y es el punto de partida de muchos de los algoritmos de decisión desarrollados. Sin embargo y aunque algunos han demostrado comportarse de forma razonable, no se tiene, hasta el momento, ningún algoritmo capaz de resolver el problema SAT en un tiempo razonable (polinomial) y se cree, de hecho, que dicho algoritmo puede no existir, debido a la conjetura $P \neq NP$.

Si acudimos a la *Teoría de la Complejidad* la categoría de problemas P corresponde aquellos que son resolubles en tiempo polinomial por algoritmos deterministas), mientras que el problema NP corresponde a aquellos que son resolubles en tiempo polinomial por algoritmos no deterministas. El hecho de que SAT, que pertenece a la categoría NP-Completo sea resoluble de forma determinista en un tiempo polinomial, implicaría automáticamente que el resto de problemas considerados como NP también lo fueran.

Por tanto, los algoritmos que veremos, que son algoritmos completos, tendrán, en el caso peor complejidad $\Theta(2^n)$.

Algoritmo 2.1: Tabla de Verdad para fórmulas proposicionales

Tabla de Verdad(F)**:**

Input : F: Una fórmula proposicional
Output: T: Una tabla cuyas entradas a cada una de las posibles interpretaciones i junto con la valoración de F respecto a dicha interpretación

1. Obtener todas las posibles interpretaciones para F:
   - 1.1 Obtener todas las variables proposicionales que intervienen en F, de forma que se tiene el conjunto $VARS = \{x_1, \ldots, x_n\}$
   - 1.2 Obtener todas las posibles interpretaciones para F con todas las combinaciones para cada variable x_i con los valores booleanos $\{0, 1\}$, esto es $\{0, 1\}^n$.
2. Crear una tabla T vacía con $n + 1$ columnas las n primeras para los valores de x_i y la última para el valor de verdad de F según la interpretación correspondiente.
3. Para cada posible interpretación v:
   - 3.1 Evaluar F respecto a v: $v(F)$
   - 3.2 Añadir a T la entrada $v_1, \ldots, v_n, v(F)$, esto es la interpretación y la evaluación de F respecto a la misma.

return T. La tabla de verdad de F

fin

Tablas de Verdad

Propuesta por Charles S. Peirce y popularizada por Ludwig Wittgenstein a principios del siglo XX, se trata de una tabla que hace corresponder a cada posible interpretación para una fórmula (o conjunto, equivalente a la conjunción de las fórmulas del conjunto) la valoración de la fórmula respecto a dicha valoración. De esta forma todas aquellas entradas con valor verdadero corresponden a los modelos y todas aquellas con valor falso a los contramodelos. Además, nótese que resulta sencilla la identificación de tautologías y contradicciones siendo para las primeras todas las entradas verdaderas y para las segundas todas falsas.

Nótese que la complejidad de dicho algoritmo es (en todo caso) de orden exponencial (dada la definición de las posibles interpretaciones como el espacio $\underbrace{\{0,1\} \times \cdots \times \{0'1\}}_{n} = \{0,1\}^n$) lo que resulta intratable para fórmulas en las que intervengan una cierta cantidad de variables. El *algoritmo 2.1* se muestra el pseudocódigo del algoritmo. Téngase en cuenta que podría aplicarse también a conjuntos de fórmulas sin más que tomar el conjunto como la conjunción de las fórmulas del mismo.

Tableros Semánticos

Propuesto a mediados del siglo XX por Evert W. Beth y Jaakko Hintikka (de forma independiente) y desarrollada por Raymond M. Smullyan en el último cuarto de siglo, un *tablero semántico* (también árbol semántico o tablero analítico) corresponde a un método basado en la sucesión de fórmulas que son generadas por descomposición, a partir de dos reglas fundamentales de operación (regla α y regla β) y cuyo propósito es reducir la satisfactibilidad de un conjunto de fórmulas a la satisfactibilidad de conjuntos de literales, proporcionando además de un método de demostración (basado en el método clásico de *Reducción al Absurdo*), un método de extracción de (contra)modelos en caso de que el conjunto no sea inconsistente.

- Reglas de reducción

Antes de establecer formalmente las reglas vamos a realizar algunas observaciones que aportarán cierto interés al desarrollo de los fundamentos de los tableros. Aunque no lo notamos en su momento, cuando definimos las funciones de verdad para las conectivas lógicas, se puede apreciar una clara analogía en la definición de las conectivas $\wedge$ y $\leftrightarrow$, pudiendo reducir el patrón $(A \leftrightarrow B)$ al patrón $(A \rightarrow B) \wedge (B \rightarrow A)$, y de las conectivas $\vee$ y $\rightarrow$, pudiendo reducir el patrón $(A \rightarrow B)$ al patrón de $(\neg A) \vee (B)$.

Estas analogías, dan lugar a dos comportamientos distintos para las fórmulas, el comportamiento conjuntivo (fórmulas α) y el comportamiento disyuntivo (fórmulas β). Las primeras establecen restricciones aditivas a la satisfactibilidad del conjunto, mientras que las segundas establecen condiciones dicotómicas a la satisfactibilidad del mismo. Y este justamente, el comportamiento de los tableros. Antes de dar una definición algorítmica para estos dos comportamientos, notemos el comportamiento de la conectiva $\neg$ cuya influencia cambia el carácter de la fórmula a la que afecta (atestiguado por las fórmulas de De Morgan), de forma que el patrón $\neg(A[\wedge/\vee]B)$ es reducible al patrón $(\neg A)[\vee/\wedge](\neg B)$.

Según lo planteado previamente, es claro que la aparición de una doble negación (fórmula dN) produce dos cambios en el comportamiento, lo que es equivalente a no producir cambio alguno, por lo que el patrón $\neg\neg A$ puede ser reducido simplemente a A.

Formalmente podemos establecer las reglas de reducción anteriores como:

- *Regla* α: Dado el conjunto $U = \{F_1, F_2, ..., F_i, ...\}$ tal que F_i es de tipo α los modelos de U, y por tanto su consistencia, equivalen a los modelos del conjunto considerando conjuntamente las componentes de F_i de forma independiente, esto es, $models(U) = models((U \backslash \{F_i\}) \cup \{F_{i1}, F_{i2}\})$.
- *Regla* β: Dado el conjunto $U = \{F_1, F_2, ..., F_i, ...\}$ tal que F_i es de tipo β los modelos de U, y por tanto su consistencia, equivalen a todos los modelos de los conjuntos considerando disyuntivamente

las componentes de F_i de forma independiente, esto es, $models(U) = models((U \backslash \{F_i\}) \cup \{F_{i1}\}) \cup models((U \backslash \{F_i\}) \cup \{F_{i2}\})$.

- *Regla dN*: Dado el conjunto $U = \{F_1, F_2, ..., F_i, ...\}$ tal que F_i es de tipo dN los modelos de U, y por tanto su consistencia, equivalen a todos los modelos de los conjuntos considerando disyuntivamente las componentes de F_i de forma independiente, esto es, $models(U) = models((U \backslash \{F_i\}) \cup \{F_{i1}) \cup models((U \backslash \{F_i\}) \cup \{F_{i2})$.

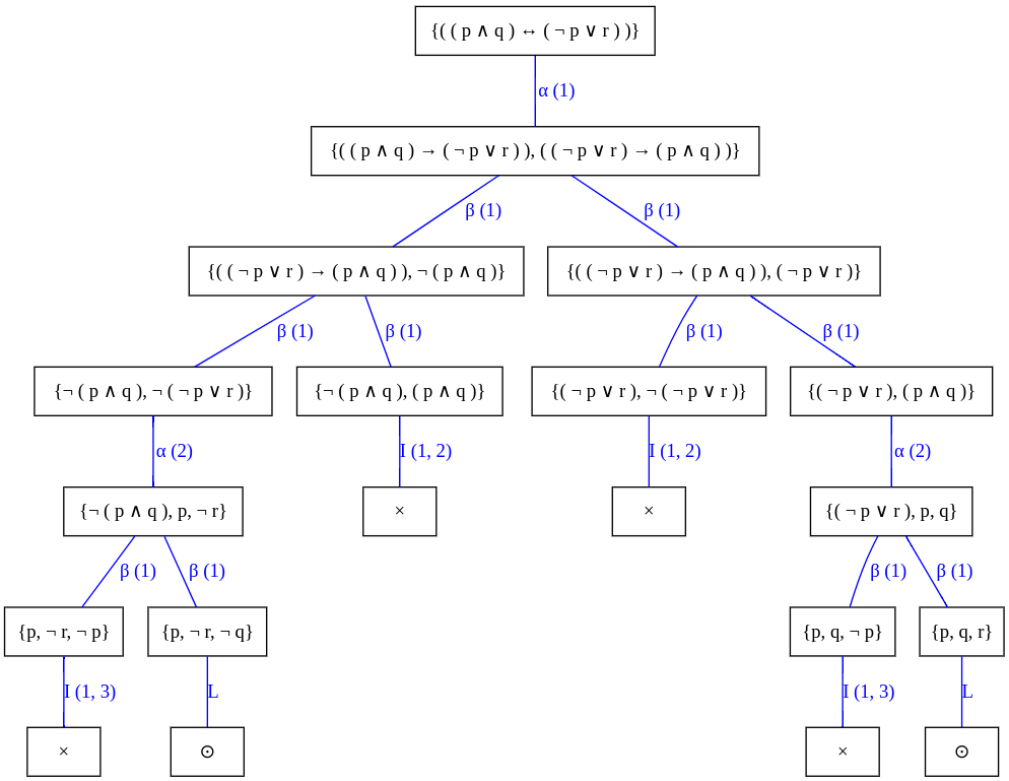


Figura 2.3: Tablero Semántico
Fuente propia: generado con LogicUS

De forma que la aplicación sucesiva de las reglas (que suele representarse en forma de árbol como el de la *figura 2.3*) reduce el cálculo de los modelos de un conjunto de fórmulas a los modelos de uno o más conjuntos de literales (átomos o negaciones de átomos), que expresan únicamente hechos en el conjunto (mundo) bajo estudio, y que no son expansibles mediante ninguna regla. Así, la presencia de literales contrarios (L y $\neg L$) en un conjunto denotan la insatisfactibilidad del mismo, mientras que la ausencia de los mismos notan la satisfactibilidad, dando los modelos considerando verdaderas las variables cuyos literales aparecen positivos y falsas las que aparecen negativos e indiferentes aquellas que no aparecen. (NOTA: La insatisfactibilidad puede ser detectada de forma anticipada mediante la presencia de dos fórmulas complementarias F y $\neg F$ en un mismo conjunto, sin necesidad de llegar a los conjuntos de literales, que saldrán, lógicamente, inconsistentes).

Hacer incapié en el carácter de refutación del procedimiento planteado, de forma que para comprobar la satisfactibilidad de un conjunto basta con que una de las ramas salga abierta, para comprobar la insatisfactibilidad todas han de salir cerradas y para comprobar la la validez lógica ha de comprobarse la insatisfactibilidad de la negación.

En el *algoritmo 2.2* se encuentra detallado el pseudocódigo del algoritmo. De forma que resulta fácil probar que el algoritmo es completo y finito. En el peor de los casos también de complejidad $\Theta(2^n)$, aunque sólo si todas las fórmulas y subfórmulas son de tipo β.

Algoritmo 2.2: Tablero Semántico para conjuntos de fórmulas proposicionales

Tablero Semántico (U)**:**

Input : U: Un conjunto de fórmulas proposicionales
Output: T: Un árbol que representa la aplicación de las reglas y los conjuntos considerados

1. Hacer r la raíz de T y etiquetarla con $U_r = U$
2. Mientras T tenga hojas no marcadas seleccionar una de ellas U_n y hacer:
   - 2.1 Si U_n contiene un par de fórmulas complementarias marcar la hoja 'cerrada' $(\times)$
   - 2.2 Si no, si U_n es un conjunto de literales marcar la hoja 'abierta' $(\bigcirc)$
   - 2.3 Si no, elegir una fórmula A, que no sea literal, de U_n
     - 2.3.1 Si A es de tipo α o dN entonces añadir a n un descendiente con el resultado de aplicar la regla correspondiente.
     - 2.3.2 Si no, A es de tipo β, por tanto añadir a n dos descendientes según la regla β.

end

Formas Normales y Cláusulas

La Formas Normales (FNC y FND) resultan de la traslación al contexto lógico de las Formas Normales definidas el contexto del álbebra de Boole (muy utilizada en circuitos electrónicos). De forma que una fórmula está en *Forma Normal Conjuntiva* si se encuentra expresada como conjunción de disyunciones de literales ; y está en *Forma Normal Disyuntiva* si se encuentra expresada como disyunción de conjunciones de literales.

$$FNC = \bigwedge_{i=1}^{n} \left(\bigvee_{j=1}^{m_i} L_{i\,j} \right) \qquad FND = \bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{m_i} L_{i\,j} \right)$$

Estas formas tienen algunas características interesantes para la detección de tautologías y contradicciones, y además resulta que existen resultados que demuestran que toda fórmula es expresable en cualquiera de las dos formas. En el *algoritmo 2.3* se expone el pseudocódigo del algoritmo de transformación a formas normales mediante el uso del álgebra (aunque existen otras aproximaciones, por ejemplo bajo el uso de Tableros Semánticos).

Algoritmo 2.3: Transformación a Formas Normales

Transformación a FN (U)**:**

Input : F: Una fórmula proposicional
Output: F': Una fórmula proposicional en FN equivalente a F

1. Eliminar todas las dobles implicaciones aplicando: $A \leftrightarrow b \equiv (A \rightarrow B) \wedge (B \rightarrow A)$.
2. Eliminar todas las implicaciones aplicando: $A \rightarrow b \equiv \neg A \rightarrow B) \vee B$.
3. Interiorizar las negaciones aplicando la leyes de De Morgan: $\neg(A[\wedge/\vee]B) \equiv \neg A[\vee/\wedge]\neg B$
4. Eliminar las dobles negaciones: $\neg\neg A \equiv A$
5. FNC: Aplicar la ley distributiva $A \vee (B \wedge c) = (A \vee B) \wedge (A \vee C)$
5. FND: Aplicar la ley distributiva $A \wedge (B \vee c) = (A \wedge B) \vee (A \wedge C)$

end

Además de otros resultados de las formas normales en cuanto a satisfactibilidad y validez, la FNC da pie a la definición de la *Forma Clausal*, que expresa cada fórmula como una colección de conjuntos de literales, de forma que cada conjunto corresponde a una disyunción de literales, mientras que la colección es concebida como la conjunción de los conjuntos, esto es, una conjunción de disyunciones de literales, o en otras palabras, una representación simplificada de la FNC.

$$Forma\,Clausal = \{\{L_{1,1}, L_{1,2}, \dots\}, \{L_{2,1}, L_{2,2}, \dots\}, \dots\}$$

Nótese que dada esta formulación, el conjunto es consistente si y sólo si lo son simultáneamente todas las cláusulas que lo forman y una cláusula será consistente si alguno de sus literales se hacen ciertos en la valoración considerada. Esta representación resulta tan cómoda y eficiente que los algoritmos más potentes desarrollados para la resolución del problema SAT, basan su funcionamiento precisamente en el manejo de las cláusulas.

Algoritmo DPLL

El algoritmo DPLL fue propuesto por Martin Davis, Hilary Putnam, George Logemann y Donald W. Loveland en la segunda mitad del siglo XX y aún hoy es utilizado como base de los sistemas más eficientes de resolución del problema SAT y demostradores automáticos.
Hemos de notar que este algoritmo no trabaja sobre los conjuntos de fórmulas 'en bruto', sino que es necesario una fase de preprocesamiento, para expresar las fórmulas en la ya vista *Forma Clausal*. Una vez expresado el conjunto de fórmulas como conjunto de cláusulas, el algoritmo se basa en dos operaciones básicas: *propagación* (para la simplificación del conjunto mediante la eliminación de cláusulas y literales) y *división* (organización de la búsqueda cuando no es posible propagar).

Algoritmo 2.4: Algoritmo Davis–Putnam–Logemann–Loveland

DPLL(U)**:**

Input : U: Un conjunto de cláusulas proposicionales
Output: *Bool* : Que indica si el conjunto es satisfactible o insatisfactible

1. Si $U = \emptyset$ entonces devolver $U \in SAT$
2. Si no, si $\square \in U$ entonces devolver $U \notin SAT$
3. Si no si U posee unidades, tomar una de ellas u y devolver *DPLL(Propagación*(u, U)*)*
4. Si no, tomar un literal L presente en U y hacer:
   - $\{U_1, U_2\} =$ *División*(L, U)
   - Devolver *DPLL(*U_1*)* $\vee$ *DPLL(*U_2*)*

end

Propagación de unidades

Se toma una cláusula unitaria del conjunto (que contiene un único literal) (se toma dicho literal como cierto, necesario para la satisfactibilidad de la cláusula y por tanto del conjunto) y se eliminan del conjunto todas las cláusulas que contienen dicho literal (y que por ende, se hacen ciertas).

Seguidamente se elimina de cada una de las clausulas, que lo contengan, el literal complementario (considerado como falso y que por tanto no puede dar lugar a la satisfactibilidad de la cláusula). Repitiéndose el proceso en tanto en cuanto se tenga alguna cláusula unitaria.

División
En caso de no tener cláusulas unitarias (y en ausencia del conjunto o cláusula vacíos), se elige un literal

(heurísticamente) dando lugar a una división de la búsqueda en dos ramas, una suponiendo dicho literal cierto y otra suponiendo el mismo falso. De forma que el método es completo, esto es, cubre toda la posible casuística y determinista (da un resultado fijo dado un conjunto de fórmulas).

Si bien, hemos planteado que el algoritmo trabaja eficientemente con las cláusulas, su complejidad es, en el caso peor, de orden exponencial $\Theta(2^n)$. En el *algoritmo 2.4* se encuentra detallado el pseudocódigo del algoritmo.

Resolución y estrategias

Más allá de la importancia intrínseca de los sistemas deductivos y el encadenamiento, su presentación sirve como justificación del último de los algoritmos que se presentan dentro de este trabajo, denominado *Resolución Proposicional* o simplemente *Resolución*. Al igual que el algoritmo DPLL, trabaja sobre cláusulas por lo que requiere de un preprocesamiento de las fórmulas; pero existe una diferencia fundamental, la resolución es utilizada únicamente para la refutación, de forma que sólo se tiene completitud en este aspecto, de forma que un conjunto de fórmulas será inconsistente si y sólo si somos capaces de obtener la cláusula vacía. Nótese que esto, es suficiente para la resolución del *Problema de la Consecuencia* y por ende el *Problema SAT*.

El *Algoritmo de Resolución* está inspirado en dos reglas clásicas (provenientes del Silogismo Aristotélico), el *Modus Ponens* y el *Modus Tollens*, y en el encadenamiento. Expresando dichas reglas en formato para fórmulas y para cláusulas:

Modus Ponens	$\dfrac{p, p \to q}{q}$	$\dfrac{\{p\}, \{\neg p, q\}}{\{q\}}$
Modus Tollens	$\dfrac{\neg q, p \to q}{\neg p}$	$\dfrac{\{\neg q\}, \{\neg p, q\}}{\{\neg p\}}$
Encadenamiento	$\dfrac{p \to q, q \to r}{p \to r}$	$\dfrac{\{\neg p, q\}, \{\neg q, r\}}{\{\neg p, r\}}$

A partir de las reglas anteriores parece verificarse, y es justamente lo que se denomina como *Regla de Resolución*, que si se tienen dos cláusulas con un literal complementario, entonces se tiene como consecuencia la unión de las cláusulas eliminando dichos literales complementarios.

$$\textit{Resolución} = \frac{\{L_1, L_2, \ldots, L, \ldots, L_n\}, \{L'_1, L'_2, \ldots, L^c, \ldots L'_m\}}{\{L_1, L_2, \ldots L_n\, L'_1, L'_2, \ldots L'_m\}}$$

Téngase en cuenta que, con vista a la refutación, una de las dos clausulas se satisface (ya que son disyunciones y o bien L o bien L^c ha de ser necesariamente cierto), por tanto, si se consigue satisfacer $\{L_1, L_2, \ldots L_n\, L'_1, L'_2, \ldots L'_m\}$, automáticamente se puede satisfacer ambos antecedentes sin más que elegir adecuadamente el valor de verdad de L, pero además, si no se consigue satisfacer $\{L_1, L_2, \ldots L_n\, L'_1, L'_2, \ldots L'_m\}$ será imposible satisfacer ambos antecendentes ya que la satisfactibilidad recaería sobre L, que haría cierta únicamente una de las dos cláusulas.

Sin embargo, este mecanismo puede generar multitud de resolventes (*Resolución por Saturación*) que no lleguen a la cláusula vacía (recordemos que el problema *SAT* es *NP*-completo). Para luchar contra ello, se han diseñado algunas mejoras (basadas en eliminación de tautologías y subsunción) así como estrategias de resolución que permiten guiar (parcialmente) la búsqueda. Las más relevantes son: *Regular*, *Lineal*, *Unitaria*, *Positiva*, *Negativa* y *Por entradas*. La técnica de resolución puede verse como un procedimiento de búsqueda de tal forma que las distintas estrategias podrían relacionarse con el modo en que se exploran los distintos elementos que son generados. En el *algoritmo 2.5* presentamos un procedimiento genérico en el que algunos pasos dependen del uso de cada heurística.

Algoritmo 2.5: Algoritmo de Resolución proposicional

Resolución(U)**:**

Input : U: Un conjunto de clausulas proposicionales
Output: *Bool* : Que indica si el conjunto es satisfactible o insatisfactible

1. Hacer $U' \leftarrow eliminaIrrelevantes(U)$,
2. Tomar C_1 de U' y hacer $U'' \leftarrow U' \bigcup_{C_2 \in U' \setminus \{C_1\}} Res(C_1, C_2)$
3. Mientras $\square \notin U''$ y $U'' \neq U'$:
   3.1 Hacer $U' \leftarrow U''$
   3.2 Tomar C_1 de U' y hacer $U'' \leftarrow U' \bigcup_{C_2 \in U \setminus \{C_1\}} Res(C_1, C_2)$
   3.3 Hacer $U'' \leftarrow eliminaIrrelevantes(U'')$
4. Si $\square \notin U''$ devolver $U \notin SAT$
5. Si no, devolver $U \in SAT$

end

De forma que la elección de la cláusula así como la limitación de las cláusulas que son resueltas con la misma dan lugar a las distintas estrategias de resolución.

2.2.4. Las limitaciones de la Lógica Proposicional

Aunque la lógica proposicional posee un semántica sencilla y existen algoritmos de decisión (aunque poco eficientes) para sus problemas básicos, como SAT o la consecuencia lógica, la expresividad de la lógica proposicional es bastante limitada, lo 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:

- Toda persona es experta en un tema si sabe más que otra persona de ese tema.
- No existe un tema en el que todo el mundo sea experto.

El razonamiento anterior es correcto y sin embargo no es modelable en el marco de la Lógica Proposicional. ¿Cómo expresar que alguien es experto en un tema?¿Cómo expresar que alguien sabe más que otro?¿cómo expresar el concepto de tema y si existen o no? Es aquí precisamente, en esta debilidad que presenta LP, donde comienza el ámbito de la Lógica de Primer Orden, que abordaremos en la siguiente sección.

2.3. El formalismo de la Lógica de Primer Orden

La lógica proposicional nos permite expresar conocimiento sobre situaciones que son de nuestro interés, mediante enunciados declarativos. Decimos que estos enunciados son declarativos en el sentido lingüístico del término, esto es, se trata de expresiones del lenguaje natural que son o bien verdaderas, o bien falsas; en contraposición a los enunciados imperativos e interrogativos. La lógica proposicional es declarativa en este sentido, las proposiciones representan hechos que se dan o no en la realidad. La Lógica de Primer Orden tiene un compromiso ontológico más fuerte, que permite abordar cuestiones como:

- Trabajar con objetos de un dominio de forma individualizada, no solo a nivel global.

- Representar propiedades de objetos y las relaciones entre ellos.
- Realizar cuantificación sobre los objetos de un dominio, esto es, expresar en qué medida se tiene una propiedad sobre un conjunto de objetos.

Como en todo sistema formal, y al igual que se presentó para la Lógica Proposicional hemos describir los elementos que conforman la Lógica de Primer Orden, esto es, la sintaxis, semántica y algoritmos de decisión.

2.3.1. Sintaxis 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 lenguajes 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. Nótese que:
    - 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.*

Este aparato permite la construcción de distintas expresiones, que componen las fórmulas de LPO. Vamos a exponer una diferenciación de dichas expresiones, distinguiendo *términos* y *fórmulas.*

Los *términos* se identifican con posibles objetos del mundo y engloban a:

- Constantes para hablar de objetos específicos.
- Variables para hablar de objetos genéricos.
- Funciones aplicadas a otros términos más pequeños, según su aridad.

Para ilustrar los distintos conceptos consideremos el siguiente lenguaje proveniente del mundo romano:

$$LR = \left\{ \underbrace{\textbf{César}, \textbf{Marco}}_{constantes}, \underbrace{P^1, L^2, O^2, R^1, IA^2}_{símbolos\ de\ predicado}, \underbrace{f^1}_{símbolo\ de\ función} \right\}$$

Donde **César** y **Marco** son constantes, P y R son predicados unarios que denotan *ser pompeyano* y *ser romano*, respectivamente; L, O, IA son predicados binarios que denotan *ser leal a*, *odiar a* , *intentar*

asesinar a, respectivamente; y f una función unaria que represente el concepto de *padre de*. De forma que son términos de LR: *Constantes*: **Marco**, **César**, *Funciones*: $f(\textbf{César})$, $f(x)$, $f(f(x))$, ..., *Variables*: x, y, x_1, ...

Las *fórmulas* se identifican con afirmaciones sobre los objetos del mundo, permitiendo hablar de la veracidad o falsedad de las afirmaciones. Están formadas a partir de predicados sobre términos, y construcciones lógicas de estos predicados (conjunciones, implicaciones, cuantificaciones, etc.). Veamos esto formalmente.

Las fórmulas pueden corresponder a:

- Átomos o Fórmulas atómicas. Corresponden a las expresiones $p(t_1, t_2, \ldots, t_n)$, tal que p es un símbolo de predicado de aridad n y t_i son términos.
- Fórmulas no atómicas. Corresponden a expresiones formadas a partir de fórmulas atómicas, mediante el empleo de conectivas y/o cuantificadores.

De esta forma, son fórmulas de L:

- Toda fórmula atómica.
- Si F y G son fórmulas de de L entonces $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$, también son fórmulas de L.
- Si x es una variable y F es una fórmula de L, entonces $\exists x\, F$ y $\forall x\, F$ son también fórmulas de L.

Volviendo al mundo romano, algunas posibles fórmulas del lenguaje LR:

- *Átomos*: $P(\textbf{César})$, $L(\textbf{César}, \textbf{Marco})$, $IA(\textbf{Marco}, f(x))$.
- *Fórmulas compuestas*: $\forall x \exists y\, L(x, y)$, $\forall x(R(x) \rightarrow (L(x, \textbf{César}) \vee O(x, \textbf{César}))$

Al igual que expusimos para la Lógica Proposicional, vamos a dar unas cuantas reglas que permiten la simplificación de la notación, con objeto de facilitar la lectura y escritura de las fórmulas. Se adoptan los siguientes criterios:

- Se omitirán los paréntesis externos.
- Las prioridades de las conectivas siguen el mismo orden que el expuesto en LP: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ (para la última se recomienda mantener los paréntesis).
- Los cuantificadores tienen prioridad sobre las conectivas.

Al igual que en las fórmulas de LP, la definición de las fórmulas en LPO presenta una estructura recursiva:

- *Caso base*. Fórmulas atómicas.
- *Casos recursivos* Aplicación de las conectivas y cuantificadores sobre fórmulas de LPO.

Y de forma análoga a la presentada en LP, se puede plasmar dicha estructura recursiva en el *Árbol de formación* (esencialmente único), de forma que el nodo raíz corresponde a la fórmula completa y las hojas corresponden a las fórmulas atómicas que participan en la fórmula. Al igual que en LP, toda fórmula que aparezca en algún nodo (ya sea un nodo interno o una hoja) diremos que es **subfórmula** de la fórmula original.

Para la fórmula $\forall x(R(x) \rightarrow (L(x, \textbf{César}) \vee O(x, \textbf{César}))$, se tendría el árbol de formación mostrado en la *figura 2.4*

Formulas bien formadas, renombrado de variables y clausura de fórmulas.

Hemos estudiado la sintaxis formal de las fórmulas, pero aún faltan algunos detalles por completar para establecer qué fórmulas están bien formadas y cuáles no. Un tema importante es el tratamiento de la cuantificación, esto es, a qué apariciones u ocurrencias (denominadas estancias) de una variable afecta un cuantificador.

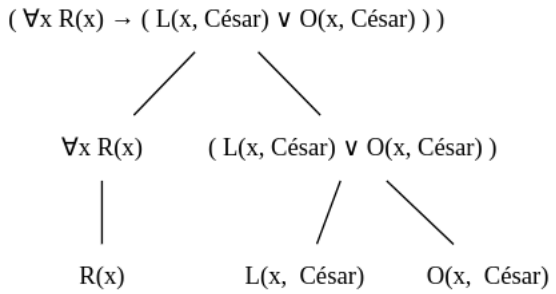
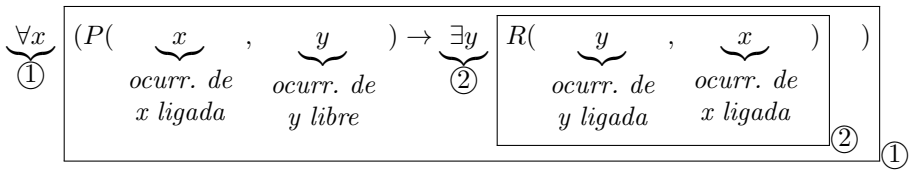


Figura 2.4: Árbol de formación de una fórmula LPO

Una ocurrencia de una variable está afectada por un cuantificador (se dice que es una *estancia u ocurrencia ligada*) si hay un cuantificador sobre dicha variable actuando sobre la (sub)fórmula que la contiene. En otro caso diremos que se trata de una *estancia u ocurrencia libre*. Ilustrémoslo con un ejemplo. En la fórmula:



En la fórmula podemos apreciar que el cuantificador, $\forall x$, afecta a toda la fórmula siguiente, por tanto todas las ocurrencias de x que aparezcan en la fórmula serán ocurrencias ligadas. Sin embargo, $P(x, y)$ no está afectado por ningún otro cuantificador, por tanto, la ocurrencia de y en $P(x, y)$ es libre. Por otra parte, $R(x, y)$ sí está afectado por el cuantificador $\exists y$, por tanto tanto la ocurrencia de x como la de y en $R(y, x)$ son ligadas.

De forma similar, una variable x se dice *variable libre* en una fórmula F si existe alguna ocurrencia libre de x en F. Una variable x se dice *variable ligada* en una fórmula F si existe alguna ocurrencia ligada de x en F. De forma que en una fórmula una variable puede ser al mismo tiempo *variable libre* y *variable ligada*. Por ejemplo, en la fórmula anterior x es una variable *exclusivamente cerrada* (todas sus ocurrencias son ligadas), sin embargo y es una variable *libre* y *ligada* (hay una ocurrencia libre y otra ligada).

En base a los conceptos anteriores se definen los conceptos:

- Un término se dice *cerrado* si no contiene ninguna variable. Por ejemplo, **César**, $f(\mathbf{Marco})$, $f(f(\mathbf{César}))$ son términos cerrados en el lenguaje LR.
- Una fórmula se dice *cerrada* si no contiene variables libres, o equivalentemente si todas las estancias de todas las variables son ligadas. Por ejemplo las fórmulas $\forall x(L(x, \mathbf{César}) \vee O(\mathbf{César}, x))$, $\forall x L(x, \mathbf{César}) \rightarrow \neg\exists y O(\mathbf{César}, y)$ son *fórmulas cerradas*, mientras que las fórmulas $\forall x L(x, \mathbf{César}) \vee O(\mathbf{César}, x)$, $\forall x L(x, \mathbf{César}) \rightarrow \neg\exists y IA(y, x)$ no lo son.
- Una fórmula se dice *abierta* si no contiene cuantificadores, esto es todas las variables son *variables exclusivamente libres*. Por ejemplo son fórmulas abiertas $P(x)$, $R(\mathbf{César})$, $P(x) \leftrightarrow \neg IA(f(x), \mathbf{Marco})$.

Aunque las fórmulas estén bien escritas sintácticamente, incluso aunque sean cerradas, es posible que su interpretación sea ambigua, esta ambigüedad suele venir dada por un mal uso de los cuantificadores y las variables, por ejemplo tengamos la fórmula:

$$\forall z \forall x \exists y \left(P(x, y) \rightarrow \left(P(x, z) \vee \exists z \left(P\left(y, \boxed{z}\right) \wedge P(x, y)\right)\right)\right)$$

En efecto, la ocurrencia de z, señalada, es claramente ligada, pero ¿a qué cuantificador?. Es ambiguo.

Precisamente por ello se establece que Diremos que una fórmula está *bien formada* (*FBF* o *WFF*) si es correcta sintácticamente y no contiene dos cuantificadores anidados actuando sobre la misma variable. Claramente, la fórmula del ejemplo previo no está **bien formada** y hay una ambigüedad en la interpretación de la fórmula. Para resolver dicha ambigüedad vamos a adoptar el criterio de que toda ocurrencia está ligada (si es que lo está) al cuantificador de nivel superior más cercano en el árbol de formación. En el caso del ejemplo al existencial.

Además de adoptar un criterio, podemos, mejor, realizar un renombrado de las variables, manteniendo el sentido original de las fórmulas, de forma que la fórmula anterior podríamos renombrarla mejor como:

$$\forall z_1 \, \forall x_1 \, \exists y_1 \, \left(P\left(x_1, y_1\right) \rightarrow \left(P\left(x_1, z_1\right) \vee \exists z_2 \, \left(P\left(y_1, z_2\right) \wedge P\left(x_1, y_1\right)\right)\right)\right)$$

El caso anterior no es el único en el que es conveniente renombrar las variables, por ejemplo, tengamos la fórmula:

$$\forall x \, \exists y \, \left(P\left(x, y\right) \rightarrow \left(P\left(x, \textcircled{z}\right) \vee \exists z \, \left(P\left(y, \boxed{z}\right) \wedge P\left(x, y\right)\right)\right)\right)$$

La primera ocurrencia de z señalada es libre y la segunda es ligada, claramente el mismo símbolo de variable representa objetos distintos, lo que produce una cierta ambigüedad en la fórmula. Sin embargo, podemos renombrar la fórmula como:

$$\forall x_1 \, \exists y_1 \, \left(P\left(x_1, y_1\right) \rightarrow \left(P\left(x_1, z\right) \vee \exists z_1 \, \left(P\left(y_1, z_1\right) \wedge P\left(x_1, y_1\right)\right)\right)\right)$$

De forma que el criterio de renombrado de variables es tomado de forma que son renombradas todas las ocurrencias ligadas de cada variable añadiendo el subíndice correspondiente según el orden y cardinalidad de aparición del cuantificador (sobre dicha variable) al que está ligada dicha ocurrencia (tomando el orden según el recorrido en profundidad del AF de la fórmula).

Como veremos más adelante, los algoritmos de decisión que vamos a estudiar trabajan únicamente sobre fórmulas cerradas, por lo que hemos de *clausurar* todas aquellas fórmulas que no sean cerradas, cuantificando las variables libres de la fórmula (renombrando aquellas variables que sean, simultáneamente, libres y ligadas). Se admiten dos tipos de clausura para las fórmulas:

- Clausura universal se trata de cerrar la fórmula a base de cuantificar universalmente (por la izquierda) las variables libres de las fórmulas. Si $v_1, \ldots, v_n$ corresponden a las variables libres de una fórmula F, la clausura universal de la fórmula corresponde a la fórmula $\forall v_1 \ldots \forall v_n \, F$
- Clausura existencial se trata de cerrar la fórmula a base de cuantificar existencialmente (por la izquierda) las variables libres de las fórmulas. Si $v_1, \ldots, v_n$ corresponden a las variables libres de una fórmula F, la clausura universal de la fórmula corresponde a la fórmula $\exists v_1 \ldots \exists v_n \, F$

Sustituciones

La aplicación de una sustitución a un término consiste en intercambiar las variables que participan en dicho término según la correspondencia que establece la sustitución y se denota por $\theta(t)$ o también $t\{x_1/t_1, \ldots, x_n/t_n\}$. Formalmente:

$$\theta(t) = \begin{cases} \theta(x_i) & si\, t = x_i \quad [t \text{ es una variable}] \\ f(\theta(t_1), \ldots, \theta(t_m)) & si\, t = f(t_1, \ldots, t_m) \quad [t \text{ es una función de aridad } m] \end{cases}$$

Nótese el carácter simultáneo de la sustitución, esto es, la sustitución de todas las variables se realiza simultáneamente de forma que no se tiene $\{x_i/x_j, x_j/t_j\} \equiv \{x_i/t_j, x_j/t_j\}$. Por ejemplo, tengamos el término del ejemplo anterior y la sustitución $\theta_2 = \{x/(x+y), y/\mathbf{0}\}$, entonces:

$$((x+y)+z)\{x/(x+y), y/\mathbf{0}\} \neq ((x+\mathbf{0})+\mathbf{0})+z$$

$$((x+y)+z)\{x/(x+y), y/\mathbf{0}\} = ((x+y)+\mathbf{0})+z$$

La aplicación de una sustitución a un término consiste en intercambiar todas las ocurrencias **libres** de las variables del dominio de la sustitución en F por los términos que les corresponden en la sustitución. Se denota por $\theta(F)$ o también $F\{x_1/t_1, \ldots, x_n/t_n\}$ de forma que:

$$\theta(F) \equiv \begin{cases} P(\theta(t_1), \ldots, \theta(t_n)) & si\, F \equiv P(t_1, \ldots, t_n) \\ \neg\theta(G) & si\, F \equiv \neg G \\ \theta(G) \wedge \theta(H) & si\, F \equiv G \wedge H \\ \theta(G) \vee \theta(H) & si\, F \equiv G \vee H \\ \theta(G) \rightarrow \theta(H) & si\, F \equiv G \rightarrow H \\ \theta(G) \leftrightarrow \theta(H) & si\, F \equiv G \leftrightarrow H \\ \exists x\, G\{x_i/t_i : x_i \in dom(\theta) \wedge x_i \neq x\} & si\, F \equiv \exists x\, G \\ \forall x\, G\{x_i/t_i : x_i \in dom(\theta) \wedge x_i \neq x\} & si\, F \equiv \forall x\, G \end{cases}$$

Sin embargo, no toda sustitución es admisible para una fórmula, por ejemplo, téngase la fórmula $F \equiv \exists x\, \neg(x = y)$ y téngase la sustitución $\theta = \{y/x\}$, entonces $\theta(F) = \exists x \neg(x = x)$. Claramente el sentido de la fórmula ha cambiado, mientras que en la primera fórmula se establece que debe haber, al menos, dos objetos distintos en la segunda se tiene que existe al menos un objeto que es distinto de sí mismo (que es sencillamente falso). Por ello, se establece que, Una variable, $x_i \in dom(\theta)$, de una fórmula, F, es sustituible por el término correspondiente, t_i, si y sólo si la aplicación de la sustitución, $\theta(F)$ no produce nuevas estancias ligadas. Formalmente se se da alguna de las siguientes condiciones:

1. F es atómica.
2. $F \equiv \neg G$ y x_i es sustituible por t_i en G
3. $F \equiv G\,(\wedge | \vee | \rightarrow | \leftrightarrow)\, H$ y x_i es sustituible por t_i en G y en H.
4. $F \equiv \exists x G$ tal que o bien $(x = x_i)$, o bien $(x \neq x_i) \wedge (x$ *no ocurre en* $t_i) \wedge (x_i$ *es sustituible en* $G)$.
5. $F \equiv \forall x G$ tal que o bien $(x = x_i)$, o bien $(x \neq x_i) \wedge (x$ *no ocurre en* $t_i) \wedge (x_i$ *es sustituible en* $G)$.

De ahora en adelante, cuando escribamos $F\{x/t\}$ supondremos que x es sustituible por t en F.

Una vez presentados los principales aspectos sintácticos en la Lógica de Primer Orden vamos a pasar a presentar la semántica. Nótese que elevar la capacidad expresiva ha producido también un incremento en la complejidad del lenguaje y, como veremos, también en la semántica, pasando incluso de un ámbito decidible a un ámbito indecidible.

2.3.2. Semántica de la Lógica de Primer Orden

Si bien la sintaxis establece qué formulas son sintácticamente correctas y cuáles no, o si es correcto cambiar un elemento por otro, la semántica tiene por objetivo dotar de significado a los términos y fórmulas de un Lenguaje de Primer Orden. Veamos cuales son los elementos que conforman la semántica.

Interpretaciones o L-estructuras

Al igual que en la fórmula se tenían las interpretaciones, que, básicamente asignaban valores de verdad a las variables proposicionales en la Lógica de Primer Orden también existe dicho concepto aunque con un mayor grado de complejidad.

Sea L un lenguaje de primer orden. Una *L-estructura* (o *interpretación*) corresponde a un par $\mathcal{M} = (M, I)$ donde M es un conjunto no vacío llamado *universo* (o dominio) de la L-estructura e I es una aplicación tal que:

- Aporta una interpretación para cada constante, dicha interpretación se denota por $c^{\mathcal{M}}$, de forma que $\forall c \in L : c^{\mathcal{M}} \in M$. Esto es, a cada constante le asigna uno y sólo uno de los elementos del universo.

- Aporta una interpretación para cada uno de los símbolos de función en L, tal que si f corresponde a un símbolo de función n-aria ($n > 0$), entonces $f^{\mathcal{M}} : M^n \longrightarrow M$.
- Aporta una interpretación booleana para cada uno de los símbolos de predicado en L, tal que si P corresponde a un símbolo de predicado n-ario, entonces $P^{\mathcal{M}} : M^n \longrightarrow \{0, 1\}$. O equivalentemente se puede dar un único conjunto de forma que las n-tuplas pertenecientes al mismo serían consideradas como verdaderas y el resto como falsas. En caso de trabajar con un LPO con igualdad: $=^{\mathcal{M}}$: $\{(a, a) : a \in M\}$

Vamos a exponer algunos ejemplos de L-estructuras. Tomemos el lenguaje

$$LA = \{\underbrace{\mathbf{0}, \mathbf{1}}_{const.}, \overbrace{<^{(2)}, =^{(2)}}^{pred.}, \underbrace{^{(2)}, +^{(2)}}_{func.}\}$$

Una L-estructura podría ser:

$$\mathcal{M}_1 = \begin{cases} M_1 = \mathbb{N} \\ \mathbf{0}^{M_1} = 0; \mathbf{1}^{M_1} = 1; \\ +^{M_1} : \mathbb{N}^2 \longrightarrow \mathbb{N},\ +^{M_1}(n_1, n_2) = n_1 + n_2 \\ {}^{M_1} : \mathbb{N}^2 \longrightarrow \mathbb{N},\ {}^{M_1}(n_1, n_2) = n_1 n_2 \\ <^{M_1}: \mathbb{N}^2 \longrightarrow \{0, 1\},\ <^{M_1} = \{(i, j) : (i, j) \in \mathbb{N}^2 \wedge (i < j)\} \\ =^{M_1}: \mathbb{N}^2 \longrightarrow \{0, 1\}, =^{M_1} = \{(i, i) : i \in \mathbb{N}\} \end{cases}$$

Pero otra podría corresponder a:

$$\mathcal{M}_2 = \begin{cases} M_2 = \mathbb{Q} \\ \mathbf{0}^{M_2} = \frac{1}{2}; \mathbf{1}^{M_2} = 2; \\ +^{M_2} : \mathbb{Q}^2 \longrightarrow \mathbb{Q},\ +^{M_2}(n_1, n_2) = |n_1 - n_2| \\ {}^{M_2} : \mathbb{Q}^2 \longrightarrow \mathbb{Q},\ {}^{M_2}(n_1, n_2) = n_1 \\ <^{M_2}: \mathbb{Q}^2 \longrightarrow \{0, 1\},\ <^{M_2} = \{(i, j) : (i, j) \in \mathbb{Q}^2 \wedge (ij > 0)\} \\ =^{M_2}: \mathbb{Q}^2 \longrightarrow \{0, 1\}, =^{M_2} = \{(i, i) : i \in \mathbb{N}\} \end{cases}$$

Interpretación de términos y fórmulas

Al igual que presentamos en en caso LP, hemos de establecer qué fórmulas consideramos verdaderas respecto de una posible interpretación y qué fórmulas consideramos falsas. Para ello hemos de dar la interpretación de las fórmulas, y para poder hacerlo hemos exponer previamente la interpretación de los términos.

Dada una L-estructura $\mathcal{M}$ a cada término término t de L, **sin variables** , le corresponde un objeto o elemento de M, que viene dado por la interpretación de la L-estructura ($t^{\mathcal{M}}$), de forma que:

$$t^{\mathcal{M}} = \begin{cases} c^{\mathcal{M}} & \text{si } t \equiv c \,(\text{con } c = cte) \\ f^{\mathcal{M}}(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}}) & \text{si } t \equiv f(t_1, \ldots, t_n) \end{cases}$$

Ahora, la interpretación de las fórmulas se expresa de forma que dada una L-estructura $\mathcal{M}$ decimos que una fórmula **cerrada**, F, se satisface en $\mathcal{M}$ (y se denota por $\mathcal{M} \models F$) si se da alguno de los siguientes supuestos:

1. $F \equiv P(t_1, \ldots, t_n)$ y además $P^{\mathcal{M}}(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}}) = True$ o, equivalentemente, $(t_1^{\mathcal{M}}, \ldots, t_n^{\mathcal{M}}) \in P^{\mathcal{M}}$
2. $F \equiv \neg F_1$ y además $\mathcal{M} \not\models F_1$
3. $F \equiv F_1 \vee F_2$ y se tiene que $\mathcal{M} \models F_1$ o $\mathcal{M} \models F_2$ (análogo para el resto de conectivas).
4. $F \equiv \exists x\, F_1$ y además hay algún elemento $e \in M$ (representado por el término e) tal que $\mathcal{M} \models F_1\{x/e\}$.

5. $F \equiv \forall x\, F_1$ y además todo elemento $e \in M$ (representado por el término e) verifica $\mathcal{M} \models F_1\{x/e\}$.

En caso de que F no sea cerrada se tiene, por definición:

$$(\mathcal{M} \models F) \Leftrightarrow (\mathcal{M} \models \forall x_1, \ldots, x_n F(x_1, \ldots, x_n))$$

Satisfactibilidad, validez, consecuencia lógica

Igual que veíamos que una fórmula en LP es satisfactible si existía al menos un modelo, en LPO se dice que una fórmula es *satisfactible* si existe, al menos, un modelo para dicha fórmula, esto es si existe una L-estructura en la que se satisface ($\mathcal{M} \models F$) y se dice *lógicamente válida* si para toda L-estructura ($\mathcal{M}$) se tiene $\mathcal{M} \models F$ (por ejemplo, $\forall x P(x) \vee \exists x \neg P(x)$). De forma análoga un conjunto de fórmulas Σ es *consistente* si existe una L-estructura tal que toda fórmula de Σ se satisface en $\mathcal{M}$.

Por último, de igual forma que veíamos el concepto de consecuencia lógica en LP, en LPO se tiene que una fórmula F es *consecuencia lógica* de un conjunto de fórmulas Σ (y se denota por $\Sigma \models F$) si se tiene que para toda L-estructura $\mathcal{M}$, $\mathcal{M} \models \Sigma \Rightarrow \mathcal{M} \models F$.

2.3.3. El problema de la indecidibilidad y Algoritmos de decisión parciales

La tesis de Church-Turing: La indecidibilidad

Es fácil advertir que los Lenguajes de Primer Orden son mucho más expresivos que los proposicionales, el precio que hay que pagar a cambio es considerable. Si bien en LP los problemas de consistencia y consecuencia lógica eran decidibles, aunque presuntamente intratables; estos mismos problema en el ámbito de la Lógica de Primer Orden no son ni siquiera decidibles, tal y como demostraron A. Church y A. Turing en 1936, esto es no existen algoritmos (procedimientos mecánicos) que permitan determinar, de forma completa, la consistencia de un conjunto de fórmulas, o equivalentemente no es posible decidir si una fórmula es consecuencia lógica de un conjunto, en los Lenguajes de Primer Orden.

Sin embargo, o equivalentemente el problema de la validez o la inconsistencia, sí son semidecidibles, esto es que existen algoritmos tales que son capaces de responder afirmativamente cuando el problema se satisface pero sin embargo no es capaz de responder negativamente en caso de no satisfacerse.

Precisamente dedicaremos este apartado al estudio de distintos algoritmos de decisión parciales.

Tableros Semánticos

Ya presentamos la versión de proposicional de este algoritmo. Recuérdese que se basaba en el método de demostración por *Reducción al Absurdo*, lo cual es especialmente interesante en el caso de LPO, dado que la inconsistencia de un conjunto sí es parcialmente decidible. El mecanismo de operación del mismo es similar al propuesto para LP incorporando, a las reglas descritas para LP, dos nuevas reglas que especifiquen la expansión del árbol a partir de fórmulas definidas a partir de cuantificación universal (regla γ) y a partir de cuantificación existencial (regla δ).

- Reglas de reducción

Recuérdese que en el caso LP se tenían tres reglas básicas: *Regla* α(conjuntiva), *Regla* β(disyuntiva), *Regla* dN(idempotente). Como hemos señalado previamente a estas reglas se incorporan dos nuevas reglas:

- *Regla* γ. Especifica el modo de operación para la expansión de fórmulas de comportamiento universal (que incluyen a fórmulas universalmente cuantificadas y a la negación de fórmulas existencialmente cuantificadas.) Debido al significado que posee el cuantificador universal, la forma de reducir esta fórmula, $F \equiv \forall x G$, a otras fórmulas más sencillas pasa por considerar todas

las posibles componentes que resultan de sustituir en G (una fórmula) la variable cuantificada universalmente (en el ejemplo, x) por cualquier término cerrado del lenguaje ($G\{x/t\}$), de forma que la satisfactibilidad de la fórmula original se reduce a la satisfactibilidad de todas estas posibles componentes.

- *Regla* δ. Especifica el modo de operación para la expansión de fórmulas de comportamiento existencial (que incluyen a fórmulas existencialmente cuantificadas y a la negación de fórmulas universalmente cuantificadas.) Debido al significado que posee el cuantificador existencial, la forma de reducir esta fórmula, $F \equiv \exists x G$, a otras fórmulas más sencillas pasa por considerar un objeto del mundo, representado por medio de una nueva constante, a, que verifica G, (Gx/a), de forma que la satisfactibilidad de la fórmula original se reduce a la satisfactibilidad de esta componente usando una nueva constante.

Al igual que en el caso LP, es común representar la sucesión de fórmulas deducidas como un árbol, como el mostrado en la *figura 2.5*. De forma que cada nodo corresponde a un a una fórmula (del conjunto original o de las que se han deducido) y cada arista indica la regla aplicada para la deducción del nodo destino y sobre qué formula se ha aplicado dicha regla; de forma que cada rama del árbol establece un conjunto independiente, encontrándose una inconsistencia si en la rama aparecen dos literales complementarios (extensible naturalmente a fórmulas complementarias).

Ahora nótese que asociado al conjunto se posee un conjunto de objetos (elementos del mundo) que inicialmente está formado por los términos cerrados que aparecen en las fórmulas y al que se irán añadiendo más elementos en la expansión del árbol con fórmulas δ y γ. Nótese que es posible que haya que considerar un conjunto infinito de elementos en el mundo (basta tener un símbolo de función). Esto hace que dado que la regla γ es aplicable con cada elemento del mundo, entonces se pueden tener árboles infinitos. Es justamente esto lo que implica el problema de la indecidibilidad, planteado al comienzo del apartado.

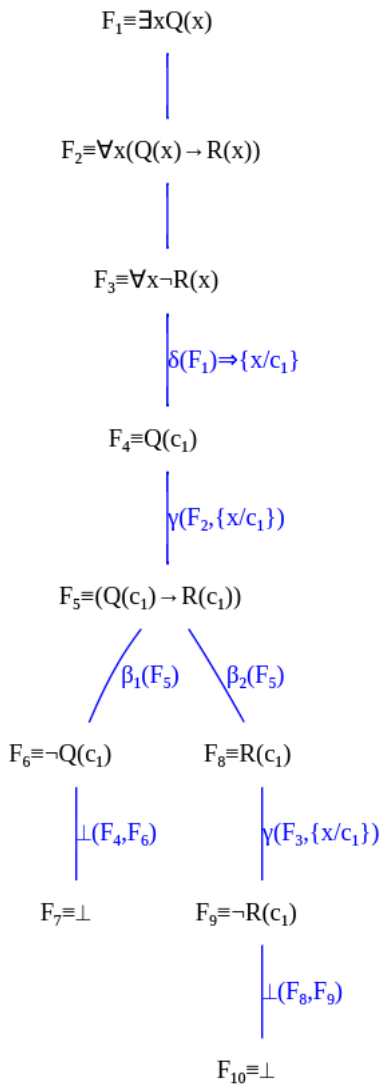


Figura 2.5: Tablero Semántico
Fuente propia: generado con LogicUS

Es preciso resaltar el carácter de refutación del procedimiento planteado, de forma que no siempre es posible determinar la satisfactibilidad de una fórmula LPO. Sin embargo, al igual que en el caso LP, en caso de poder determinar que una rama es abierta, dicha rama establecería un posible modelo para la fórmula, o al menos, daría algunas propiedades que ha de cumplir el mismo.

En el *algoritmo 2.6* se encuentra detallado el pseudocódigo del algoritmo para la construcción del tablero. Dicho algoritmo (dado el teorema de Correción y Completitud asociado al mismo) garantiza la (semi)decidibilidad de la inconsistencia del conjunto. Ha de tenerse presente que en caso de que el conjunto sea consistente entonces la parada del mismo no está garantizada.

El algoritmo de construcción que se presenta, similar al del caso LP, añadiendo las consideraciones necesarias para las dos reglas nuevas (γ y δ). Sin embargo, como en el caso de LPO los conjuntos

de fórmulas pueden crecer mucho más (sobre todo, debido a las fórmulas de tipo γ), usaremos una convención distinta a la hora de representar los árboles, que consistirá en ir añadiendo las nuevas fórmulas como nodos nuevos, de forma que en cada nodo el conjunto que se maneja corresponde a todos los nodos que forman parte de dicha rama (desde él hasta la raíz).

Para diferenciar qué fórmulas se pueden aplicar y cuáles no (por ejemplo, las fórmulas α, β y δ, aunque sigan apareciendo en los nodos, no se pueden usar más de una vez, pero las fórmulas γ se pueden usar tantas veces como términos cerrados se puedan construir) se usará una marca adicional que indica si una fórmula es usable o no.

Algoritmo 2.6: Tablero Semántico para conjuntos de fórmulas en Primer Orden

Tablero Semático (U)**:**

Input : U: Un conjunto de fórmulas LPO cerradas y bien formadas
Output: T: Un árbol que representa la aplicación de las reglas y las fórmulas y conjuntos que se han explorado

1. Hacer $T = N_1 - N_2 - \cdots - N_k$ un árbol lineal en el que los nodos están etiquetados con las fórmulas del conjunto original, tomando como raíz la primera y añadiendo sucesivamente las demás como hijos de la anterior.
2. Mientras T tenga ramas no marcadas seleccionar una de ellas T_{br} y hacer:
   - 2.1 Si T_{br} contiene un par de fórmulas complementarias (o la fórmula insatisfactible) marcar la rama *cerrada* ($\times$)
   - 2.2 Sino, si existe una fórmula F, usable tal que:
     - 2.2.1 F es de tipo dN, entonces extender T_{br} con $dN(F)$ y marcar F como *no usable*.
     - 2.2.2 F es de tipo α, entonces extender T_{br} con $\alpha_1(F)$ y $\alpha_2(F)$ y marcar F como *no usable*.
     - 2.2.3 F es de tipo β, entonces bifurcar T_{br} con dos ramas T_{br1} T_{br2} extendiéndolas con $\beta_1(F)$ y $\beta_2(F)$, respectivamente; y marcar F como *no usable*.
     - 2.2.4 F es de tipo δ, entonces extender T_{br} con $\delta_c(F)$ introduciendo una nueva constante, c; y marcar F como *no usable*.
     - 2.2.5 F es de tipo γ, entonces extender T_{br} con $\gamma_t(F)$, siendo t un término cerrado no utilizado previamente en F.
   - 2.3 Si no, marcar la rama *abierta* ($\circ$)

end

Formas normales

Al igual que hemos presentado la extensión de Tableros Semánticos de LP a LPO, vamos a presentar la extensión de las formas normales y también de un mecanismos de reducción de LPO a LP, en la llamada *Extensión de Herbrand*, la cual hemos mencionado previamente, como elemento fundamental en la demostración de la semidecidibilidad del problema de la consecuencia lógica (y sus equivalentes); aunque más allá de ello, el objetivo corresponde a tratar hasta qué punto podemos trasladar los algoritmos que se desarrollan para LP al contexto de Primer Orden. En este caso, no solo tendremos que tratar la aparición de cuantificadores para poder manipular las fórmulas de la manera adecuada, sino que también tendremos que ver de qué forma se puede extender el lenguaje (si es necesario) para poder tratar las fórmulas como si fueran abiertas y cerradas (para que se parezcan a fórmulas proposicionales, sin cuantificadores, y sin variables libres). Para ello, introduciremos dos nuevas formas (*Prenex* y *Skolem*), que servirán como modo de preprocesado de las fórmulas LPO para poder aplicar entonces las Formas Normales FNC y FND similares a las vistas en el caso proposicional, y veremos cómo los trabajos del matemático Jaques Herbrand nos permitirán en gran medida reducir la complejidad de las

interpretaciones LPO a modelos finitos parecidos a los que se tratan en LP.

Antes de nada, como ocurría con las transformaciones vistas para LP, las transformaciones que abordaremos durante este punto están basadas en el concepto de equivalencia, de tal manera que se establece que dos fórmulas de un Lenguaje de Primer Orden, F y G, son equivalentes, $F \equiv G$ si y sólo si $F \leftrightarrow G$ es lógicamente válida.

Conviene notar que en LPO, si ambas fórmulas son cerradas entonces se tiene la misma definición que en el caso LP, esto es, F y G son equivalentes si y sólo si poseen, exactamente, los mismos modelos. Para las fórmulas no cerradas la equivalencia queda como: $F \equiv G \iff \models \forall \boldsymbol{x}(F(\boldsymbol{x}) \leftrightarrow G(\boldsymbol{x}))$.

Visto el concepto de equivalencia en LPO, podemos pasar a describir el formato y el cálculo de formas *Prenex* asociadas a una fórmula LPO. Dicho cálculo es siempre realizable, es decir, para cualquier fórmula F existe una fórmula en forma Prenex F' tal que $F \equiv F'$.

Forma Prenex:Una fórmula se dice que está en forma Prenex si en ella se tiene que ninguna conectiva es aplicada sobre subfórmulas cuantificadas. Esto es,

$$F_{prenex} : Q_1 x_1 Q_2 x_2 \cdots Q_k x_k G(x_1, x_2, \ldots, x_k)$$

tal que $Q_k \in \{\forall, \exists\}$ y $G(x_1, x_2, \ldots, x_k)$ es abierta.

El cálculo de dichas formas se pude hacer a través de los pasos:

1. Eliminación de equivalencias
$$F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F)$$
2. Interiorización de las negaciones. Para ello, además de la leyes de De Morgan se tiene:
$$\neg(\forall x F(x)) = \exists x(\neg F(x))$$
$$\neg(\exists x F(x)) = \forall x(\neg F(x))$$
3. Extracción de los cuantificadores.

Para la extracción de los cuantificadores se definen las siguientes reglas. Nótese que, dado que se han renombrado previamente las fórmulas, en todos los casos $x \notin VL(H)$:

- $(\forall|\exists)xF_1(x)(\wedge|\vee)H \equiv (\forall|\exists)x(F_1(x)(\wedge|\vee)H)$
- $(\forall|\exists)xF_1(x) \rightarrow H \equiv (\exists|\forall)x(F_1(x) \rightarrow H)$
- $H \rightarrow (\forall|\exists)xF_1(x) \equiv (\forall|\exists)x(H \rightarrow F_1(x))$

De forma que ahora se pueden ir extrayendo los cuantificadores desde las subfórmulas más básicas a hasta la fórmula completa, tal y como se muestra en *figura 2.6*. Nótese que se podrían haber extraído primero los universales y después los existenciales, pero veremos a continuación, cuando hablemos de la skolemización de las fórmulas la relevancia que tiene el orden de extracción de los cuantificadores.
Tras haber obtenido una forma Prenex de la fórmula, el siguiente paso para situarnos más cerca del ámbito LP es la eliminación de los cuantificadores, mediante el proceso conocido como Skolemización. La *Forma de Skolem* trata de restringir la complejidad de las fórmulas en LPO sin perder la expresividad de las mismas.

Forma Skolem:Una fórmula cerrada se dice que está en forma de Skolem si está en forma Prenex y además carece de cuantificadores existenciales. Esto es,

$$F_{Skolem} : \forall x_1 \forall x_2 \cdots \forall x_k G(x_1, x_2, \ldots, x_k)$$

Para transformar una fórmula (Prenex) en una fórmula skolemizada, se hace uso de las funciones de Skolem de forma que cada existencial de la cabeza es eliminado aplicando sobre la fórmula una

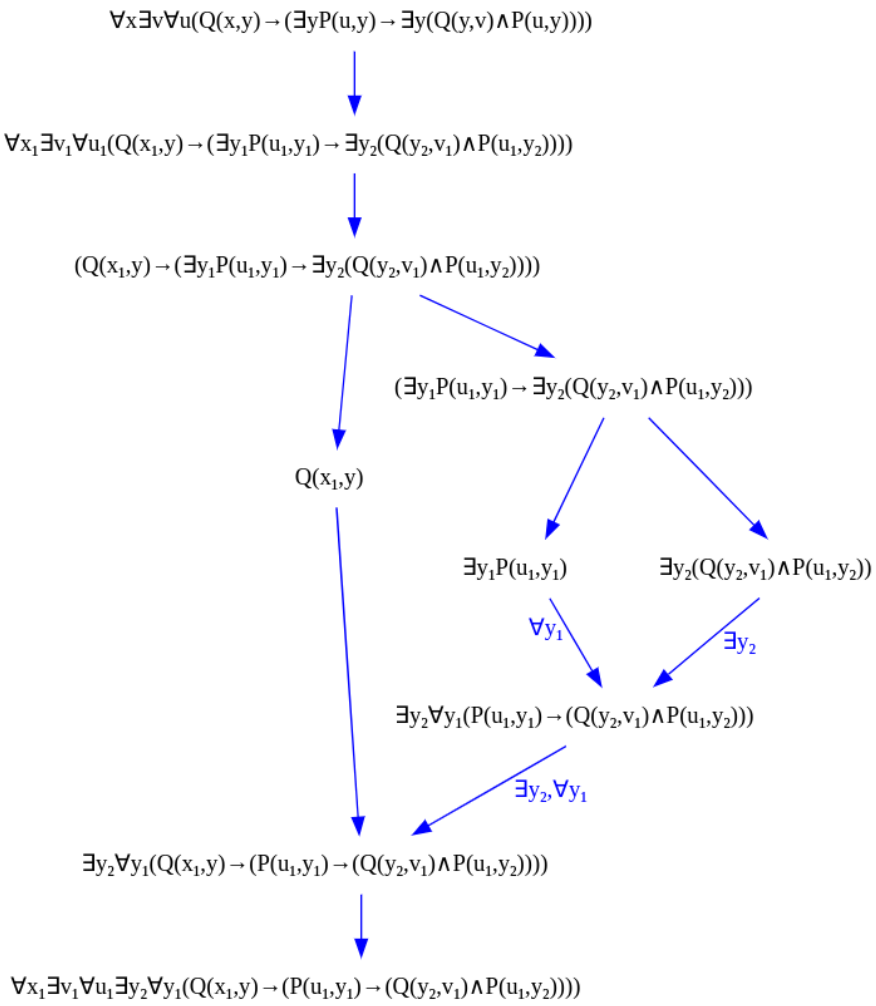


Figura 2.6: Extracción de cuantificadores
Fuente propia: generado con LogicUS

sustitución en la que la variable del cuantificador universal, x_i es sustituida por una función dependiente de las variables de los cuantificadores universales de orden superior en la fórmula (que aparecen antes de él). Si dicha función es de aridad 0, se denomina constante de Skolem, y en otro caso se denomina, de manera general, función de Skolem.

Nótese que el aumento de cuantificadores universales delante de los existenciales hace aumentar la aridad de las funciones, lo que influye, de manera directa, en la complejidad de las fórmulas que habrán de ser tratadas; y por ende en el proceso de decisión del mismo.

Ahora bien, hemos de destacar que en este paso no se mantiene la equivalencia entre las fórmulas, en cambio, sí se mantiene la equiconsistencia. Esto es, la fórmula original es satisfactible sí y solo sí lo es la fórmula obtenida en forma de Skolem. Además todos los resultados y métodos aquí explicados son extensibles de manera directa a los conjuntos, sin más que aplicar de forma independiente a cada fórmula del conjunto el proceso de skolemización.

Como cuestión final, es corriente expresar las fórmulas en forma de Skolem sin los cuantificadores (como fórmulas abiertas), sobreentendiendo que todas ellas están cuantificadas universalmente. De forma que, siguiendo con el ejemplo anterior, la forma de Skolem correspondería a:

$$P(x_1, a) \rightarrow (P(x, \boldsymbol{c_{sk}}) \rightarrow P(x_2, a))$$

Esta fórmula parece ya, casi proposicional. De hecho, desde este punto es posible definir y calcular la formas normales abordadas en el ámbito de la lógica proposicional: FNN, FNC, FND, Clausal, tratando los predicados como átomos y operando de forma exactamente equivalente a como se mostró en el ámbito LP.

Finalmente vamos a mostrar una vía de reducción de la consistencia de un conjunto de fórmulas en un lenguaje de primer orden a la de un conjunto de fórmulas proposicionales usando los trabajos que, en esta dirección, realizó el matemático Jaques Herdbrand.

Si bien el cálculo de formas de Skolem proporciona un método para la reducción de la consistencia de un conjunto de fórmulas Γ de un lenguaje L en un conjunto de fórmulas abiertas Σ de un lenguaje L', los trabajos de Herbrand proporcionan una traslación, casi total, del ámbito LP al ámbito LP. De manera que, la *extensión de Herbrand* de Σ, notada por $EH(\Sigma)$ corresponde a un conjunto formado por todas las fórmulas cerradas que pueden obtenerse (de todas las formas posibles) las variables de las fórmulas por términos cerrados del lenguaje.

En este sentido, los términos cerrados del lenguaje representan todos aquellos objetos del mundo que el lenguaje es capaz de nombrar directamente, usando los símbolos del lenguaje. Sin embargo, conviene recordar que el conjunto de elementos de un lenguaje es, habitualmente, un conjunto infinito y por ende lo será también la extensión. Esto hace que la verificación de la satisfactibilidad de una extensión de Herbrand no sea siempre un proceso finito. Ahora bien, esto no es extraño, recordemos que el problema de la consistencia es indecidible, en cambio, el de la consistencia sí es parcialmente decidible, y ello es demostrable, como ya indicamos, gracias a la extensión y al teorema de Compacidad de Herbrand, que establece que $EH(\Sigma)$ es inconsistente si y sólo si existe un subconjunto finito de la misma que también lo es. De esta forma es fácil notar que el problema de la inconsistencia es parcialmente decidible sin más que considerar los siguientes pasos:

Algoritmo 2.7: Algoritmo de semidecidibilidad de la inconsistencia

Tablero Semántico (U)**:**

Input : U: Un conjunto de fórmulas LPO: Γ
Output: SÍ si el conjunto es inconsistente, NO si el conjunto es consistente y su EH finita, $\uparrow$ si el conjunto es consistente y su EH infinita

1. Σ el conjunto de formas de Skolem de Γ.
2. Hacer $EH(\Sigma) = \{G_1, G_2, \dots\}$ la extensión de Herbrand de Σ
3. Para $i = 1, 2, \dots, |EH(\Sigma)|$ hacer:
   3.1 Si el conjunto $\{G_1, G_2, \dots, G_i\}$ es inconsistente (por ejemplo mediante Resolución Proposicional) entonces *PARAR* y *DEVOLVER SÍ*
4. *PARAR* y *DEVOLVER NO*

end

De forma que dicho algoritmo establece una demostración de la decidibilidad parcial del problema de la inconsistencia (y sus equivalentes), tal y como comentamos al inicio de este punto.

Nótese la importancia de los conceptos y técnicas aportadas en este punto, que permiten establecer una reducibilidad (parcial) de la Lógica de Primer Orden en la Lógica Proposicional. Nótese además la gran relevancia de la forma Prenex y de Skolem, y de la extracción prioritaria de los existenciales, dado que el aumento de la aridad de las funciones de Skolem, ya que no solo podemos pasar de una extensión finita a una infinita, sino que además supone un aumento exponencial (intratable) de la cardinalidad de la extensión de Herbrand, derivado del problema combinatorio.

Unificación y Resolución en Primer Orden

Como ya se introdujo en el primer apartado, cuando presentamos la demostración de la decidibilidad parcial de la inconsistencia de un conjunto en Primer Orden, existe un mecanismo, haciendo uso de la Resolución Proposicional sobre la extensión de Herbrand. Este método, sin embargo, tiene el problema de que al tratar con conjuntos infinitos el camino de resolución proposicional puede hacerse altamente ineficiente ya que generamos muchas cláusulas "proposicionales"que no usamos en el proceso de alcanzar la cláusula vacía (en caso de ser refutable), teniendo en cuenta, además, que ya de por sí la insatisfactibilidad proposicional es un problema intratable.

Aunque una primera opción pasa por realizar una *resolución restringida*, generando solo aquellas cláusulas útiles, sin una estrategia de cómo elegir cuáles serán dichas cláusulas y con qué asignación de términos cerrados se obtienen, esta opción no asegura encontrar el camino a la solución (y no podemos tener la esperanza de obtener tal estrategia debido a la indecidibilidad de la LPO).

Una opción más interesante consiste en utilizar un recurso similar pero sin necesidad de llegar a hacer sustituciones de todas las variables libres por términos cerrados, resolución no restringida, que permite un mejor aprovechamiento de cálculos intermedios, ahorrando sustituciones completas que pueden simplificarse y reutilizarse si se hacen sustituciones más generales. Dichas sustituciones tienen dos propósitos fundamentales:

- Hacer que varios literales de una misma cláusula se conviertan en un mismo literal, con el fin de tener cláusulas más cortas, y por ende más cercanas a la cláusula vacía.
- Hacer que literales de distinto signo de dos cláusulas distintas se conviertan en literales complementarios, con el fin de ser usados para aplicar una resolvente entre ambas cláusulas.

Ambos casos se enmarcan de lo que se conoce como *unificadores*. En concreto un unificador es una sustitución tal que hace que los átomos (independientemente de su signo) sean equivalentes. Si entre dos literales dicha sustitución existe entonces se dice que son unificables. Nótese que para dos literales puede existir más de un unificador, de forma que esto da pie a definir el concepto de generalidad comparativa de dos unificadores. De forma que un unificador, τ es más general que otro, σ, si existe uno 'intermedio', ε que transforma el segundo en el primero, esto es $\sigma = \varepsilon \circ \tau$. Nótese que aplicar el segundo permite permite unificar las cláusulas, a la par que siguen quedando opciones de hacer otros cambios de variable posteriores y reutilizar las cláusulas obtenidas.

Esta generalidad comparativa permite, a su vez, definir el concepto de *Unificador de Máxima Generalidad* para un conjunto de cláusulas unificables como aquél tal que no existe otro que sea más general que él. Además se puede probar, que, en caso de tener un conjunto de cláusulas unificables, siempre existe un UMG para ellas. Un algoritmo para calcular este UMG consiste en ir identificando los términos que aparecen en las diversas componentes de los literales que se quieren unificar y componer (cuando sea posible) una sustitución que haga que esas identificaciones sean posibles. Formalizamos ese algoritmo en lo dispuesto en el *Algoritmo 2.8*.

Visto esto, se puede abordar la resolución no restringida para cláusulas de primer orden, consiguiendo los unificadores adecuados, el proceso de resolución no restringida permite obtener cadenas de resoluciones que, generalmente, son más cortas y eficientes que las obtenidas por resoluciones no restringidas, produciendo cláusulas intermedias que son más generales y reutilizables.

Con todo ello, podemos definir la regla de resolución para Lenguajes de Primer Orden. De forma que se establece que $\mathcal{R}$ es una resolvente de dos cláusulas C_1 y C_2 sin variables comunes (si fuera necesario se renombrarían) si se verifica:

- $C_1 = A \cup L = \{l_1, \dots l_m\}$ y $C_2 = B \cup L' = \{l'_1 \dots l'_m\}$ tal que el conjunto $\Gamma = \{a_1, \dots, a_m, a'1, \dots a'_m\}$ es unificable (con a_i el átomo referente al literal l_i) bajo el UMG σ.
- $\mathcal{R} = (A \cup B)\{\sigma\}$

Algoritmo 2.8: Algoritmo de unificación de átomos

term2Unification (t_1, t_2)**:**

Input : t_1, t_2: Dos términos de un Lenguaje de Primer Orden
Output: Un UNM de t_1 y t_2 si son unificables. FALLO si no son unificables.

1. Si $t_1, t_2 \in VAR$ entonces DEVOLVER $\{t_1/t_2\}$
2. Si no, si $t_1 \in VAR$, entonces si t_1 no aparece en t_2 DEVOLVER $\{t_1/t_2\}$, en otro caso FALLO.
3. Si no, si $t_2 \in VAR$, entonces si t_2 no aparece en t_1 DEVOLVER $\{t_2/t_1\}$, en otro caso FALLO.
4. Si no, si $t_1 \equiv F(\boldsymbol{ts_1})$ y $t_1 \equiv F(\boldsymbol{ts_2})$ entonces DEVOLVER termsUnification($\boldsymbol{ts_1}$,$\boldsymbol{ts_2}$), en otro caso FALLO.

end

termsUnification (ts_1, ts_2)**:**

Input : t_1, t_2: Dos listas de términos de un Lenguaje de Primer Orden
Output: Un UNM de ts_1 y ts_2 si son unificables, (unificación por pares). FALLO si no son unificables.

1. Si $ts_1 = ts_2 = \emptyset$ entonces DEVOLVER $\{\}$
2. Si no, si $ts_1 = \emptyset$ o $ts_2 = \emptyset$ entonces FALLO.
3. Si no, sean t_1 y t_2 los primeros términos de cada una de las listas. Y ts'_1 y ts'_2 los restos de las listas. Entonces:
   3.1. Hacer $s' \leftarrow$ term2Unification(t_1, t_2)
   3.2. Si $s' = FALLO$ entonces FALLO.
   3.3. Si no, hacer $s'' \leftarrow$ termsUnification$(ts'_1\{s'\}, ts'_2\{s'\})$
   3.4. Si $s'' = FALLO$ entonces FALLO.
   3.5. Hacer s el resultado de componer las sustituciones. DEVOLVER s

end

atom2Unification (a_1, a_2)**:**

Input : a_1, a_2: Dos átomos de un Lenguaje de Primer Orden
Output: Un UNM de a_1 y a_2 si son unificables. FALLO si no son unificables.

1. Si $a_1 \equiv P(ts_1)$ y $a_2 \equiv P(ts_2)$ entonces DEVOLVER termsUnification(ts_1, ts_2).
2. Si no, FALLO.

end

Y en tal caso la regla de resolución correspondería a :

$$\frac{A \cup L\ ,\ B \cup L'}{(A \cup B)\{UMG(\{a_1, \ldots, a_m, a_1, \ldots a'_m\})\}}$$

Como ilustración de la misma considérense las cláusulas

$$C_1 = \{P(f(x)), \neg Q(z), P(z)\} \text{ y } C_2 = \{\neg P(x), R(g(x))\}$$

Entonces la regla de resolución podría aplicarse como:

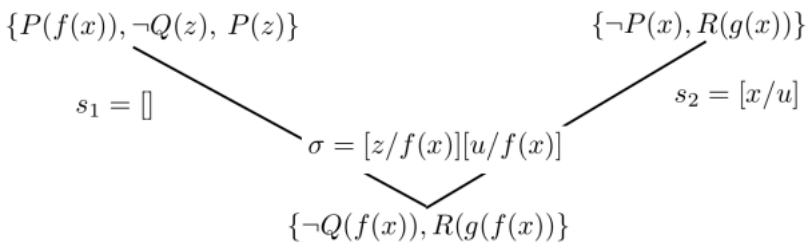


Figura 2.7: Aplicación de la regla de resolución
Fuente: Logic for Computer Scientists [?]

Además el Teorema de Resolución, permite asegurar que dado un conjunto de cláusulas de un Lenguaje de Primer Orden, C, dicho conjunto es inconsistente si y sólo si $C \vdash_{Res} \square$. En el *Algoritmo 2.9* presentamos un esquema algorítmico del procedimiento de resolución, que sintetiza los conceptos presentados hasta el momento.

Algoritmo 2.9: Algoritmo de resolución en Primer Orden

FOLResolution(C)**:**

Input : F: Un conjunto de fórmulas de un Lenguaje de Primer Orden
Output: SÍ sólo si el conjunto es insatisfactible

1. Hacer S el conjunto de cláusulas asociadas F
2. Mientras haya dos cláusulas $C_a, C_b \in G$ y una resolvente $\mathcal{R} \notin S$
   2.1. Hacer $S \leftarrow S \cup \mathcal{R}$
   2.2 Si $\square \in S$ entonces PARAR y DEVOLVER SÍ

end

2.3.4. Las limitaciones de los Lenguajes de Primer Orden

Si bien la Lógica de Primer Orden posee mucha más capacidad expresiva que la lógica proposicional, también posee ciertas limitaciones. Un ejemplo bien conocido de ellas es la *clausura transitiva* o *cierre transitivo* de una relación, que no es expresable en la Lógica de Primer Orden; u otras cuestiones como la veracidad parcial de predicados tampoco es expresable, por ello existen otras lógicas tales como la Lógica de Segundo Orden, que permite la cuantificación de predicados y funciones (aumentando el poder expresivo de la LPO sin necesidad de agregar nuevos símbolos) o la lógica difusa (fuzzy) que asigna grados de verdad a los predicados (entre 0 y 1) y permite el razonamiento aproximado.

Vistos los fundamentos formales de las Lógicas tratadas en este trabajo, pasaremos, en el siguiente capítulo a tratar las principales tecnologías utilizadas en el desarrollo de este trabajo, las razones de su elección, y la contribución que éstas hacen al proyecto desarrollado.

3 Tecnologías de desarrollo e integración

En este capítulo introduciremos y detallaremos las tecnologías utilizadas en el desarrollo del proyecto así como los métodos de integración de las mismas y la comunicación entre ellas. En concreto, presentaremos una visión general del principal lenguaje de desarrollo utilizado, Elm, así como de sus principales características, detallando aquellos aspectos que intervienen de forma decisiva tanto en la creación de los módulos de cálculo y representación (en adelante *core*) como de la parte gráfica.

Complementariamente al lenguaje anterior, presentaremos las tecnologías clásicas de desarrollo web (*HTML*, *CSS* y *JS*), justificando su uso y explicitando los detalles más importantes de los usos que se le han dado, haciendo un especial hincapié en la comunicación con Elm en una arquitectura cliente-servidor a través del uso de *puertos*.

Finalmente, se presentan algunas herramientas complementarias de otros proyectos que resultan de interés aplicativo de los materiales que se desarrollan en el proyecto. En concreto, trataremos con la herramienta *Literate Visualization*[2] de giCentre para la elaboración de documentación mediante el uso del lenguaje *Markdown* y la integración de fragmentos de código Elm (*chuncks*).

3.1. Lenguaje Elm

Elm [3] es un lenguaje de dominio específico enmarcado en los paradigmas funcional y reactivo, y enfocado al desarrollo front-end.

3.1.1. Motivación de Elm

Todo buen desarrollador debe tener en consideración la robustez, usabilidad, y mantenibilidad del código que genera, y además no sólo la resolución de los problemas actuales que han dirigido su diseño, sino también la capacidad de adaptación del mismo ante nuevos requisitos que puedan surgir en el futuro. Sin embargo, el cumplimiento de dichos requisitos no es fácil y no debido a la falta de herramientas que nos permitan la distribución de programas fácilmente mantenibles.

La familia de los *Meta Lenguajes*, *ML*, entre los que se encuentran lenguajes como Haskell o Ocaml son bien conocidos por su fuerte resistencia a la aparición de errores en tiempo de ejecución, ya que son capaces de detectar los errores antes de llevar el código a la fase de producción.

Sin embargo, y a pesar de los grandes beneficios de este tipo de lenguajes, ninguno de ellos se encuentra entre los más utilizados por la comunidad de desarrolladores de soluciones web, que prefieren sistemas como (*jQuery*, *React.js* y *Angular*) basados en *JavaScript* que no proporciona de tales beneficios.

Para justificar este hecho suelen indicarse causas como:

- Los lenguajes de la familia *ML* son más adecuados para crear sistemas de back-end.

- Los navegadores sólo están capacitados para ejecutar código JavaScript, dificultando la interacción con otros lenguajes.
- Los lenguajes *ML* poseen una curva de aprendizaje más alta.

En este sentido, lo ideal sería contar con un lenguaje que no sólo proporcionase la solidez inherente a los lenguajes de la familia *ML*, sino que también fuese fácilmente utilizable como JavaScript. Y es justamente en este punto en el que *Elm* destaca, tal y como representa el creador de Elm en la figura 3.1.

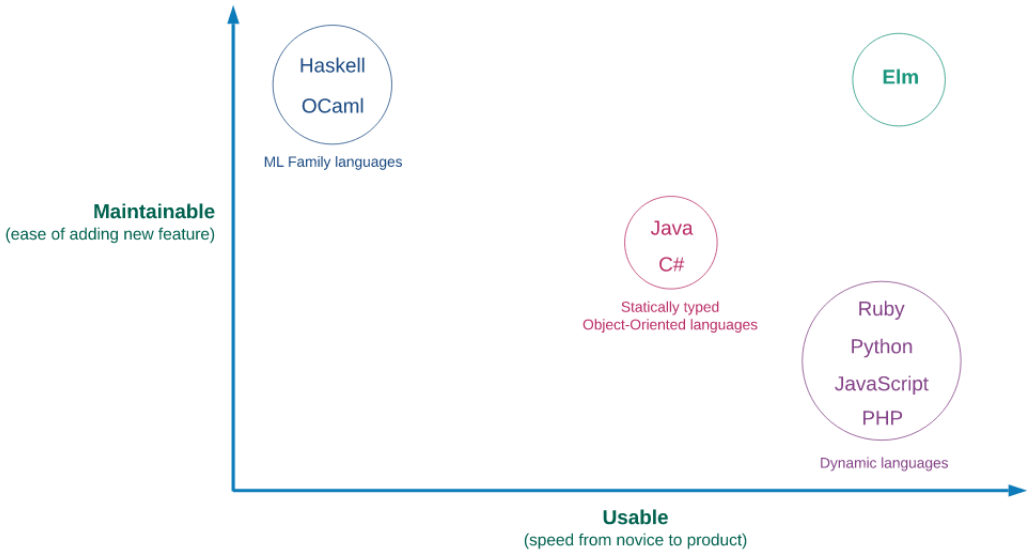


Figura 3.1: Mantenibilidad vs Usabilidad
Fuente: Evan Czaplicki - Let's be mainstream! User focused design in Elm - Curry On

Elm es considerado un lenguaje funcional y fuertemente tipado, marcado por los principios de la familia *ML*, de hecho está escrito en Haskell y son patentes las características heredadas de este lenguaje, entre las que se incluyen valores inmutables, funciones sin estado, inferencia de tipos, coincidencia de patrones, formateo automático de código y un gran verificador de tipos estáticos. Es gracias a ello por lo que la ausencia de errores en tiempo de ejecución está, prácticamente, asegurada. Además, propone una arquitectura novedosa en la que la organización del código facilita la gestión del flujo de datos entre los distintos componentes, favoreciendo la usabilidad y mantenibilidad del software desarrollado.

Complementariamente, y aunque la implementación inicial presentada por Evan Czaplicki en su tesis en el año 2012 [4] se enfocaba únicamente a HTML, CSS y JS, y estaba disponible únicamente para sistemas Linux, su gran usabilidad ha conllevado un gran desarrollo incluyendo otros elementos que también resultan de interés para el proyecto, tales como una consola *REPL* (*Read–Eval–Print Loop*), un gestor de paquetes (y publicación de los mismos), un sistema de detección e información de errores fuerte y detallado (habitualmente con mensajes específicos del posible error y posibles soluciones) e instaladores para macOS y Windows.

Por último, y para completar todos los beneficios expuestos previamente, la comunidad de desarrolladores trabaja en el desarrollo y publicación de nuevas librerías y también en editores de Elm como Litvis, dedicado a la integración de Markdown y Elm, o Ellie, que permite el desarrollo y guardado de módulos Elm online (incluyendo la inclusión de paquetes de la comunidad).

Comenzaremos en los siguientes apartados mostrando una panorámica del lenguaje Elm con el fin de facilitar la comprensión de las decisiones de diseño que se han llevado a cabo en las implementaciones realizadas.

3.1.2. Elm como lenguaje funcional

Los lenguajes de programación se han dividido clásicamente en dos grandes paradigmas: imperativo y funcional, aunque en la actualidad existen multitud de lenguajes que combinan elementos de distintos paradigmas y pueden encontrarse multitud de taxonomías y clasificaciones según distintos criterios. En la *tabla 3.1* se muestran las principales diferencias entre ambos paradigmas.

Programación Funcional	Programación Imperativa
Basado en la composición de funciones puras, evitando o minimizando efectos secundarios, datos compartidos y datos mutables.	Basado en la descripción de pasos que cambian el estado de la computadora.
Orientado a los resultados, esto es, los programas (funciones) deben ejecutarse u operar.	Orientado al proceso, esto es, describen cómo se ejecuta u opera el programa.
Basado en el uso de funciones	Basado en el uso de declaraciones que cambian el estado de un programa.
Requiere la representación explícita de las estructuras de datos que se utilizan.	Requiere que la estructura de datos se represente como cambios de estado.
Programas estructurados como sucesivas llamadas funcionales anidadas de forma recursiva. Programas en términos de funciones y estructuras matemáticas.	Programas estructurados como asignaciones sucesivas de valores a nombres de variables y controles de flujo (bucles, bloques condicionales,...). Programas como una serie de instrucciones o declaraciones que pueden modificar activamente la memoria.
Sus características incluyen código libre de errores, aumento del rendimiento, mejor encapsulación, aumento de la reutilización, aumento de la capacidad de prueba, baja importancia del orden de ejecución, modelo de programación sin estado, unidades de manipulación primaria, control de flujo primario, altas prestaciones para la programación concurrente, etc.	Sus características incluyen secuencia de declaraciones, manejo de estados, pueden tener algunos efectos secundarios, importancia del orden de ejecución, modelo de programación con estado, etc.

Tabla 3.1: Programación funcional vs Programación Imperativa

Son muchos los lenguajes puramente funcionales (Haskell, F#, Scala, etc.) y son aún más los que incorporan elementos propios de este tipo de lenguajes, formando parte de algunos de los más utilizados en la actualidad (Python, Javascript, etc.), e incluso algunos que en origen no fueron concebidos a tales efectos (Java, C++, etc.).

Una vez vista una pequeña perspectiva de la programación funcional, pasamos a comentar algunas de las principales características propias de Elm dentro de este paradigma:

- *Programación funcional pura*, que permite una mayor legibilidad y facilidad de interpretación, ya que establece exactamente una analogía entre la función matemática y el código utilizado. Dado que es usual que el tiempo que se dedica a leer y repasar código sea más de diez veces del tiempo que se dedica a escribirlo, la facilidad de la lectura e interpretación hace que el desarrollo se realice más rápido y con mayor seguridad.

- *Facilidad de testeo, debugging y ausencia de errores en tiempo de ejecución.* El tipado fuerte, los datos inmutables y la ausencia de efectos colaterales (gracias a las funciones puras) permiten elaborar test automáticos de pruebas (basadas en propiedades) fácilmente, permiten la detección

de errores de forma anticipada y con un gran detalle, y evitan la presencia de errores en tiempo de ejecución.

- *Código compacto y modular*. Dos características básicas de los lenguajes funcionales (y también de Elm) es la capacidad de 'hacer mucho escribiendo poco' y la capacidad de reutilización de funciones. En cuanto a la primera, características como el tratamiento de las funciones como variables (que pueden ser pasadas como argumentos), las funciones anónimas, la ausencia de jerarquías profundas en los tipos (habituales en las clases en OOP) o las funciones de orden superior, hacen que el código sea fácilmente interpretable, conciso y replicable. Además, la creación de módulos, cuyas funciones son exportables e importables, confieren al lenguaje una gran capacidad de organización y reutilización del código, complementándose con la capacidad de 'importación selectiva', que permite añadir solamente las funciones o tipos que sean necesarios sin requerir la importación de todo el módulo.
- *Facilidad para la programación concurrente/paralela*. Un programa funcional está listo para la concurrencia sin más modificaciones. Nunca tendrá que preocuparse por los puntos muertos y las condiciones de carrera porque no necesita usar bloqueos ya que, como se ha expuesto, carece de estados y todas las instancias son independientes, por lo que ningún dato en un programa funcional es modificado dos veces por el mismo hilo, y mucho menos por dos hilos diferentes. Esto significa que puede agregar subprocesos fácilmente sin tener que pensar en los problemas convencionales que afectan a las aplicaciones de concurrencia.

3.1.3. Elm como lenguaje de desarrollo Web

Todas las características anteriormente mencionadas hacen de un lenguaje funcional la herramienta idónea para trasladar los métodos y conceptos presentados en el capítulo 2 sobre el formalismo lógico, aportándonos además la capacidad de trabajar con conjuntos y procesos no finitos (especialmente beneficioso para LPO), aunque se truncarán los procesos según las necesidades de los usuarios. Además, de forma complementaria, Elm constituye un framework de desarrollo Web (front-end) y a estos efectos proporciona un compilador que permite generar archivos HTML + CSS + JS directamente del código escrito en Elm, además de promover el uso de la *Elm Architecture* que aporta la capacidad organizativa necesaria para la mantenibilidad del código.

Por el momento nos centraremos en la presentación del *Compilador de Elm* aunque volveremos a tratar con la *Elm Architecture* cuando tratemos el desarrollo de aplicaciones en apartados sucesivos.

Compilador de Elm

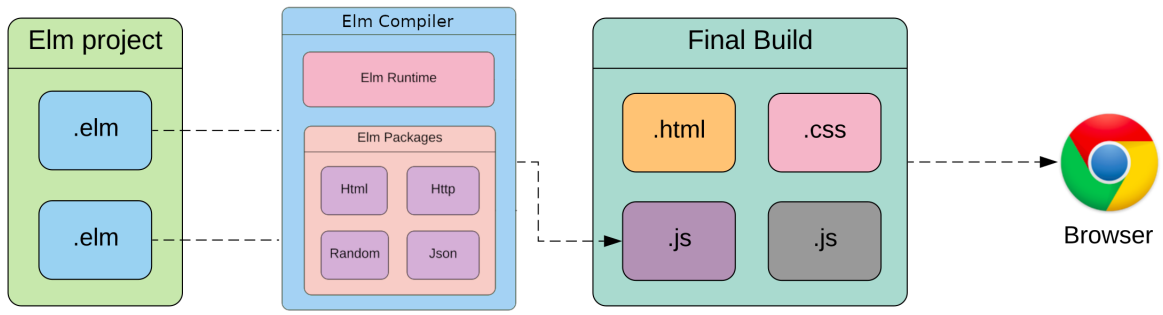


Figura 3.2: Compilador de Elm a JavaScript
Fuente: creada a partir de Beginning Elm - Elm Compiler

Como hemos señalado, los módulos interactivos creados en Elm son compilables directamente a JavaScript. La figura *figura 3.2* muestra un diagrama con el proceso de compilación de una aplicación

en Elm. Obsérvese que partiendo de una serie de módulos desarrollados en Elm llamando al compilador mediante la instrucción `elm make`, éste genera un sólo archivo en JavaScript. Ahora bien, una de las ventajas de Elm es su capacidad de integración con otros proyectos, permitiendo desarrollar sola una parte del proyecto en Elm o hacerlo en su totalidad, y facilitando la comunicación entre sus partes.

Sin embargo, los navegadores trabajan únicamente con JavaScript, por ello todo lo que es utilizado como parte del lenguaje elm ha de ser compilado también a JavaScript para que éste pueda ser correctamente interpretado en tiempo de ejecución, incluyendo tanto las librerías propias de Elm como los módulos (básicos y de terceros) utilizados. Para entender y poder operar no es necesario conocer cómo Elm lleva a cabo esta tarea, por lo que, para no añadir una complejidad innecesaria, no mostraremos las tareas de bajo nivel involucradas en el proceso de compilación.

Hasta ahora hemos abordado los aspectos más generales del lenguaje, lo que es suficiente para poder apreciar que parece haber dos partes bien diferenciadas, una dedicada al trabajo operativo, y otra a la parte interactiva. En efecto, Elm permite dos tareas complementarias, por una parte el desarrollo y publicación de paquetes y, por otra, el desarrollo de front-end para ser compilado a JavaScript. En la siguiente sección incidiremos precisamente en estos dos aspectos, ya que se hará uso de ambas partes durante el desarrollo del proyecto.

3.1.4. Creación y desarrollo de proyectos en Elm.

Hemos visto que en los aspectos generales del lenguaje aparecen como actores fundamentales los *paquetes* y las aplicaciones. Veamos la diferencia entre ellos:

- **Paquetes**: Los paquetes son básicamente librerías operativas que establecen tipos y funciones para operar con elementos o crear estructuras de datos específicas. Corresponden al desarrollo back-end de las aplicaciones, pero además pueden ser publicados para que todo desarrollador de Elm haga uso de los mismos. Elm cuenta con un repositorio que ofrece todos los paquetes públicos, *Elm Packages*. Daremos más detalles en futuros apartados.
- **Aplicaciones**. Corresponden a la interfaz web interactiva de un proyecto software, esto es, al desarrollo front-end del proyecto. En ella se hará uso de los paquetes y también se establecerán la estructura de la interfaz, los mecanismos de interacción, etc. Haciendo uso de una estructura Modelo-Vista-Controlador, que trataremos en sucesivas secciones.

Antes de iniciar un proyecto es necesario disponer de algunos componentes instalados. De forma somera, los principales recursos con que se debe contar son:

- **Elm**. Se han de instalar los binarios de Elm. Para ello, se dispone de los instaladores e instrucciones necesarias disponibles tanto para sistemas Windows, como Linux y MAC.
- **Node.js**. Es un entorno de ejecución de JavaScript para crear principalmente aplicaciones del lado del servidor, sin embargo resultará de gran utilidad para algunas herramientas, como la consola REPL de Elm o el formato de la documentación del código, entre otras.

Una vez instalados, para crear un proyecto basta con ir al directorio en el que se vaya a desarrollar el proyecto y en una terminal ejecutar `elm init`. Esto creará el archivo de configuración del proyecto *elm.json* y la carpeta raíz donde se alojarán los módulos que se desarrollen, *src/*.

El archivo de configuración, cuyo contenido inicial se muestra en el *código 3.1*, se crea inicializado para el desarrollo de una aplicación. Por lo que, si lo que se pretende es desarrollar un paquete, hemos de ser nosotros mismos los que hagamos los cambios necesarios. Veamos las diferencias entre los archivos de configuración.

Archivo *elm.json* para aplicaciones

El archivo JSON creado al iniciar un proyecto contiene algunos parámetros que establecen la configuración de la aplicación:

- `type`: Establece el tipo de proyecto. Para el desarrollo de aplicaciones debe establecerse como `"application"`.
- `source-directories`: Un lista de los directorios que albergan los módulos de la aplicación. Normalmente sólo es utilizado el directorio *src/*.
- `elm-version`: La versión exacta de Elm con la que se construye. La actual corresponde a la 0.19.1.
- `dependencies`: Una lista de todos los paquetes de que dependen los módulos del proyecto (`"direct"`). A su vez estos módulos pueden tener sus propias dependencias que son incorporadas a las dependencias como dependencias indirectas `"indirect"`.
- `test-dependencies`: Todos los paquetes implicados en los test.

```
{
    "type": "application",
    "source-directories": [
        "src"
    ],
    "elm-version": "0.19.1",
    "dependencies": {
        "direct": {
            "elm/browser": "1.0.2",
            "elm/core": "1.0.5",
            "elm/html": "1.0.0"
        },
        "indirect": {
            "elm/json": "1.1.3",
            "elm/time": "1.0.0",
            "elm/url": "1.0.0",
            "elm/virtual-dom": "1.0.2"
        }
    },
    "test-dependencies": {
        "direct": {},
        "indirect": {}
    }
}
```

Código 3.1: Archivo elm.json para aplicaciones

Archivo *elm.json* para paquetes

En caso de que vayamos a desarrollar un paquete hemos de modificar expresamente el archivo configurando los parámetros adecuados. En la práctica este cambio se suele hacer al final, ya que para el desarrollo de los módulos del paquete resulta irrelevante utilizar uno u otro. Vamos a pasar a detallar los campos de que consta el archivo de configuración para los paquetes:

- `type`: Establece el tipo de proyecto. Para el desarrollo de paquetes debe establecerse como `"package"`.
- `name`: El nombre del repositorio de GitHub que almacenará el código del módulo.
- `summary`: Un resumen de a lo sumo 80 caracteres del objetivo del módulo que aparecerá en la descripción del paquete en el repositorio de paquetes de Elm.
- `license`: Un código SPDX aprobado por OSI como `"BSD-3-Clause"` o `"MIT"`, que establece las condiciones bajo las que se distribuirá el paquete.
- `version`: La versión del paquete. Se inicia siempre con el valor 1.0.0, quedando la versión 0.y.z reservada para acciones de desarrollo. Elm actualizará dicho campo utilizando el versionado semántico (*Major.Minor.Patch*). En este esquema, el riesgo y la funcionalidad son las medidas de referencia. Los cambios rotundos (o drásticos) se indican aumentando el número principal (Major), las nuevas características no rotativas incrementan el número menor (Minor) y los cambios muy ligeros de forma que el riesgo de problemas en softwares derivados es muy pequeño se indican por Patch. Inicialmente, todos los paquetes son publicados en versión 1.0.0,
- `exposed-modules`: Una lista con los módulos que expone públicamente el paquete. Hay que tener en cuenta que el orden en el que son declarados será el orden en el que aparezcan en la documentación del repositorio de Elm. Los módulos se pueden agrupar en categorías.

- `elm-version`: El rango de versiones de Elm con las que se puede trabajar con el paquete.
- `dependencies`: Una lista de todos los paquetes de los que dependen los módulos del paquete junto con los rangos de versiones aceptados de los mismos.
- `test-dependencies`: Todos los paquetes implicados en los test. No deben aparecer en los paquetes finales, sino que quedan reservadas para las pruebas durante el desarrollo.

Vistas las configuraciones de cada uno de los tipos de proyectos, vamos a pasar a exponer los componentes implicados en cada uno de ellos, primero abordando el desarrollo de paquetes (back-end) y después el desarrollo de aplicaciones (front-end).

3.1.5. Desarrollo de Paquetes

Ya se sabe que el desarrollo de una librería, al igual que cualquier otro software, es como el proceso de resolver puzzles, se comienza con pocas piezas y es fácil encajar cada una en su sitio pero, a medida que aumenta su complejidad, la cantidad de piezas hace que se dificulte la interpretación de la imagen final y su manipulación.

```
{
  "type": "package",
  "name": "elm-lang/core",
  "summary": "Elm's standard libraries",
  "license": "BSD-3-Clause",
  "version": "6.0.0",
  "exposed-modules": {
    "Primitives": [
      "Basics",
      "String",
      "Char",
      "Bitwise",
      "Tuple"
    ],
    "Collections": [
      "List",
      "Dict",
      "Set",
      "Array"
    ],
    "Error Handling": [
      "Maybe",
      "Result"
    ],
    "Debug": [
      "Debug"
    ],
    "Effects": [
      "Platform",
      "Platform.Cmd",
      "Platform.Sub",
      "Task",
      "Process"
    ]
  },
  "elm-version": "0.19.0 <= v < 0.20.0",
  "dependencies": {},
  "test-dependencies": {}
```

Código 3.2: Archivo elm.json del paquete core

En un proyecto de software la filosofía es la misma, es preferible construir módulos dedicados y después establecer módulos que hagan de todos ellos una pieza perfectamente sincronizada, para lo cual resulta fundamental mantener buena documentación y estructura organizativa, de forma que tendamos a minimizar los errores y facilitar su detección y corrección, así como permitir que los propios, u otros, desarrolladores puedan realizar tareas de mantenimiento de forma fácil.

Algunas de las características de Elm que exploramos anteriormente, como la inmutabilidad, las funciones puras, las pruebas, y un poderoso sistema de tipos nos ayudan a escribir programas robustos, pero no permiten organizar el código de una manera fácil de mantener.

Sin embargo, Elm proporciona tres características diseñadas específicamente para la organización y documentación: organización del código en módulos y paquetes, formato de documentación estructurado y preciso y la Elm Architecture. Como señalamos previamente, esta última está más enfocada al desarrollo de aplicaciones, por lo que será tratada un poco más adelante, cuando se vea dicho apartado.

Valores, Expresiones y Funciones en Elm

Si realizamos una jerarquía de los elementos del lenguaje Elm, los elementos más básicos corresponden a los valores, que corresponden a instancias de un determinado tipo. Elm cuenta con una serie de tipos fundamentales definidos en distintos módulos del paquete *elm/core*, entre los que se encuentran `Int`,

`Float`, `Char`, `String`, `Bool`, *Maybe*, etc. Así como distintos contenedores de los mismos *Array*, *List*, *Tuple*, `Set`, *Dict*, *Record* ...

Con dichos valores podemos crear expresiones que corresponden a una agrupación de distintos valores según determinados operadores y dependiendo del tipo, o tipos, de las variables que aparezcan en la expresión. Dicha expresión es evaluada obteniéndose un valor. Como en la mayoría de lenguajes, también se pueden definir variables y crear expresiones con ellas. Para la definición de variables se utiliza la notación *Camel case* utilizando el operador = para asignarle un valor.

A medida que escribimos más expresiones, inevitablemente desearemos reutilizar algunas de ellas en varios lugares. Podemos lograr la reutilización mediante el uso de funciones para encapsular un montón de expresiones y dar nombres a la tarea colectiva que realizan esas expresiones.

En realidad, en Elm todo son funciones, algunas constantes (como los valores) y otras que dependen de variables (expresiones dependientes de variables, funciones nombradas (`areaCirculo`) o funciones lambda `i ->i+3`).

```
---- Elm 0.19.1 ------------------------------------------------------------------
Say :help for help and :exit to exit! More at <https://elm-lang.org/0.19.1/repl>
--------------------------------------------------------------------------------
> 1+1
2 : number
> "hola" ++ "¿Cómo estás?"
"hola¿Cómo estás?" : String
> (3 <= 5 ) || False
True : Bool
> 2::[3,7]
[2,3,7] : List number
>
> x = 5
5 : number
> 2*pi*x
31.41592653589793 : Float
> y = 25
25 : number
> y = "roma"
"roma" : String
>
> areaCirculo r=
|   pi*r^2
|
<function> : Float -> Float
> areaCirculo 3
28.274333882308138 : Float
> List.map (\i -> i+3) [1,2,3,4]
[4,5,6,7] : List number
>
```

Figura 3.3: Ejemplos de expresiones en Elm
Fuente propia. Utilzando la herramienta Elm REPL

Gracias al tipado fuerte y al mecanismo de inferencia de tipos Elm muestra los tipos de los valores (o funciones) obtenidos (ver *figura 3.3*). Aunque Elm es capaz de inferir los tipos, cuando desarrollemos un módulo deberemos establecer los tipos de las funciones que definamos.

Si bien Elm provee algunos tipos básicos, también proporciona la posibilidad de definir nuestras propias estructuras de datos a través de dos mecanismos: los *tipos* y los *alias*.

- **Tipos**. También denominados *union types* y equivalente a lo que generalmente se conoce como *Algebraic Data Type (ADT)*. Es un tipo estructurado que se forma al componer otros tipos a través de constructores, e incluso de sí mismos cuando se definen tipos recursivos. Para ilustrarlo veamos un par de ejemplos:

  - Supóngase que se quiere definir un tipo para `Plot`, que representa una gráfica que puede estar en 2D o 3D. Este tipo podría definirse en Elm como:

```
type Plot =
    Plot2D  List (Float, Float)
    | Plot3D  List (Float, Float,Float)
```

    De forma que para crear un objeto de tipo `Plot` debemos hacerlo a través de los constructores `Plot2D` y `Plot3D` aportando a cada uno una lista de tuplas de reales (los puntos) 2-arias y 3-arias, respectivamente.

  - Supóngase ahora, que se quiere definir un tipo para `BinaryTree`, que representa un árbol binario, tal que todo nodo, o bien es una hoja, o bien tiene dos hijos, los cuales corresponden a su vez a dos árboles binarios. Este tipo se construye a partir de elementos de su mismo tipo, esto es, un tipo recursivo. En Elm podríamos definirlo como:

```
type BinaryTree a =
    Node a (BinaryTree a) (BinaryTree a)
    | Leaf a
```

    Nótese que hemos introducido en el tipo un parámetro, ya que los árboles binarios pueden contener valores de distinto tipo, de forma que podemos crear un árbol de números enteros (o cualquier otro tipo) a través de los constructores. Por ejemplo, el árbol mostrado en la *figura 3.4* podríamos crearlo como:

```
binaryTree =
    Node 5
        ( Node 3
            (Leaf 1)
            (Leaf 4)
        )
        ( Leaf 7)
```

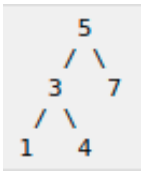


Figura 3.4: BinaryTree

  - Pero no sólo sirve para definir tipos con varios constructores, también puede tener un único constructor. Imaginemos, por ejemplo, el tipo `Tree` similar al anterior pero donde ahora el número de hijos es indeterminado (puede ser 0, 1, 2, o más). Este tipo podemos definirlo como un nodo (el nodo raíz) con una lista de subárboles correspondientes a cada uno de los hijos, de forma que si la lista es vacía entonces estaríamos ante una hoja. En Elm podríamos definirlo como:

```
type Tree a = Node a (List (Tree a))
```

    De forma que podríamos definir el árbol de la *figura 3.4* como:

```
tree = Node 5 [Node 3 [Node 1 [], Node 4 []], Node 7 []]
```

- **Alias**. Un alias de tipo es un nombre más corto para un tipo. Por ejemplo, imaginemos que queremos definir el tipo *SimpleDigraph* que corresponde a un grafo dirigido simple (sin aristas múltiples ni lazos). Esto podríamos hacerlo con un 'objeto' con dos campos, uno para los nodos correspondiente a un diccionario con claves los ids de los nodos (enteros) y con valores las etiquetas

de los mismos; y un campo para las aristas correspondiente a un diccionario de claves formadas por pares de enteros, y con valores en el tipo de la etiqueta de la arista.

Aunque no lo hemos comentado antes, ese tipo de objetos con atributos pueden crearse en Elm a partir del tipo `Record`, similar al `Struct` de C o al propio `Record` en Haskell. La definición anterior podría, entonces, escribirse como:

```
type alias Graph a b =
    { nodes : Dict Int a
      edges : Dict (Int, Int) b
    }
```

Visto todo lo anterior, ya podríamos desarrollar un pequeño programa en Elm. Sin embargo, no hemos mostrado ninguna instrucción que permita tomar distintas alternativas. Para este fin Elm provee dos estructuras de control básicas: *bloques condicionales* y *búsqueda o coincidencia de patrones*.

La forma más común de expresar una lógica condicional en Elm es a través de un bloque *if-then-else*. Este tipo de instrucción requiere de tres partes:

- Una condición.
- Una expresión (rama) que será evaluada si la condición es verdadera.
- Una expresión (rama) que será evaluada si la condición es falsa.

Ilustrémoslo con un ejemplo. Imagínese que se quiere definir la función `collatz`, que calcula la secuencia de Collatz de un número natural que se construye realizando sucesivamente y hasta llegar a 1 las siguientes operaciones:

- Si el número es par, entonces se divide por 2.
- Si el número es impar, entonces se multiplica por 3 y se suma 1.

De forma que, por ejemplo, para $n = 6$ se tendría la sucesión: $6, 3, 10, 5, 16, 8, 4, 2, 1$. En Elm podemos programar fácilmente esta función haciendo uso de la recursión, concatenación de elementos a una lista `(::)` y de los bloques *if-then-else*, tal y como se muestra a continuación:

```
> collatz n =
|   if n == 1 then
|     [1]
|   else
|     if modBy 2 n == 0 then
|       n::(collatz (n//2))
|     else
|       n::(collatz (3*n+1))
|
<function> : Int -> List Int
> collatz 6
[6,3,10,5,16,8,4,2,1]
    : List Int
>
```

Figura 3.5: Ejemplo de bloque condicional en Elm
Fuente propia. Utilizando la herramienta Elm REPL

Pero, una vez visto los tipos, cómo lo hacemos si queremos tomar distintas alternativas dependiendo del subtipo de un elemento. Por ejemplo, imagínese que sobre el tipo `Plot` se quiere tener una función que calcule la homotecia de centro c y razón k, definida como una transformación afín tal que $\vec{p'} = k\vec{p} + (1-k)\vec{c}$ a partir de una gráfica.

Aunque esta función posee cierta complejidad, hemos decidido exponerla para tratar unos últimos detalles que no han sido abordados hasta el momento, al menos con cierta profundidad, y que veremos junto a la exposición de un caso de uso que ilustre estos aspectos. La *figura 3.6* muestra la función `homotethyPlot` que calcula justamente la homotecia de la nube de puntos:

```
> homothetyPlot : List Float -> Float -> Plot -> Maybe Plot
| homothetyPlot c k plot =
|   case c of
|     [cx, cy] ->
|       let
|         homothetyPlot2D ps =
|           Plot2D <|
|             List.map (\(px, py) -> (k*px + (1-k)*cx, k*py + (1-k)*cy)) ps
|       in
|       case plot of
|         Plot2D ps -> (Just << homothetyPlot2D) ps
|         Plot3D _ -> Nothing
|     [cx, cy, cz] ->
|       let
|         homothetyPlot3D ps =
|            Plot3D <|
|              List.map
|                (\(px, py, pz) -> (k*px + (1-k)*cx, k*py + (1-k)*cy, k*pz + (1-k)*cz))
|                ps
|       in
|       case plot of
|         Plot3D ps -> (Just << homothetyPlot3D) ps
|         Plot2D _ -> Nothing
|     _ -> Nothing
|
<function> : List Float -> Float -> Plot -> Maybe Plot
```

Figura 3.6: Ejmplo de uso de Pattern Matching y Nested Functions
Fuente propia. Utilizando la herramienta Elm REPL

- **El comodín** `_`. Este símbolo se puede utilizar cuando un valor que se tiene como entrada no va a ser utilizado en la evaluación de dicha función o dicha rama. Tal y como se muestra en la figura anterior, el conjunto de puntos para aquellos casos que no tienen la aridad adecuada es irrelevante, ya que se devuelve `Nothing`.

- **El tipo de dato** `Maybe a`**.** Corresponde a los llamados tipos polimórficos, esto es, tipos que encapsulan otros tipos. En el caso de `Maybe`, su uso más común corresponde al trato con una función parcial (a diferencia de una función total), porque el dominio de nuestra función (las entradas) solo se asigna parcialmente al codominio (una salida). En otras palabras, hay entradas para nuestra función que son técnicamente válidas (desde una perspectiva de tipo), pero que no pueden producir una salida que realmente tenga sentido. Esto podría hacer que se produjesen errores en tiempo de ejecución, cosa que, como ya hemos resaltado, Elm intenta evitar por todos los medios. De forma que con `Maybe`, la existencia, o no, de un valor se refleja clara y completamente en el tipo, y nos vemos obligados a manejar el caso en el que no obtenemos ningún valor significativo.

- **Expresiones** `let`: Crean un ámbito local, esto es, una región de un programa donde existen funciones que son accesibles sólo en dicho ámbito. Estas expresiones son sensibles a la indentación por lo que las funciones definidas en una expresión `let` deben ir indentadas al menos en un espacio. Las instrucciones de una misma rama situadas debajo de `in` establecen el alcance de las funciones, esto es, sólo ellas podrán acceder a las funciones definidas en ese contexto.

- **Las funciones lambda**. Son básicamente funciones anónimas de un solo uso, normalmente como argumento de funciones de orden superior. Toda función lambda es, normalmente, escrita entre paréntesis y se nota con una barra invertida seguida de los parámetros (separados por espacios), del símbolo `->` y, tras él, el cuerpo de la función.

- **Las funciones de orden superior**: Las funciones de Elm pueden tomar funciones como parámetros y funciones de retorno como valores de retorno. En tal caso, se denomina *función de orden superior*. Las funciones de orden superior no son sólo una parte de la experiencia de Elm, son prácticamente LA experiencia de Elm. Vamos a presentar algunas de las más importantes:

- `map`. Toma una función y una lista y aplica dicha función a cada uno de los elementos de la lista por separado, produciendo una nueva lista de elementos del mismo u otro tipo.
- `filter`. Toma un predicado (un predicado es una función total que establece si un elemento cumple una condición o no, devolviendo un valor booleano) y una lista y devuelve una lista con los elementos que satisfacen dicho predicado.
- `takeWhile/dropWhile`. Toma un predicado y un contenedor (ordenado) y devuelve una lista que contiene/elimina sólo los elementos anteriores al primer elemento que incumple el predicado.
- `foldl` y `foldr`. La recursión es bien conocida como técnica sustitutiva de los bucles en los lenguajes funcionales. Sin embargo, las funciones de plegado son otra alternativa (casi siempre la alternativa) cuando se ajusta a un patrón claro. Las funciones de plegado *fold* toman una función binaria, un valor inicial (denominado generalmente acumulador) y un contenedor (ordenado) sobre el que realizar el plegado (por lo general una lista). La función binaria toma como parámetros un elemento del tipo del contenedor y el acumulador y produce un nuevo acumulador. En su acción, se va recorriendo la lista aplicando la función de plegado sucesivamente sobre el elemento y el acumulador considerados (ambos) en cada paso. `foldl` lo hace recorriendo el contenedor de principio a fin y `foldr` al revés.

- *Composición de funciones.* En realidad, todas las funciones en Elm toman un único parámetro de entrada, sin embargo es posible definir funciones con varios parámetros gracias a la *currificación*. Esto quiere decir que Elm es capaz de generar funciones parcialmente aplicadas, esto es, que esperan un parámetro para ser evaluadas. La currificación también tiene un operador asociado, que se ha utilizado intensamente pero hasta ahora de forma puramente espontánea, el operador (espacio). En efecto, el espacio también es un operador (una función) y además goza de la mayor prioridad. De forma que por ejemplo la función `max 3 4` es en realidad `(max 3)` que devuelve una función `Int ->Int` cuyo resultado corresponde al argumento, si éste es mayor que 3, y a 3 en otro caso, que es aplicada sobre el argumento 4.

  Esto pudiera parecer irrelevante pero es el motivo por el cual podemos definir funciones que están parcialmente evaluadas pero requieren de algún parámetro para poder ser completamente evaluadas. Además, esto permite componer funciones con relativa facilidad, para lo cual existen 2 pares de operadores básicos:

  - *Operador de avance* `|>` y *Operador de retroceso* `<|`. Corresponden a la aplicación de una función. Su presencia, bajo la existencia del operador espacio, pudiera parecer inútil, pero no lo es. Mientras que la aplicación de una función normal (poner un espacio entre el nombre de la función y el operando) tiene una precedencia muy alta, las funciones `|>` y `<|` tienen la precedencia más baja, esto es, primero se evalúa la expresión (situada en el lado de `|`) y el resultado se pasa a la función (situada en el lado de `<` o `>`). Normalmente, suele ser muy conveniente para evitar el uso excesivo de paréntesis.
  - *Operadores de composición* `>>` *y* `<<`. Equivalen a la pura composición matemática. De forma que `f << g` (esto es, $f \circ g$) toma un valor del tipo que tomase g y devuelve uno del tipo que devolviese f. Sin embargo, solo es aplicable sobre funciones que reciben un sólo parámetro(gracias a la currificación, es el caso general en Elm). Si, por ejemplo, se quiere calcular $-(max(3,4))$, podría hacerse como `(negate << max 3) 4`.

Como hemos podido comprobar, las funciones (no hemos llegado a presentar ningún algoritmo, sino simples funciones matemáticas) pueden requerir cierta complejidad. De este modo, cuando pasamos de desarrollar un pequeño programa a una librería completa, la cantidad de funciones y tipos necesarios puede ser enorme, de forma que si tuviésemos que tenerlas todas juntas en un mismo fichero para poder reutilizarlas el tratamiento y mantenimiento del mismo resultaría inviable. Pero es más, normalmente unas funciones tendrán mucha relación con unas pocas y muy poca con el resto, de forma que tampoco resulta coherente mantenerlas todas juntas. Se ha de proporcionar otro mecanismo, que abordaremos justamente en el siguiente punto.

Desarrollo de módulos en Elm

Tal y como avanzamos en el final del punto previo, Elm permite organizar los tipos y funciones en *módulos*, normalmente según dominios de aplicación más específicos, que permiten a los desarrolladores exportar (exponer) aquellos tipos y funciones que sean relevantes, manteniendo ocultas las que posean un carácter auxiliar, de forma que las funciones expuestas estén disponibles para su importación en otros módulos externos que requieran de su uso. Esto, además, proporciona un mecanismo (parcial) para que las funciones puedan ser expuestas públicamente y otros desarrolladores puedan hacer uso de ellas en sus propias implementaciones.

Para definir un módulo se utiliza la siguiente estructura al comienzo del fichero:

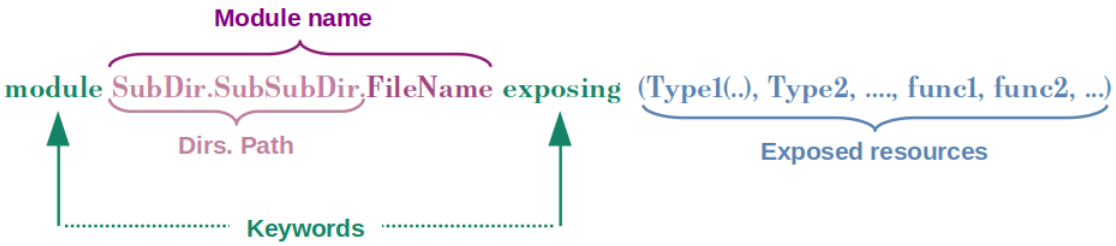


Figura 3.7: Definición de un módulo en Elm
Fuente propia. Inspirada en Beginning Elm - Easier Code Organization

Precedido de la palabra clave `module`, el nombre del módulo se ajusta al sistema de archivos en el que se encuentre separando mediante un punto cada subdirectorio desde la raíz */src* hasta el archivo que contiene el módulo, cuyo nombre debe corresponder precisamente a la última parte del nombre del módulo. Se recomienda que el nombre del módulo completo no supere los 20 caracteres. Posteriormente, precedida de la palabra clave `exposing` y entre paréntesis, se enumeran los tipos y funciones que se exponen del módulo, las que se quieren hacer importables en otros módulos. Una cuestión particular es la de los tipos personalizados que deben ir seguidos de (..) si se quieren exponer también los constructores y no solo el tipo como tal. Aunque no se recomienda su uso, la secuencia .. sirve a modo de comodín para exponer todo el contenido del módulo.

Importación de módulos

La organización en módulos permite el uso de funciones de otros módulos y paquetes a través de la importación, pero es necesario que el paquete correspondiente esté referenciando en las dependencias del proyecto. Hasta el momento no hemos hecho referencia a las mismas, así que es el momento de dar algunos detalles sobre ellas.

Como ya se mencionó anteriormente, el archivo *elm.json* contiene la información sobre el proyecto, con un apartado para las dependencias. Aunque inicialmente se suele contar únicamente con la dependencia *elm/core*, cuando queremos hacer uso de otros paquetes hemos de instalarlos y actualizar el archivo *elm.json* añadiéndolos a las dependencias. Para ello, es suficiente con ejecutar en un terminal desde el directorio principal del proyecto (el que contiene el archivo *elm.json*) el comando `elm install githubname/package` de forma que, además de instalar el paquete, las dependencias son actualizadas automáticamente (si el desarrollador lo desea).

Una vez instalado el paquete correspondiente ya podemos hacer uso de las funciones expuestas en el mismo importándolas del módulo correspondiente mediante el uso de la instrucción mostrada en la *figura 3.8*. Aunque podemos importar directamente las funciones, la recomendación es importar solamente los tipos y utilizar el alias o el nombre (si no se pone *as Alias*) para el resto de las funciones.

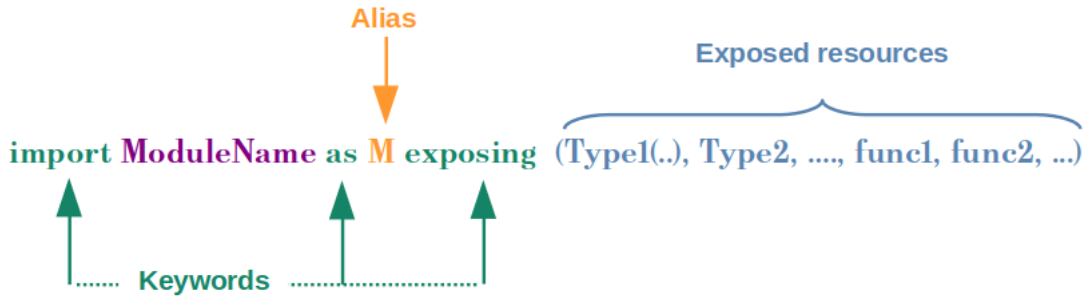


Figura 3.8: Importación de un módulo en Elm
Fuente propia: inspirada por Beginning Elm - Easier Code Organization

Las importaciones de los módulos han de realizarse, obligatoriamente, al comienzo del módulo, en concreto, justo detrás de la cabecera. A continuación ya se pueden utilizar todas las funciones del paquete. Aquellas funciones y tipos incluidos en algún campo `exposing` pueden ser llamadas directamente por su nombre. Sin embargo, para evitar conflictos, es recomendable que sólo se expongan los tipos en los imports y las funciones se llamen a través del nombre del propio módulo o, preferiblemente , por el alias (sólo porque es más corto). De forma que, por ejemplo, si queremos crear un grafo, para lo que utilizaremos el módulo `Graph` del paquete `Graph` que previamente habrá sido instalado utilizando `elm install elm-community/graph`, tal como se muestra en la *figura 3.9* importaremos el módulo y definiremos el grafo utilizando la función `fromNodesAndEdges`.

```
> import Graph as G exposing (Graph, Node, Edge)
> import String as S
> g =
|   G.fromNodesAndEdges
|     [Node 1 5, Node 2 3, Node 3 7, Node 4 1, Node 5 4]
|     [Edge 1 2 (), Edge 1 3 (), Edge 2 4 (), Edge 2 5 ()]
|
Graph (Inner { left = Inner { left = Leaf { key = 1, value = { incoming = Empty, node = { id = 1, lab
el = 5 }, outgoing = Inner { left = Leaf { key = 2, value = () }, prefix = { branchingBit = 1, prefix
Bits = 2 }, right = Leaf { key = 3, value = () }, size = 2 } } }, prefix = { branchingBit = 2, prefix
Bits = 0 }, right = Inner { left = Leaf { key = 2, value = { incoming = Leaf { key = 1, value = () },
 node = { id = 2, label = 3 }, outgoing = Inner { left = Leaf { key = 4, value = () }, prefix = { bra
nchingBit = 1, prefixBits = 4 }, right = Leaf { key = 5, value = () }, size = 2 } } }, prefix = { bra
nchingBit = 1, prefixBits = 2 }, right = Leaf { key = 3, value = { incoming = Leaf { key = 1, value =
 () }, node = { id = 3, label = 7 }, outgoing = Empty } }, size = 2 }, size = 3 }, prefix = { branchi
ngBit = 4, prefixBits = 0 }, right = Inner { left = Leaf { key = 4, value = { incoming = Leaf { key =
 2, value = () }, node = { id = 4, label = 1 }, outgoing = Empty } }, prefix = { branchingBit = 1, pr
efixBits = 4 }, right = Leaf { key = 5, value = { incoming = Leaf { key = 2, value = () }, node = { i
d = 5, label = 4 }, outgoing = Empty } }, size = 2 }, size = 5 })
    : Graph number ()
> G.toString (Just<<S.fromInt) (\_ -> Nothing) g
"Graph [Node 1 (5), Node 2 (3), Node 3 (7), Node 4 (1), Node 5 (4)] [Edge 2->5, Edge 2->4, Edge 1->3,
 Edge 1->2]"
    : S.String
```

Figura 3.9: Ejemplo de importación de módulos
Fuente propia. Utilizando la herramienta Elm REPL

En resumen, Elm hace que sea muy fácil agrupar funciones, definiciones de tipos y otros valores mediante módulos. La sintaxis para crear e importar módulos es sencilla y permite compartir nuestros módulos con otros programadores. Para ello, es necesario incluirlos formando un paquete, documentarlo bajo unas guías concretas y finalmente publicarlo a través de GitHub y el catálogo en línea, tal y como mostraremos en el siguiente punto.

Documentación y publicación de paquetes en Elm

Antes de publicar un paquete es necesario, además de realizar las pruebas de funcionamiento pertinentes, desarrollar cierta documentación de cada uno de los módulos, tipos y funciones que se expondrán en el paquete.

En ese sentido, Elm posee un sistema de documentación basado en Markdown que es incorporado directamente sobre los módulos, lo que facilita las tareas de mantenimiento tanto de la documentación como del código para el desarrollador, además de favorecer su consulta de la misma publicándola para todos los paquetes bajo un mismo formato corporativo. A continuación se presenta las partes básicas de que debe constar la documentación de un módulo.

En primer lugar, justo debajo de la cabecera (y antes de la sección de *imports*) se ha de definir la estructura de la documentación del módulo, que constará de un breve resumen del objetivo del mismo así como una enumeración de los elementos que se exportan. Nótese que la estructura de dicha enumeración (al igual que el resto de la documentación) es descrita entre los símbolos de comentario de documentación {-— ... -} en formato Markdown, renderizándose según el orden de dichas declaraciones, de forma que podemos crear las secciones que creamos convenientes para que la documentación esté bien organizada y sea fácil de consultar.

Para llevar a cabo esas declaraciones, la palabra clave `@docs` precede a una lista de elementos (nombres de funciones o tipos definidos en el paquete) cuya descripción aparecerá estrictamente en ese orden. Estas declaraciones serán sustituidas por la documentación específica de cada uno de los elementos descritos. Para ello, delante de cada tipo o función expuesta ha de aparecer, entre los símbolos de comentario de documentación, un pequeño texto descriptivo del elemento correspondiente. Es común incluir además un fragmento con algún ejemplo de uso de la función, indentando dicho bloque para que aparezca renderizado como código.

Nótese que las reglas son claras pero no limitan la capacidad expresiva ni explicativa de la documentación. En la *figura 3.10* se presenta un módulo completo y documentado junto a la renderización del mismo. En concreto, se presenta el módulo LogicUS.PL.DPLL que abordaremos más tarde en la implementación.

Como últimos detalles en la documentación del paquete hay que añadir un archivo *README.md* que servirá de presentación del módulo y en el que se expondrán las principales funcionalidades del mismo.

Los últimos pasos para la publicación del paquete consisten en actualizar el código del repositorio con el código final del paquete y publicarlo. Para ello:

1. En el campo `exposed-modules` del archivo de configuración *elm.json* se deben indicar los módulos que se exportan en el paquete.
2. Crear o actualizar el repositorio de GitHub que albergará el código, crear una tag con la versión del mismo y publicar el paquete con el comando `elm publish`.

Finalmente, para realizar actualizaciones del paquete, una vez se hayan hecho los cambios oportunos, se puede ejecutar `elm diff` y `elm bump` para rastrear los cambios y actualizar la versión semántica según los mismos. Tras esto, basta etiquetar la confirmación, enviar a GitHub y ejecutar `elm publish` para publicar la nueva versión del paquete.

Hasta aquí hemos presentado el procedimiento de desarrollo de un paquete de forma totalmente detallada, sus elementos, organización, importación/exportación, así como la documentación y publicación del mismo. En el próximo apartado pasaremos a ver la faceta front-end con la conocida como *Elm Architecture*.

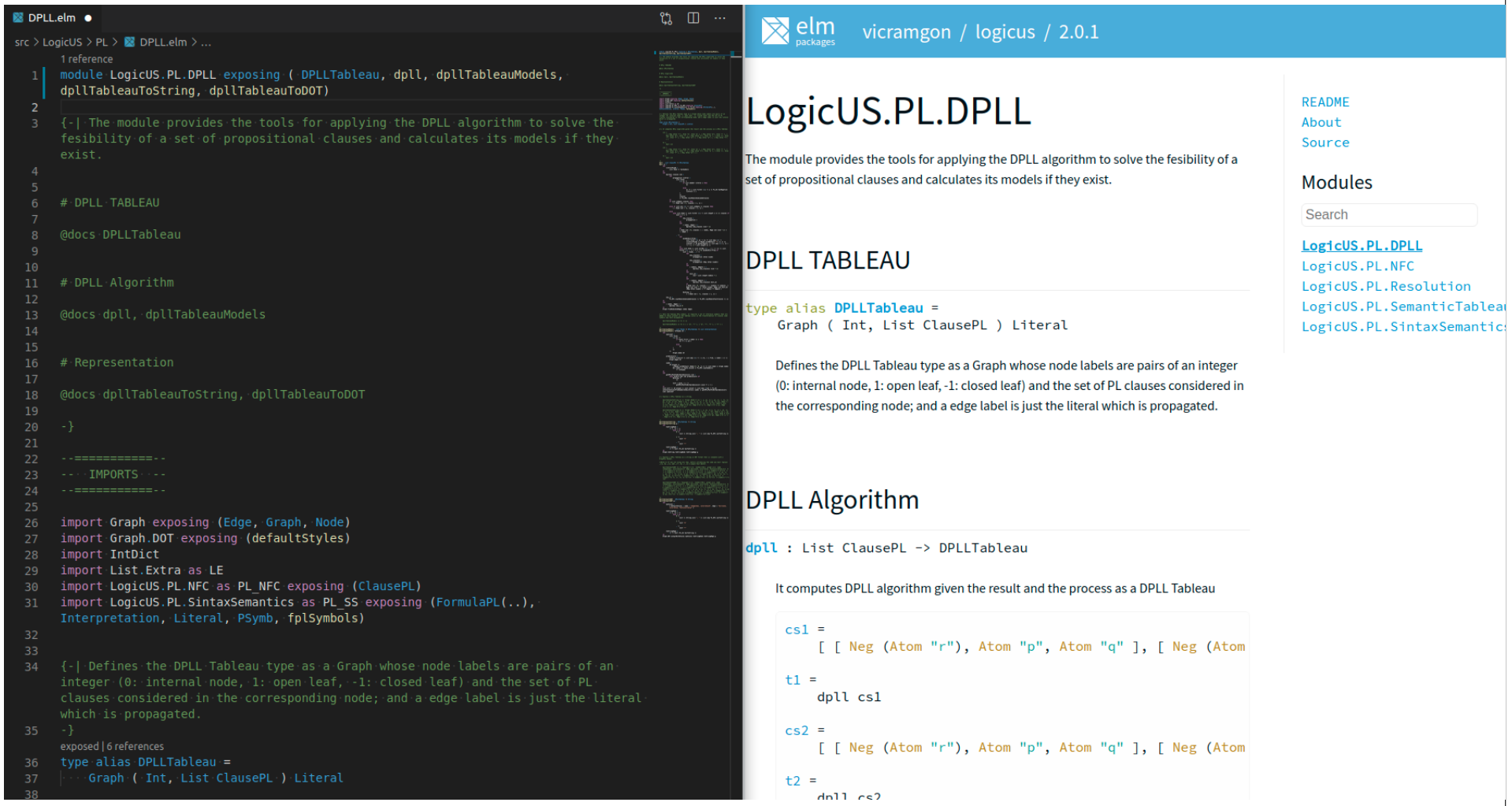


Figura 3.10: Ejemplo de documentación y publicación de paquetes en Elm
Fuente propia

3.1.6. Desarrollo de Aplicaciones en Elm

Elm Architecture

A muy alto nivel, podemos detectar que las aplicaciones web suelen tener dos partes principales: estado, e interfaz de usuario (UI). Una aplicación comienza con un estado inicial y presenta ese estado al usuario a través de la interfaz de usuario. El usuario realiza alguna acción a través de un elemento de la interfaz de usuario que modifica el estado inicial. El nuevo estado se presenta al usuario para que realice más acciones. La *figura 3.5* muestra la interacción entre un estado y la interfaz de usuario en una aplicación hipotética que permite a los usuarios registrados crear publicaciones de blog.

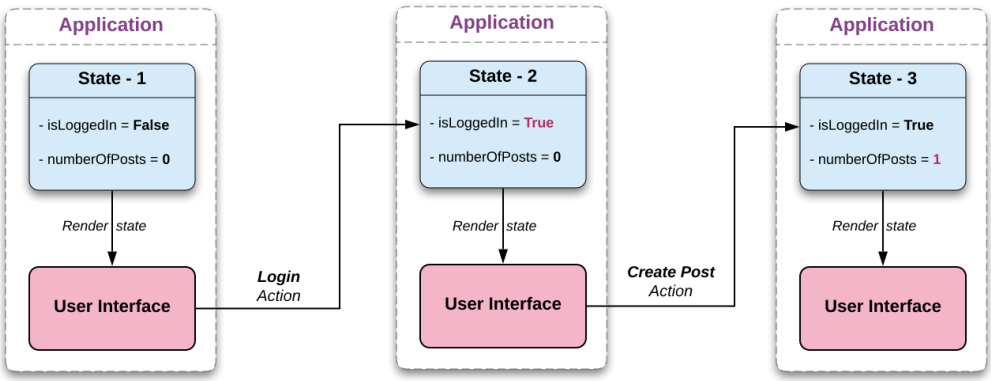


Figura 3.11: Ejemplo de interacción en aplicación Web
Fuente: Beginning Elm - Model View Update - Part I

Este proceso es modelado en Elm según la llamada Elm Architecture, un patrón para el diseño de aplicaciones interactivas como pueden ser las aplicaciones web o incluso juegos. Esta arquitectura surgió de forma natural en Elm. En lugar de que alguien lo definiese, los primeros programadores de Elm comenzaron a definir los mismos patrones básicos en su código, dando lugar a la arquitectura que expondremos a lo largo de este apartado. De hecho, es tan intuitiva y adecuada que otros proyectos (frameworks) se han inspirado en estos mismos patrones.

La arquitectura propuesta propone un patrón que incorpora algunas modificaciones al modelo MVC (*Model-View-Controller*) y que constituye el patrón MVU (*Model-View-Updater*), cuyos elementos corresponden a:

- *Modelo*: Representa el estado de la aplicación y corresponde a una estructura de datos (típicamente un registro, análogo a los objetos de otros lenguajes, por ejemplo JSON en JS).
- *Vista*: Corresponde precisamente a la parte gráfica de la interfaz. De hecho, en Elm suele corresponderse con una función que toma como entrada un modelo y devuelve el código HTML asociado al modelo.
- *Updater*: Corresponde a un elemento que toma un evento que es lanzado desde la vista (evento interactivo) y realiza los cambios pertinentes en el modelo.

De forma general, esta arquitectura se puede ver como una enorme máquina que se ejecuta a perpetuidad, siguiendo el patrón mostrado en la *figura 3.11*, en el que el programa opera siguiendo los siguientes pasos: tomar un modelo inicial, presentarlo al usuario, escuchar los mensajes, actualizar el modelo en base a esos mensajes y presentar el modelo actualizado nuevamente al usuario.

En los siguientes puntos vamos a concretar cómo es modelado en Elm cada uno de los elementos presentados: el modelo (almacén de estado), la estructura de la vista, los mensajes de eventos de cambios, y el manejo de la actualización.

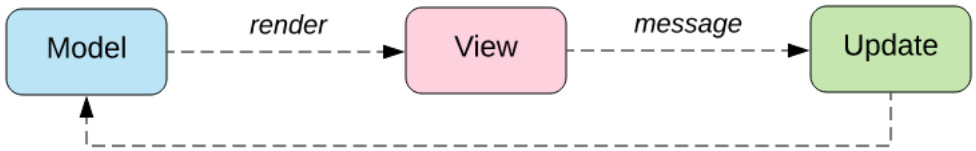


Figura 3.12: Patrón de funcionamiento general de la *Elm Architecture*
Fuente: Beginning Elm - Model View Update - Part I

Definición del Modelo en Elm

Para la definición del modelo en Elm es necesario tener en cuenta qué datos son necesarios en la aplicación que se esté considerando. Si, por ejemplo, lo que se estuviese considerando fuese una aplicación para un comercio para contabilizar el número de individuos presentes en un local, entonces el estado correspondería en cada instante al número de personas presentes en el local, de forma que esto podríamos representarlo simplemente por un entero (Int).

Sin embargo, los modelos no siempre son tan simples, todo dependerá de la complejidad de la aplicación. Un recurso muy frecuente a la hora de crear el tipo de un modelo es el uso del tipo `Record`, en el que cada uno de los elementos necesarios es añadido al mismo con una clave distinta y con el tipo necesario para albergar la información de dicho campo. Nótese que esto nos da la posibilidad de implementar aplicaciones con un alto grado de complejidad, proporcionándonos además, a través de unos adecuados nombres para las claves, un valor semántico de los elementos del modelo.

Para ilustrar lo anterior, piénsese en una aplicación para el control de embarque en un vuelo (usaremos este ejemplo durante todo el apartado para ir ilustrando los distintos conceptos que se expondrán) de forma que lo único que requiere es de un registro que determine si una persona posee un billete para dicho vuelo y, en caso positivo, le informe del número de asiento. Nuestra aplicación debe manejar como datos el identificador del pasajero (DNI), el código del billete, la relación billete-persona-asiento y si ha accedido ya o no. Por tanto, podríamos definir el modelo, haciendo uso del tipo `Record` como:

```
type alias TicketInfo =
    { passenger : String
     ,seat : Int
    }

type alias Model =
    { passengerID :String
      , ticketID : String
      , registry : Dict String (TicketInfo, Bool)
      , message : String
    }
```

En primer lugar utilizaremos un primer record para definir los elementos que lleva asociado el biellete, correspondientes al pasajero de dicho billete (DNI, pasaporte...) que podríamos representar por una `String` y el asiento asociado, que podemos modelar con un formato numérico.

Ahora el modelo de nuestra aplicación debe contemplar la identificación del pasajero y del ticket que se van a registrar, el registro de vuelo y un elemento que guarde el mensaje que se va a mostrar en pantalla. El registro correspondería a una asociación entre la identificación del billete, su información y un elemento booleano que permita distinguir los ya registrados. Por ello, es conveniente utilizar un diccionario en el que las claves corresponden a la identificación de los billetes (en formato String) y los

valores asociados a un par con la información del billete y un booleano que indique si ha sido ya registrado.

Inicialización del modelo

Una vez diseñado el modelo, hemos de establecer cómo se inicializará el mismo en el despliegue de la aplicación. Para ello es necesario definir un elemento (función) que indique el estado inicial del modelo.

En el ejemplo del controlador de aforo la inicialización es sencilla, basta establecer el valor del modelo a 0. Ahora, en el caso del registro de embarque, si bien es claro que los campos `passengerID`, `ticketID` y `message`, serían inicializados como cadenas vacías (`""`), el registro ha de inicializarse con la relación billete-persona-asiento y con acceso falso para todos los registros. Por lo que para cada vuelo hemos de reprogramar la aplicación, lo cual no parece muy cómodo.

Elm provee varios mecanismos para la interacción con JavaScript y otros servidores. Abordaremos este tema en mayor profundidad en futuros puntos.

Presentación al usuario: Vista

La vista corresponde a lo que los usuarios deben ver al utilizar la aplicación, la interfaz final. La Arquitectura Elm, tal y como se muestra en la *figura 3.11*, crea la vista a partir de los datos del modelo, generando el código HTML de la misma. Para ello, se utilizan distintas funciones incluídas en el paquete *HTML*, que proporciona la gran mayoría de los elementos que existen en HTML a través de funciones y parámetros (que se corresponden con los atributos de dichos elementos en el lenguaje HTML).

Para diseñar la vista basta con realizar el diseño en HTML y después pasarlo a Elm con las funciones correspondientes. Por ejemplo, para el caso del registro de embarque sería suficiente contar con dos inputs de texto para el código del billete y el id del pasajero, un botón de comprobación del formulario, y un campo de texto para los mensajes, de forma que en Html podría tenerse algo como:

```
<div>
    <div class="form">
        <input type="text" placeholder="ID Ticket">
        <input type="text" placeholder="ID Passenger">
        <button> Check </button>
    </div>
    <div class="message">
    </div>
</div>
```

Ahora que tenemos la estructura podemos crear la función `view` exactamente con esa misma estructura y se visualizaría como se muestra en la *figura 3.13*.

Nótese que la página es generada con estilo base de HTML. Sin embargo Elm sí provee las herramientas necesarias para definir estilos CSS. Para ello contamos con varias alternativas:

- Uso de estilos en linea.
- Uso de un archivo externo CSS.
- Uso de un marco CSS.

Para utilizar estilos en linea Elm ofrece la función `style` del paquete `Html.Attributes`. Esta función toma dos argumentos correspondientes al nombre de la propiedad y el valor de la misma. Por ejemplo, para establecerun título `h2` con un margen a izquierda y derecha de 70px:

```
h2 [style "margin" "0 70px"] [text "Flight Registry"]
```

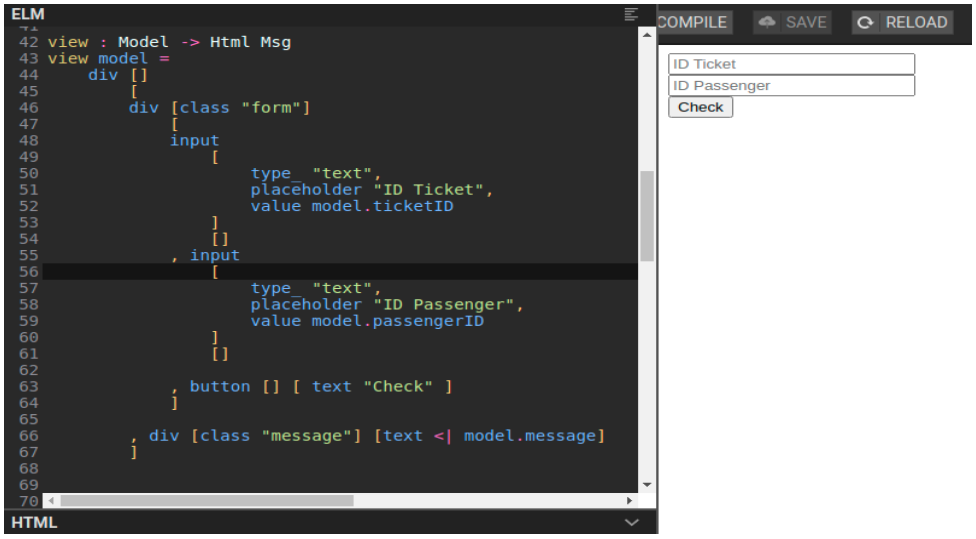


Figura 3.13: Ejemplo de una vista en Elm
Fuente propia. Utilizando la herramienta Ellie

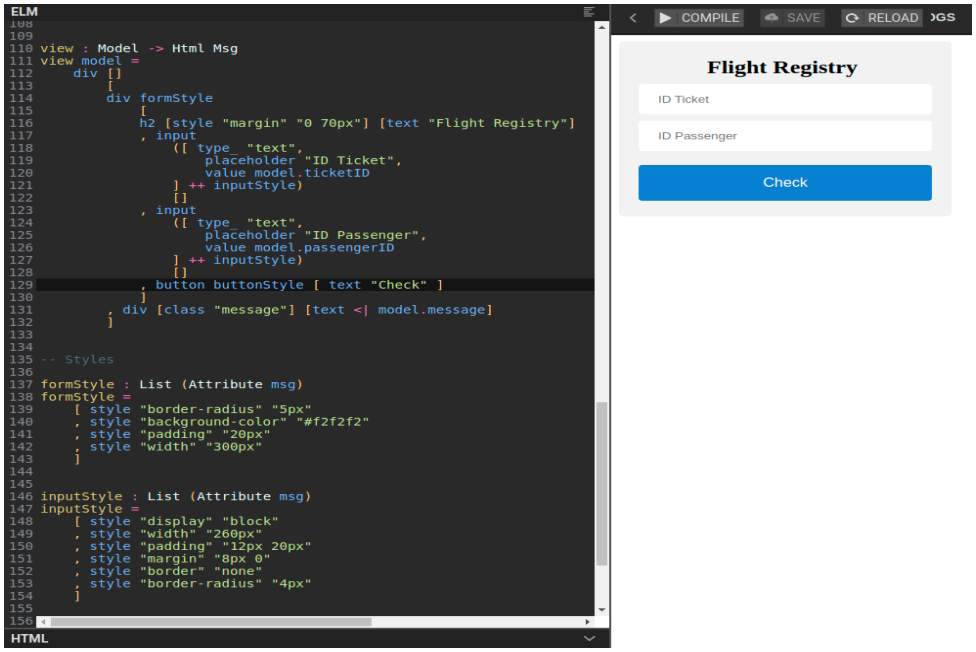


Figura 3.14: Incorporación de estilos CSS en línea en Elm
Fuente propia. Utilizando la herramienta Ellie

Vamos a dar entonces un estilo al formulario. Como establecer todos los valores directamente en la vista puede quedar algo confuso, vamos a hacerlo definiendo los estilos como listas y agregándolas a los demás atributos ya establecidos tal y como se muestra en la *figura 3.14*.

El uso de este tipo de estilos, al igual que ocurre en HTML, es poco recomendado, por lo que debe evitarse siempre que sea posible y utilizar en su lugar un archivo externo, para lo cual es necesario desarrollar un fichero CSS externo, y agregar las clases (por convención las clases son utilizadas para el CSS y los ids para JS) una vez se haya compilado la aplicación y se haya generado el HTML y JS correspondientes. Destacamos también que, como en HTML convencional, se pueden agregar el uso de librerías (como Bootstrap) adecuando convenientemente los nombres de las clases, aunque también se han desarrollado módulos especializados (elm-bootstrap) de elm que permiten su uso directo.

Manejo de mensajes con Updater

Si bien hemos presentado la composición de una aplicación básica, no hemos provisto ningún mecanismo de interacción con la misma, de forma que solo podríamos crear páginas estáticas. Para manejar los eventos y actualizar el modelo se utiliza la función de actualización (por convención `update`). En ella debe definirse el tratamiento para los eventos que corresponden a lo que en el ámbito de Elm se denominan *mensajes*. Los mensajes son un tipo que debe definir el usuario (normalmente `Msg`) que establecen las distintas acciones que se pueden llevar a cabo en la aplicación.

En el ejemplo anterior las acciones que se deben de realizar son la actualización de los campos leídos y el chequeo con el registro de referencia del modelo (actualizándolo marcando los ya chequeados), por lo que tendremos que definir tres tipos de mensaje, el de actualización del `passengerID`, el de actualización del `ticketID` y el de chequeo:

```
type Msg =
    ChangePassengerID String
    | ChangeTicketID String
    Checking
```

Una vez definidos los tipos hay que describir las acciones que se han de llevar a cabo sobre el modelo según cada uno de los mensajes en la función `update`. Para ello, se utiliza la coincidencia de patrones y las funciones adecuadas para la actualización del modelo. En nuestro ejemplo, correspondería a la actualización de los campos de passengerID, ticketID, por parte de los dos primeros tipos y del registro y el mensaje en el último caso, comprobando si el pasajero pertenece a dicho vuelo o no, y si ya ha subido a bordo.

```
update : Msg -> Model -> (Model, Cmd Msg)
update msg model =
    case msg of
        ChangePassengerID s ->
            ({ model |  passengerID = s }, Cmd.none)

        ChangeTicketID s ->
            ({ model |  ticketID = s }, Cmd.none)

        Checking ->
            case Dict.get model.ticketID model.registry of
                Nothing ->
                    ({ model |  message="Invalid ticketID"}, Cmd.none)
                Just (tinfo, acc) ->
                    if tinfo.passenger /= model.passengerID then
                        ({ model |  message="Unmatched passengerID"}, Cmd.none)
                    else
                        if acc then
                            ({ model |  message=
                                "Passenger has already accesed. Seat:" ++
                                    String.fromInt tinfo.seat}
                            ,Cmd.none)
                        else
                            ({ model |
                                registry =
                                    Dict.insert
                                        model.ticketID
                                        (tinfo, True)
                                        model.registry
                                , message = "welcome aboard. Seat:" ++
                                    String.fromInt tinfo.seat}
                            , Cmd.none)
```

Nótese que, además de devolver un nuevo modelo, también se devuelve un comando (no hemos hablado de ellos porque su uso es extraño e inusual, al menos de forma explícita).

Hemos presentado un mecanismo de manejo de los mensajes, pero no hemos presentado cómo se generan. Al igual que los eventos en JavaScript, hay que establecer cuándo se generan en los elementos de la aplicación. En Elm esto se hace a través de los eventos como atributos, `onInput`, `onClick`, etc. En el caso planteado los campos de los identificadores se actualizarán con la modificación de los elementos `input` correspondientes, y el mensaje de `Checking` se generará al pulsar el botón.

```
view : Model -> Html Msg
view model =
    div []
        [
        div formStyle
            [
            h2 [style "margin" "0 70px"] [text "Flight Registry"]
            , input
                ([ type_ "text",
                    placeholder "ID Ticket",
                    value model.ticketID,
                    onInput ChangeTicketID
                ] ++ inputStyle)
                []
            , input
                ([ type_ "text",
                    placeholder "ID Passenger",
                    value model.passengerID,
                     onInput ChangePassengerID
                ] ++ inputStyle)
                []
            , button
                ([ onClick Checking ] ++ buttonStyle)
                [ text "Check" ]
            ]
        , div [class "message"] [text <| model.message]
        ]
```

Una vez presentados todos los elementos por separado, vamos a exponer algún mecanismo de interacción de los mismos. Para ello es necesario definir una función `main` (es obligatorio adoptar ese nombre), que interrelaciona dichos elementos. Existen distintas alternativas en el módulo `Browser`, pero nosotros vamos a presentar dos de las más utilizadas:

- Arquitectura `sandbox`. Este tipo de arquitectura está destinada a aplicaciones de carácter específico, en las que además no intervienen datos externos. De hecho, no permite la comunicación con otros elementos, sino que todos los datos deben proceder del interior del modelo (o de elementos html como inputs). Esta arquitectura suele ser utilizada en la iniciación al lenguaje, pero cuando se requiere de tareas más avanzadas que requieran la comunicación con JS e incluso otros servidores hay que recurrir a otras arquitecturas más sofisticadas.

- Arquitectura `element`. A diferencia de la anterior, sí puede interactuar con otros elementos, incluido JavaScript, por medio de `flags` y `ports`, que trataremos con más detalle en el siguiente punto.

Tras esto, ya tenemos una aplicación (sencilla) totalmente operativa tal como se muestra en la *figura 3.15*.

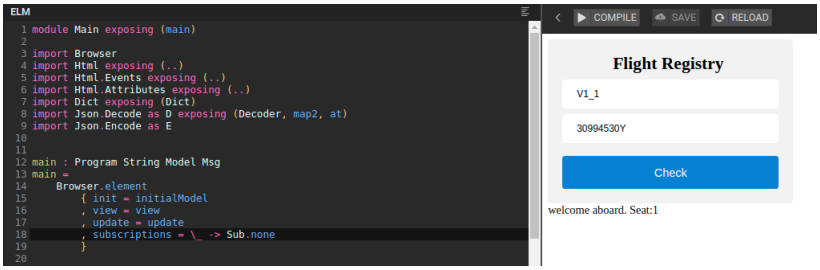


Figura 3.15: Ejemplo de aplicación con Elm Architecture
Fuente propia. Utilizando la herramienta Ellie

Comunicación con JavaScript: Flags y Puertos

Elm ofrece dos mecanismos básicos para la comunicación con JavaScript: *flags* y *ports*.

Las banderas, o `flags`, establecen un método de comunicación entre JavaScript y Elm en la inicialización de la aplicación. Los usos comunes son la transferencia de claves de API, variables de entorno y datos de usuario. Esto puede resultar especialmente útil si se genera el HTML de forma dinámica. Por ejemplo, es común que a través de flags se envíe la información necesaria para extraer los datos de un servidor de base de datos, un servidor Web u otro tipo de servidores.

Todo valor de JavaScript expresable mediante JSON se puede obtener en Elm a través del uso de `flags` y de las herramientas de decodificación del módulo `Json`.

Para ilustrar lo anteriormente expuesto vamos a completar la definición del modelo para el ejemplo del registro de vuelo que dejamos pendiente en su momento. Supóngase que se tiene el registro en JS como un array (nombrado) que posee como claves los identificadores de los billetes, a los cuales asigna como valores otros arrays nombrados con las propiedades del id del pasajero y del número del asiento asignado. Por ejemplo, algo como:

```
const flightReg={
    "V1_1": {"passenger": "30994530Y", "seat":1},
    "V1_2": {"passenger": "87053089K", "seat":2}
}
```

Entonces, si queremos pasarlo a través de flags, hemos de hacerlo con una cadena en formato JSON (utilizando `JSON.stringfy`), para después decodificarlo en Elm. Para ello, el registro se puede representar como un diccionario que usa como claves los identificadores de los billetes (`String`), y como valores objetos (*records*) con las propiedades `passenger` y `seat`. Para decodificar la cadena podemos importar el módulo `JSON.Decode` (con el alias `D`) y hacer uso de sus funciones tal y como mostramos a continuación:

```
decodeRegistry : Decoder (Dict String TicketInfo)
decodeRegistry =
    D.dict (decodeTicketInfo)

decodeTicketInfo : Decoder TicketInfo
decodeTicketInfo =
    map2 TicketInfo
    (at ["passenger"] D.string)
    (at ["seat"] D.int)
```

Finalmente, establecemos el modelo inicial (*initialModel*) con los datos recibidos (no de forma estática como vimos para el contador), de forma que ahora el modelo inicial recibe una *String* y devuelve un modelo (y un mensaje). Así, si hay error aparecerá un mensaje en la aplicación indicando el error en

la lectura, y en caso contrario el modelo será inicializado normalmente, de forma que, junto con los elementos proporcionados previamente, se podrá hacer uso de la aplicación de forma completa. En la *figura 3.16* se muestra tanto la inicialización del modelo como el paso de flags y varios ejemplos del funcionamiento del sistema.

Por otra parte, los *puertos* permiten la comunicación entre Elm y JavaScript y son utilizados, normalmente, a través de *WebSockets*, de forma que este mecanismo puede verse como un sistema de subscripción que, cuando uno de ellos publica por alguno de los canales a los que están suscritos, el otro lo recibe y actúa en consecuencia. Para esta tarea existen dos funciones básicas (en ambos lados de la comunicación), correspondientes a `sendMessage` (mensajes salientes) y `receiveMessage` (mensajes entrantes).

- La función `sendMessage` permite enviar mensajes desde Elm a JS. De forma que JS los recibirá y actuará en consecuencia:

```
-- En el módulo Elm (envío del mensaje)
port sendMessage : String -> Cmd msg


-- En el archivo JS (lectura del mensaje)
app.ports.sendMessage.subscribe(function(message) {
        socket.send(message);
        /*
        const processed = process(message);
        ...
        */
});
```

- La función `messageReceiver` permite enviar mensajes desde JS a Elm. De forma que Elm los recibirá y actuará en consecuencia, como si fuese un evento más:

```
-- En el módulo Elm (lectura del mensaje)
port messageReceiver : (String -> msg) -> Sub msg


-- En el archivo JS (envío del mensaje)
socket.addEventListener("message", function(event) {
    app.ports.messageReceiver.send(event.data);
});
```

Hay que tener en cuenta algunas consideraciones a la hora de utilizar los puertos:

- No es conveniente programar una cantidad enorme de puertos, y de hecho se aconseja su uso solo en situaciones muy concretas. En vez de elevar la cantidad de puertos es preferible elevar la capacidad expresiva de los mismos, enviando mensajes más ricos a través de elementos JSON.

- Los módulos que contienen puertos han de añadir en la cabecera el distintivo `port module` en vez de `module`.

- Los puertos son para aplicaciones. Un módulo de puerto está disponible en aplicaciones, pero no en paquetes. Esto asegura que los autores de aplicaciones tengan la flexibilidad que necesitan, pero el ecosistema del paquete está escrito completamente en Elm.

3.1.7. Consideraciones finales

Hemos mostrado cómo algunas de las características presentes en Elm permiten escribir programas confiables y fáciles de mantener. Gran parte de esa confiabilidad proviene de valores inmutables, funciones puras y un poderoso sistema de tipos. Estas tres particularidades son el núcleo de algunas de las características más importantes del sistema, como son el sistema de módulos y paquetes, o la Elm Architecture.

Aunque hemos dejado algunos aspectos de Elm sin abordar, lo expuesto es suficiente para poder comprender todos los detalles de la implementación que se expondrá en el capítulo de *Implementación*, en el que mostraremos el uso de Elm dedicado tanto al desarrollo de paquetes como al desarrollo de aplicaciones.

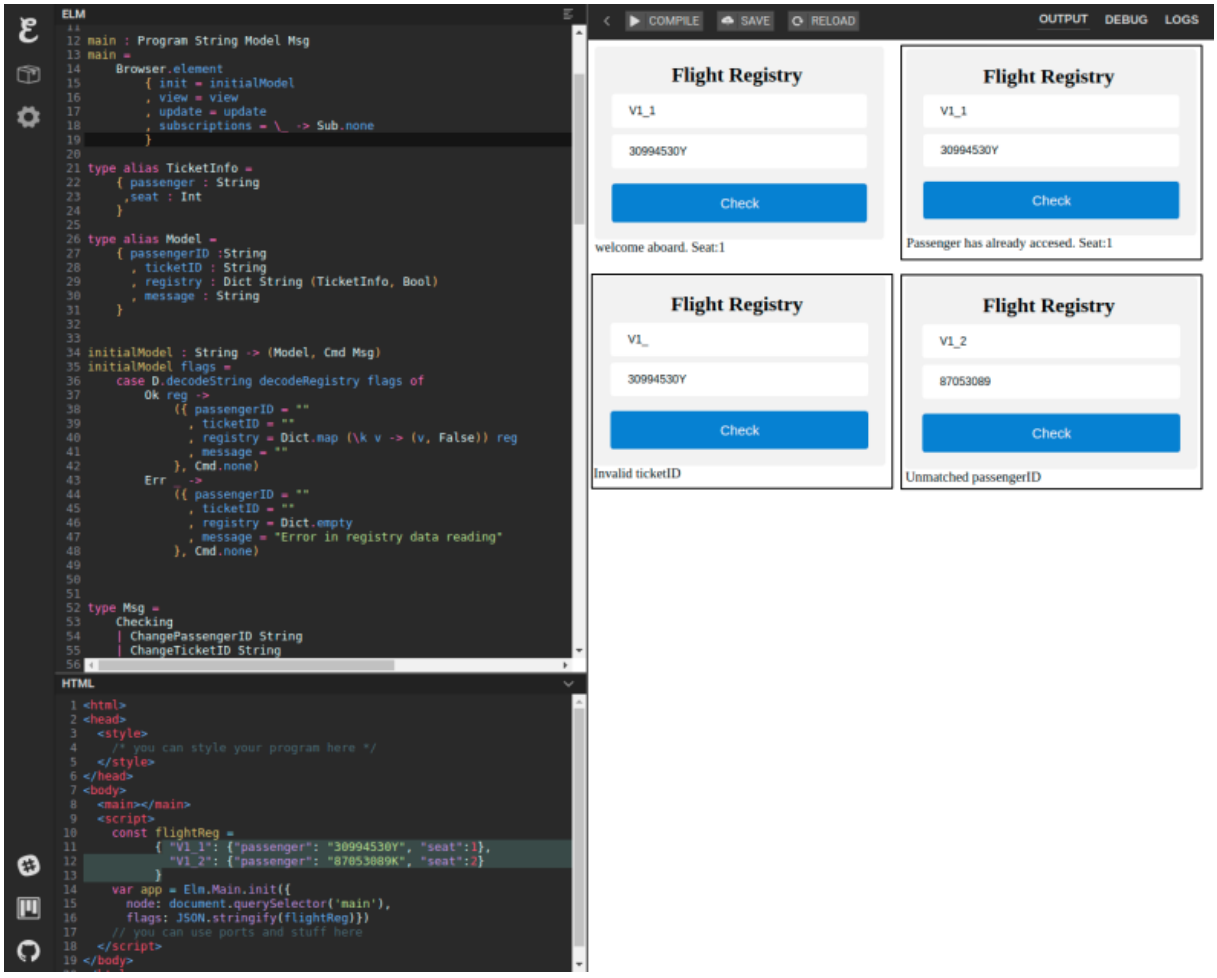


Figura 3.16: Ejemplo de aplicación: Registro de vuelo
Fuente propia. Utilizando la herramienta Ellie

3.2. Litvis (*Literate Visualization*)

A raíz del aumento de desarrolladores e implementaciones y a la emergencia de las colaboraciones entre los mismos, queda patente la necesidad de una correcta visualización y documentación de los procesos de diseño e implementación. Este hecho dio lugar al surgimiento de lo que se conoce actualmente como *Literate Programming*, propuesta por Knuth [5]. Aunque con muchos detractores iniciales debido a una incorrecta concepción, ha sido posteriormente retomada, y en los últimos años se ha hecho patente la tendencia de incorporar en las propias implementaciones la documentación de una forma organizada y

visual en herramientas como Jupyter Notebooks o R-Markdown, entre otras.

En este ámbito, la City University London presentó en el artículo *"Design Exposition with Literate Visualization"* [6] el lanzamiento de una nueva herramienta construida a partir de código Elm y Type Script con la que se pretende motivar a los desarrolladores de, entre otros muchos, el lenguaje Elm al uso de este tipo de herramientas tanto a nivel académico como en producción.

Los propios autores definen esta tecnología como una forma de narración de historias en la que los desarrolladores especifican no solo su implementación, sino también la razón fundamental detrás del diseño de la misma. Por lo tanto, es un acoplamiento estrecho de la narrativa textual y la especificación de implementación más formal, como las instrucciones de programación ejecutables, llevando las propias implementaciones a lo que los autores sugieren como casi una forma de "narrativa computacional", incorporando además la potencia de visualización incluso de forma dinámica e interactiva a través de librerías como elm-Vega o elm-VegaLite.

3.2.1. La estructura de los documentos litvis

Un documento litvis corresponde a un archivo markdown que utiliza algunas características especiales que han de ser renderizadas utilizando el visualizador litvis (*Markdown Preview Enhanced with Litvis*). En él se pueden incluir todos los elementos que se incluyen en un documento markdown pero además incluye algunas características que le otorgan una mayor potencia de trabajo, sobre todo en la posibilidad de crear bloques de código Elm ejecutables y renderizables para la visualización de texto y figuras, incluso de forma interactiva.

Cabecera Litvis

Para poder ejecutar estos bloques de código es necesario incorporar al inicio del documento una pequeña cabecera en formato *yaml* que recoge la información sobre los paquetes importados en el documento de forma análoga a como lo hace Elm en el archivo de configuración del paquete (*elm.json*).

```
---
elm:
  dependencies:
    package : version
    package : version

  source-directories:
    - ...
    - ...
---
```

Código 3.3: Cabecera de un documento Litvis

Esta cabecera consta de dos apartados:

- *dependencies*. En él se indican los paquetes (públicos del repositorio de Elm) que se van a utilizar así como la versión de los mismos. Para la versión se puede utilizar la palabra clave *latest*, que indica que se toma la última versión publicada del mismo.

- *source-directories*. Este apartado es de carácter optativo y se utiliza para indicar el origen de los módulos utilizados en el documento que no han sido publicados. Es común hacer uso de este apartado en alguna etapa de desarrollo o en implementaciones de apartados específicos que no quieran ser publicados.

Una vez establecida la cabecera ya podemos hacer uso de los bloques de código ejecutables, chunks. Aunque existen varios tipos de chunks, vamos a comentar los tres principales:

Literate Elm Chunks

Corresponden a bloques de código Elm ejecutables. Al igual que otros bloques de código (chunk) de markdown-litvis, se abre y se cierra con tres comillas invertidas (‘‘‘). La referencia del lenguaje *elm* debe seguir inmediatamente a las comillas inversas sin espacio. Los argumentos que determinan el comportamiento del bloque deben colocarse encerrados por llaves, separados por espacios y sin saltos de línea.

```
‘‘‘elm {...attributes}
...
‘‘‘
```

Código 3.4: Literate Elm Chunks

Los atributos han de ser descritos de acuerdo a las siguientes reglas:

- Los atributos que corresponden a pares atributo-valor son especificados como *tag=value*. La aparición solamente de la etiqueta, sin valor asociado, es considerada como *tag=True*.
- No se admiten espacios ni antes ni después de =, sino que ha de corresponder estrictamente al patrón descrito *tag=value* (o, según el caso, solo *tag*).
- Los corchetes denotan un array de valores, que van separados por coma. Si el valor contuviese comas, espacios o punto y coma debe utilizarse la barra invertida como símbolo de escape (e.g.‘ ‘).
- Los valores pueden ir entre comillas (”, ’, ‘) habilitando que contengan espacios o algunos caracteres de control como ‘=‘, ‘[ ]‘ o ‘( )‘. También se pueden utilizar las comillas dentro de los valores, pero han de ir precedidas por la barra invertida de escape.

Los atributos permitidos en los elm-chunks corresponden a :

- ‘l‘ (o ‘literate‘) es un indicador de un bloque de código alfabetizado, que lo diferencia de un bloque de código de markdown standard. De forma predeterminada, cualquier declaración de Elm es accesible en otros bloques de código alfabetizados dentro del documento. Si ‘l = hidden‘, el código aún se evalúa pero su fuente no se muestra en la salida renderizada. Esto es útil para el código de configuración, como declaraciones de importación y casos en los que la implementación no es fundamental para la narrativa.
- ‘r‘ (o‘ raw‘) indica que se espera que el bloque de código imprima el valor bruto de una función o funciones. Esto puede ser útil para ’Elm alfabetizado’ donde se mostrará el valor generado por una función. Si no se asigna ningún valor a ‘r‘, se renderiza la última variable definida en el bloque de código.
- ‘m‘ (o‘ markdown‘) funciona de manera similar a ‘r‘ pero la salida sin procesar se interpreta como markdown. Esto permite que una función elm genere una salida formateada, por ejemplo con el uso de ’...’ para la representación de las fórmulas como latex.
- ‘j‘ (o‘ json‘) funciona igual que ‘r‘ /‘ raw‘ excepto que el valor se analiza como JSON y se formatea.
- ‘v‘ (o‘ visualize‘) indica que se espera que el bloque de código genere algún resultado en este punto del documento. El formato de esta salida está determinado por el contenido de un símbolo o una expresión a representar. Actualmente, las especificaciones Vega-Lite y Vega generadas por elm-vegalite y elm-vega son compatibles. Si no se asigna ningún valor a ‘v‘, se utiliza el último símbolo definido en el bloque de código.
- ‘context‘ es un atributo que permite el aislamiento de código dentro de un documento. Los bloques de código en diferentes contextos funcionan en paralelo y no comparten importaciones ni declaraciones de símbolos. Todos los bloques pertenecen a un contexto “predeterminado” implícito si el atributo no está definido. Los contextos se evalúan de forma independiente, lo que reduce la propagación de los errores de compilación de Elm y los conflictos de espacios de nombres (en un mismo contexto no pueden aparecer dos declaraciones para la misma variable).

- ‘id’ asigna un identificador a un bloque de código para que pueda ser referenciado en otros bloques de código (ver ‘follows’).
- ‘follows’ se puede usar en el primer bloque de código de un nuevo contexto para derivar de un contexto existente. Esto hace que las declaraciones e importaciones definidas antes del bloque referenciado y de su mismo contexto sean heredadas por el nuevo contexto. Muy útil, por ejemplo, para definir una única vez los *imports* en vez de tener que hacerlo al inicio de cada contexto.
- Otros :‘i’, ‘s’, ‘interactive’. Puede consultarse su uso detallado en la documentación del propio proyecto gicentre/litvis [7].

Además hay que tener en cuenta dos cuestiones adicionales:

- El orden de ‘l’,‘ v’, ‘r’,‘ m’ o ‘j’ en los atributos determina el orden de representación.
- No es necesario definir ‘l’ para alfabetizar el bloque de código si ya se han dado ‘v’, ‘r’, ‘m’ o ‘j’ (esto implica ‘l=hidden’, sí es necesario poner ‘l’ si se desea que sí sea visible).

Referencias ^^^

Renderizan el resultado de la llamada en cualquier parte del documento markdown-litvis. Permiten la representación en línea o llamadas repetidas a la misma representación dentro de un documento. Se indican encerrados entre tres símbolos ^^^y usan la misma sintaxis de atributos que los bloques anteriores.

```
^^^elm {tag=(...) attributes}^^^
```

Código 3.5: Triple Hat Reference

Los atributos permitidos en este tipo de bloques corresponden a:

- ‘v’ (o ‘visualize’) indica qué símbolos y expresiones renderizar. En el escenario más común, el valor es una única constante Elm de tipo *Spec*. Se especifica como ‘v = myChart’.
- ‘r’ (o ‘raw’) indica qué símbolos y expresiones generar sin formato. Se especifica como ‘r = myVar’ o también con funciones ‘r=(function ...)’.
- ‘m’ (o ‘markdown’) indica los símbolos que se generarán como markdown para formatearlos. Se especifica como ‘m = myVar’ o también con funciones ‘m=(function ...)’.
- ‘j’ (o ‘json’) indica qué símbolos y expresiones generar como formato JSON. Se especifica como ‘j= myVar’.
- ‘context’ es un atributo opcional para especificar a qué contexto de bloque de código se debe hacer referencia. Se asume el contexto ‘predeterminado’ si el atributo no está definido.
- ‘interactive’ (o ‘interactive=True’) hace que las especificaciones visualizadas sean interactivas si corresponde.

El orden de ‘v’, ‘r’, ‘m’ y ‘j’ determina el orden de la salida aunque en este tipo de chunks se considera una buena práctica evitar múltiples formatos de salida y especificaciones para controlar mejor el diseño de la salida.

Diagramas

Markdown Preview Enhanced admite la representación de diagramas de flujo, diagramas de secuencia, sirena, PlantUML, WaveDrom, GraphViz, Vega y Vega-lite, y Ditaa. También puede renderizar TikZ, Python Matplotlib, Plotly y todo tipo de gráficos y diagramas utilizando Code Chunk.

Al igual que otros bloques de código (chunk) de markdown-litvis, se abre y se cierra con tres comillas invertidas (“‘). La referencia del lenguaje correspondiente debe seguir inmediatamente a las comillas

inversas sin espacio. Los argumentos que determinan el comportamiento del bloque deben colocarse encerrados por llaves, separados por espacios y sin saltos de línea. Cada lenguaje posee una serie de atributos que pueden consultarse en la documentación correspondiente [8].

```
```dot{...attributes}
...
```
```

Código 3.6: DOT diagram chunk

3.2.2. Uso de la herramienta en el ámbito del proyecto

Las características que hemos visto, así como su libre distribución y facilidad de instalación (basta una simple instalación de Elm, nodejs y VScode (o Atom) con sus correspondientes extensiones), hacen de esta herramienta el entorno ideal para poder desempeñar la gran parte de las actividades formativas que hasta ahora se vienen impartiendo en la asignatura de Lógica Informática de los distintos Grados que se imparten en el departamento, de forma que se puedan proporcionar los contenidos de forma teórico-práctica con una única herramienta sin limitar la capacidad expresiva de la documentación ni la potencia de ejecución de un lenguaje funcional como Elm.

En el *Capítulo 5* veremos algunos ejemplos concretos de uso que permitan entender mejor la relevancia de este sistema dentro del proyecto LogicUS.

4 | Implementación de LogicUS

En este capítulo presentaremos de forma detallada las implementaciones más relevantes de cada uno de los módulos, especialmente aquellas referentes a la sintaxis, la semántica y los algoritmos de decisión.

4.1. Los componentes de LogicUS

4.2. Implementación del Paquete LogicUS

4.2.1. Visión general de LogicUS

Las librerías LogicUS conforman la base de cálculo del software desarrollado, donde se ha realizado las implementación de las definiciones de todos los tipos necesarios, de las funciones requeridas para su manipulación, y de los algoritmos de decisión que permiten trabajar tanto con la Lógica Proposicional como con la Lógica de Primer Orden.

Dada la clara diferenciación entre ambas se ha decidido organizar el paquete en 2 secciones, LogicUS.PL (para el trabajo con la Lógica Proposicional) y LogicUS.FOL (para el trabajo con la Lógica de Primer Orden). Dentro de cada uno de ellos se han desarrollado una serie de módulos de dominio específico que se organizan de forma similar a como hemos presentado el desarrollo formal de la lógica, de forma que ambos siguen una estructura similar, con un módulo base que implementa la sintaxis y semántica de la correspondiente Lógica y algunos módulos específicos que implementan cuestiones particulares tales como la transformación de las fórmulas (a formas normales, por ejemplo) o algunos algoritmos de decisión concretos.

Cada uno de los módulos cuenta con varios elementos que podemos distinguir en:

- **Tipos**. En el desarrollo se requerirán distintas estructuras que permitan almacenar la información relativa a las fórmulas y conjuntos de fórmulas, interpretaciones, clases de fórmulas, grafos de desarrollo de algoritmos, sustituciones, ...
- **Funciones operativas**. Las que realicen el trabajo con las fórmulas y conjuntos, transformaciones, evaluaciones, deducciones, resolventes, sustituciones, extracción de modelos, ...
- **Funciones de representación**. Además de las funciones operativas se han implementado una serie de métodos que permitan crear representaciones de los distintos elementos, incluyendo representaciones en formato cadena, en formato matemático (Latex) e incluso visualizable (por ejemplo, para los grafos).
- **Funciones de parser**. Además en los módulos base se han implementado una serie de funciones que permitan la interacción directa con el usuario y acomoden la definición de instancias a través del uso de cadenas.

Antes de abordar el contenido de cada uno de los módulos de forma detallada, presentamos una visión general, tanto de la estructura del proyecto como de la organización y contenido de los mismos.

Archivo de configuración: *elm.json*

En este archivo se detalla toda la configuración del paquete, incluyendo como elementos destacados los módulos expuestos y las dependencias con otros paquetes, así como la versión de los mismos. Aunque se ha ido actualizando a lo largo del proyecto, el contenido final del mismo corresponde al presentado en el *código 4.1*

En cuanto a los módulos expuestos, se presenta la división en secciones previamente comentada, así como los módulos incluidos en cada una de las mismas y que, como veremos, refleja claramente la estructura de directorios que debe tener nuestro proyecto bajo el directorio raíz *src/*.

```
{
  "type": "package",
  "name": "vicramgon/logicus",
  "summary": "Elm packages for working with
    Propositional and First Order Logic algorithms.",
  "license": "BSD-3-Clause",
  "version": "7.2.0",
  "exposed-modules": {
    "Propositional Logic": [
    "LogicUS.PL.SintaxSemantics",
    "LogicUS.PL.SemanticTableaux",
    "LogicUS.PL.NormalForms",
    "LogicUS.PL.Clauses",
    "LogicUS.PL.DPLL",
    "LogicUS.PL.Resolution",
    "LogicUS.PL.CSP"
    ],
    "First Order Logic": [
    "LogicUS.FOL.SintaxSemantics",
    "LogicUS.FOL.SemanticTableaux",
    "LogicUS.FOL.NormalForms",
    "LogicUS.FOL.Clauses",
    "LogicUS.FOL.Herbrand",
    "LogicUS.FOL.Unification",
    "LogicUS.FOL.Resolution"
    ]
  },
  "elm-version": "0.19.1 <= v < 0.20.0",
  "dependencies": {
    "elm/core": "1.0.5 <= v < 2.0.0",
    "elm/parser": "1.1.0 <= v < 2.0.0",
    "elm-community/graph": "6.0.0 <= v < 7.0.0",
    "elm-community/intdict": "3.0.0 <= v < 4.0.0",
    "elm-community/list-extra": "8.3.0 <= v < 9.0.0",
    "elm-community/maybe-extra": "5.2.0 <= v < 6.0.0",
    "elm-community/string-extra": "4.0.1 <= v < 5.0.0"
  },
  "test-dependencies": {}
}
```

Código 4.1: Archivo de configuración del paquete LogicUS

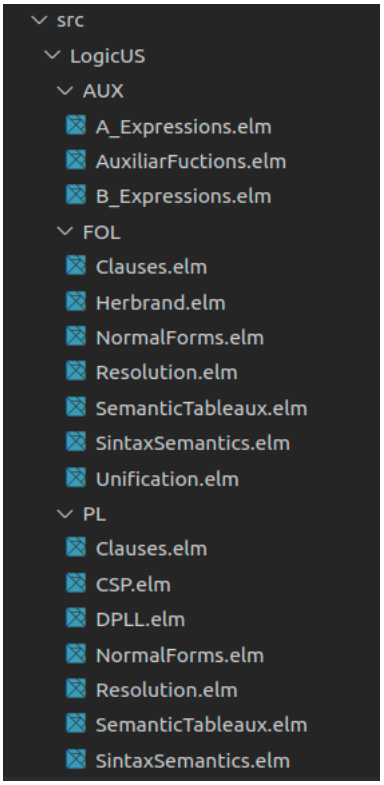


Figura 4.1: Estructura del paquete LogicUS

Estructura y contenido del proyecto

Ajustándonos a lo expuesto previamente, la estructura del proyecto está organizada tal como se muestra en la *figura 4.1*. El directorio *src/* consta de un subdirectorio *LogicUS/* que contiene a su vez dos subdirectorios: *PL/* y *FOL/*, que albergan los módulos correspondientes de la Lógica Proposicional y la Lógica de Primer Orden condicionando que los nombres de los módulos correspondan precisamente a *LogicUS.PL.** y *LogicUS.FOL.**, respectivamente.

A continuación mostraremos los contenidos de cada una de las secciones y módulos de forma detallada, explicitando las implementaciones más relevantes en cada uno de ellos, con fin de exponer con claridad las funcionalidades del sistema y justificar las decisiones tomadas en su implementación.

4.2.2. Implementación de LogicUS.PL

Como ya se ha señalado, esta sección del paquete está dedicada al trabajo con la Lógica Proposicional, y a tal efecto se ha dotado al sistema con las prestaciones necesarias para el objetivo marcado.

Visión global de LogicUS.PL

A modo de visión general, se expone una breve descripción de las principales funcionalidades y contenidos incluidos en las mismas:

- **LogicUS.PL.SintaxSemantics**: Constituye la base de la Lógica Proposicional, donde se expone la sintaxis para la definición de las fórmulas y la semántica para la interpretación de las mismas. Además, se encuentra implementado un Parser que permite definir las fórmula en una notación más cercana al formalismo lógico y una función de representación para las mismas así como para los principales resultados obtenidos en las funciones, tanto en versión unicode como en versión Latex.
- **LogicUS.PL.SemanticTableaux**: Desarrolla todas las herramientas necesarias para trabajar con tableros semánticos, distinguiendo los diferentes tipos de fórmulas y reglas, y aportando, además, herramientas para la visualización del tablero completo.
- **LogicUS.PL.NormalForms**: Contiene las funciones necesarias para la transformación de fórmulas en formas normales (negativa, conjuntiva y disyuntiva).
- **LogicUS.PL.Clauses**: Proporciona algunas funciones que permiten trabajar con cláusulas proposicionales: definición, operaciones, transformación de fórmulas, conjuntos clausales, parsers, representación, etc.
- **LogicUS.PL.DPLL**: Define las funciones necesarias para la aplicación del algoritmo de resolución DPLL a conjuntos de cláusulas proposicionales, así como la búsqueda de modelos basada en esta técnica. Además, aporta las herramientas necesarias para la visualización del tablero completo.
- **LogicUS.PL.Resolution**: Define las funciones para trabajar con los algoritmos de resolución implementando diferentes estrategias: saturación, regular, mejor primero, lineal, positivo, negativo, unitario, por entradas, etc.

Visto esto, y antes de pasar a detallar los contenidos de cada uno de los módulos, vamos a exponer la organización y dependencias internas entre los módulos comentados previamente. La *figura 4.2* presenta un diagrama de las dependencias internas de dichos módulos, donde queda patente la estructura comentada con anterioridad, en la que el módulo *LogicUS.PL.SintaxSemantics* es la base sobre la que actúan el resto de los módulos. Además, se expone también la importancia del módulo *LogicUS.PL.NormalForms*, encargado de la representación de las cláusulas, e imprescindible para los algoritmos DPLL y Resolución Proposicional.

4.2.3. LogicUS.PL.SintaxSemantics

Como hemos señalado, es el módulo que establece la base sobre la que construir la implementación del formalismo proposicional, definiendo las estructuras que almacenan las fórmulas proposicionales y sus interpretaciones, y proporciona las funciones auxiliares para las funciones de evaluación, generación de árboles de formación, tablas de verdad, etc.

Sintaxis de fórmulas en LogicUS

Comenzaremos presentando la sintáxis y definiendo las estructuras que almacenarán las fórmulas y que permitirán trabajar con ellas. Lo más habitual es que la definición de las fórmulas proposicionales venga dada de forma recursiva. En los lenguajes funcionales, como Haskell o Elm, este tipo de estructuras

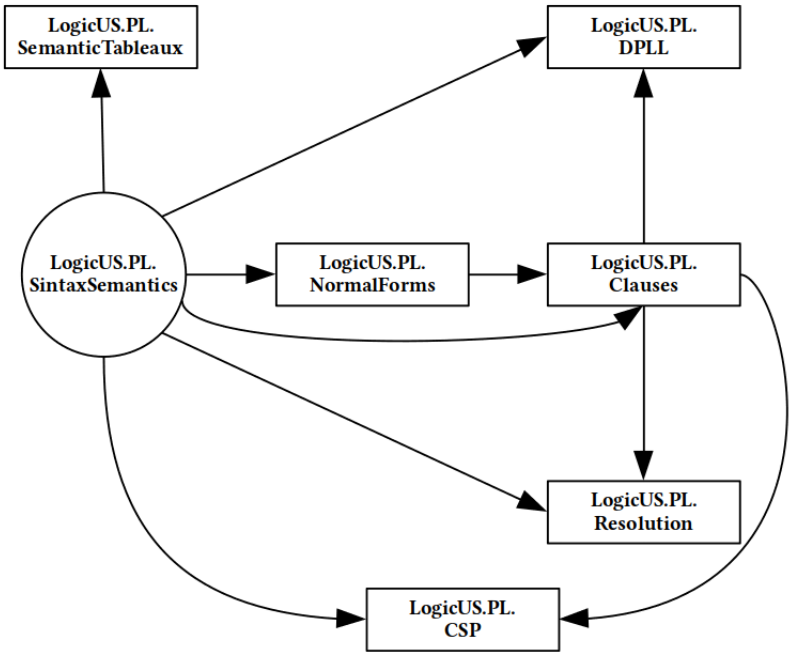


Figura 4.2: Dependencias internas de los módulos de LogicUS.PL

son muy corrientes y, de hecho, son las estructuras básicas que almacenan muchos de los distintos tipos básicos. Así pues, teniendo presente la definición recursiva de las fórmulas a partir de átomos como elementos fundamentales y las conectivas como modificadores y nexos entre los átomos, podemos definir las fórmulas proposicionales como:

```
type alias PSymb = (String, List Int)

type FormulaLP =
    Atom PSymb
    | Neg FormulaLP
    | Conj FormulaLP FormulaLP
    | Disj FormulaLP FormulaLP
    | Impl FormulaLP FormulaLP
    | Equi FormulaLP FormulaLP
    | Insat
    | Taut
```

Código 4.2: Estructura de las Fórmulas en LogicUS

Siguiendo una total analogía entre la estructura presentada en el formalismo y la definida en el lenguaje, donde el elemento fundamental corresponde a un átomo, construido a partir del constructor `Atom`, y por un símbolo proposicional (de tipo `String`). Las conectivas quedan representadas por los constructores correspondientes a partir de fórmulas proposicionales (según la aridad).

Para definir una fórmula sería suficiente con utilizar los constructores de forma anidada y prefija para crear la fórmula. Por ejemplo, la fórmula $p \rightarrow q \vee \neg r$ sería descrita en el sistema como:

```
Impl (Atom "p") (Disj (Atom "q") (Neg (Atom "r")))
```

Sin embargo, la notación infija y sin necesidad de acudir a los constructores parece, a priori, una alternativa más cómoda. Al contrario que Haskell, y aunque sí lo permitía en versiones anteriores, con Elm 0.19 la opción de definir operadores infijos está deshabilitada, por lo que si se quiere hacer uso de los mismos es necesario construir un Parser. El módulo Parser de Elm define una serie de funciones

para la creación de parsers. A continuación detallaremos la implementación del mismo. Esta parte puede resultar un poco ardua y no es indispensable para entender el funcionamiento del sistema completo, por lo que tampoco es esencial su comprensión absoluta.

Así, comencemos estableciendo una serie de reglas y viendo cómo trasladar sus requisitos a la implementación en Elm.

En primer lugar, un símbolo proposicional debe corresponder a una cadena de caracteres alfabéticos (comenzando en minúscula, se recomienda seguir la notación camelCase) acompañada opcionalmente de uno o más subíndices, indicados entre '_{' y '}', y separados por comas. Si acudimos a la notación en forma de expresión regular el criterio correspondería a `[a-z][a-zA-Z]*(_{[0-9]+(,[0-9]+)*)?}`.

La implementación en Elm se correspondería con el código mostrado en el *código 4.3*. La función `Parser.oneOf` permite establecer las distintas alternativas para la lectura: los símbolos con subíndice y los que no lo tienen. Al igual que ocurre con muchos otros Parsers, Elm decide qué rama tomar como aquella con la que coincida el primer carácter de la cadena, lo que puede presentar un problema en caso de ambigüedad, por ello, la función `Parser.backtrackeable` permite que el parser pueda recuperarse de una rama mal elegida.

Así pues, leeremos la variable como indexada y, en caso fallido, intentaremos leerla como variable simple, obteniendo cadena del símbolo, y si procede, la lista de índices.

```
plVarParser : Parser ( String, List Int )
plVarParser =
  Parser.oneOf
  [ Parser.succeed identity
      |= Parser.backtrackable plVarIdentifierSubindexedParser
  , Parser.succeed (\x -> ( x, [] ))
      |= plVarNameParser
  ]

plVarNameParser : Parser String
plVarNameParser =
  Parser.succeed ()
    |. Parser.chompIf Char.isLower
    |. Parser.chompWhile Char.isLower
    |> Parser.getChompedString

plVarIdentifierSubindexedParser : Parser ( String, List Int )
plVarIdentifierSubindexedParser =
  Parser.succeed Tuple.pair
    |= plVarNameParser
    |= Parser.sequence
      { start = "_{"
      , separator = ","
      , end = "}"
      , spaces = Parser.spaces
      , item = Parser.int
      , trailing = Forbidden
      }
```

Código 4.3: Implementación del Parser de Símbolos Proposicionales

Una vez descrito el reconocimiento de los símbolos proposicionales, hagamos lo mismo con las conectivas. Comenzaremos con las conectivas binarias (que se describen habitualmente en notación infija) y dejaremos para más adelante la conectiva de negación, que se mantiene en notación prefija y por tanto el tratamiento es diferente. Como suele ser habitual en muchos sistemas de lógica, con el fin de facilitar la introducción de las fórmulas con el teclado, vamos a establecer la siguiente asociación entre los símbolos de representación y las conectivas:

$\wedge \equiv$ `&` $\quad \vee \equiv$ `|` $\quad \rightarrow \equiv$ `->` $\quad \leftrightarrow \equiv$ `<->`

Para definir estas asociaciones se ha creado un tipo `Operator` que representa estas cuatro conectivas, y después asociamos a dichos operadores los símbolos correspondientes (en este aspecto Elm sí es capaz

de capturar símbolos de varios caracteres). En el *código 4.4* se muestra la implementación realizada de acuerdo con el criterio presentado.

```
type Operator
  = AndOp | OrOp | ImplOp | EquivOp

operator : Parser Operator
operator =
Parser.oneOf
  [ Parser.succeed AndOp
    |. Parser.symbol "&"
  , Parser.succeed OrOp
    |. Parser.symbol "|"
  , Parser.succeed ImplOp
    |. Parser.symbol "->"
  , Parser.succeed EquivOp
    |. Parser.symbol "<->"
  ]
```

Código 4.4: Implementación del Parser de conectivas binarias

Queda por describir la estructura de las fórmulas (y la conectiva de negación), así como las reglas de prioridad y de asociación entre las conectivas. Para ello, tengamos en cuenta que al procesar la cadena el Parser no conoce de antemano los elementos que van a venir después, de forma que para realizar una correcta lectura es necesario leer todas las partes y realizar posteriormente un postprocesamiento.

Lo primero, lógicamente, es asegurar que se lee toda la cadena y no sólo una parte de ella. Para ello, la función `Parser.end` indica que la cadena debe terminar (es decir, que no hay más caracteres por leer en la misma). Formalmente, para facilitar el análisis de las fórmulas se suele exigir que todas las fórmulas y subfórmulas deben ir entre paréntesis, lo que resuelve el problema de la ambigüedad y la prioridad, pero convierte la tarea de escribir fórmulas en una labor tediosa e indeseable para nuestro uso, por lo que se van a definir las reglas de prioridad y de asociación, aunque sí se mantendrán los paréntesis externos de la fórmula por cuestiones técnicas, pero serán añadidos por la función de lectura y el usuario podrá prescindir de ellos desde un punto de vista práctico.

La función `fplParser` simplemente exige que se lea hasta el final la cadena, por lo que podemos pasar a `fplParserAux`, que viene dado con una definición por casos:

- si lee el símbolo ¬ o el símbolo `-` entonces devuelve la negación de la fórmula que sea leída a continuación;
- si lee el símbolo `!F` entonces devuelve la fórmula insatisfactible;
- si lee el símbolo `!T` entonces devuelve la fórmula válida (tautología);
- si lee un símbolo de variable, devuelve un átomo;
- y si lee un paréntesis entonces leerá la fórmula que se encuentre dentro de ella.

Es precisamente en el último punto donde está la necesidad de los paréntesis externos, si no estuviesen, el Parser intentaría leer la negación, el átomo, la formula insatisfactible o la fórmula válida, de forma que una vez leído el término correspondiente no seguiría leyendo y por tanto devolvería error. Si bien esto se puede solucionar mediante elementos `backtrackeable` el esfuerzo de lectura sería mucho mayor que simplemente introduciendo esta pequeña modificación, que ni siquiera el usuario llegará a notar ya que, independientemente de que haya puesto o no los paréntesis externos, el programa los añadirá por seguridad, ya que su inclusión no afecta a la corrección en la lectura de las fórmulas.

```
fplParser : Parser FormulaPL
fplParser =
  Parser.succeed identity
    |= fplParserAux
    |. Parser.end


fplParserAux : Parser FormulaPL
fplParserAux =
  Parser.oneOf
    [ Parser.succeed identity
      |. Parser.symbol "("
      |= Parser.lazy (\_ -> expression)
      |. Parser.symbol ")"
    , Parser.succeed Neg
        |. Parser.oneOf
                  [ Parser.symbol "¬"
                  , Parser.symbol "-"
                  ]
      |= Parser.lazy (\_ -> fplParserAux)
    , Parser.succeed Insat
      |. Parser.symbol "!F"
    , Parser.succeed Taut
      |. Parser.symbol "!V"
    , Parser.succeed Atom
      |= plVarParser
    ]


expression : Parser FormulaPL
expression =
  fplParserAux |> Parser.andThen (expressionAux [])



expressionAux : List ( FormulaPL, Operator ) -> FormulaPL -> Parser FormulaPL
expressionAux revOps expr =
  Parser.oneOf
  [ Parser.succeed Tuple.pair
    |= operator
    |= fplParserAux
    |> Parser.andThen
      (\( op, newExpr ) -> expressionAux (( expr, op ) :: revOps) newExpr)
  , Parser.lazy (\_ -> Parser.succeed (finalize revOps expr))
  ]
```

Código 4.5: Implementación del Parser de fórmulas I

Como estamos construyendo un Parser recursivo, en el caso de la negación hemos de leer una nueva fórmula que será modificada por la negación. La función `Parser.lazy` cumple precisamente con esta función, la de permitir la definición de Parser recursivos, de forma que se indica que lea una formula mediante el propio parser que hemos definido. Además, nótese que si lo que lee después es un átomo entonces devolverá la negación de dicho átomo, pero si lee un paréntesis entonces primero resolverá el paréntesis y después aplicará la negación, justo como definimos en el propio formalismo de la lógica, por tanto, en el propio parser se define la prioridad de la conectiva de negación como prioridad máxima sólo por detrás de los paréntesis. Desarrollemos ahora la lectura de fórmulas dentro de paréntesis.

Cuando leemos un paréntesis tenemos que leer la fórmula del interior, entonces ¿por qué no llamamos directamente a la función como hacíamos con la negación? Nótese que no hemos definido la lectura de las conectivas dentro del parser, al tratarse de notación infija hemos de leer primero uno de los operandos y después el símbolo de la conectiva, que es precisamente lo que hacen las funciones `expression` y `expresionAux`. Cuando se leen los paréntesis se recurre a estas funciones. La función `expression` (apoyada en el uso recursivo de `expressionAux`) leerá el primer elemento de la fórmula (el primer operando de una operación binaria), después el símbolo de la conectiva, y por último el segundo operando.

```
finalize : List ( FormulaPL, Operator ) -> FormulaPL -> FormulaPL
finalize revOps finalExpr =
case revOps of
[] ->
finalExpr

( expr, AndOp ) :: otherRevOps ->
finalize otherRevOps (Conj expr finalExpr)

( expr, OrOp ) :: ( expr2, AndOp ) :: otherRevOps ->
Disj (finalize (( expr2, AndOp ) :: otherRevOps) expr) finalExpr

( expr, OrOp ) :: otherRevOps ->
finalize otherRevOps (Disj expr finalExpr)

( expr, ImplOp ) :: ( expr2, AndOp ) :: otherRevOps ->
Impl (finalize (( expr2, AndOp ) :: otherRevOps) expr) finalExpr

( expr, ImplOp ) :: ( expr2, OrOp ) :: otherRevOps ->
Impl (finalize (( expr2, OrOp ) :: otherRevOps) expr) finalExpr

( expr, ImplOp ) :: otherRevOps ->
finalize otherRevOps (Impl expr finalExpr)

( expr, EquivOp ) :: ( expr2, AndOp ) :: otherRevOps ->
Equi (finalize (( expr2, AndOp ) :: otherRevOps) expr) finalExpr

( expr, EquivOp ) :: ( expr2, OrOp ) :: otherRevOps ->
Equi (finalize (( expr2, OrOp ) :: otherRevOps) expr) finalExpr

( expr, EquivOp ) :: ( expr2, ImplOp ) :: otherRevOps ->
Equi (finalize (( expr2, ImplOp ) :: otherRevOps) expr) finalExpr

( expr, EquivOp ) :: otherRevOps ->
finalize otherRevOps (Equi expr finalExpr)
```

Código 4.6: Implementación del Parser de fórmulas II

Vamos a explicar el funcionamiento de las funciones definidas en el *código 4.6* por medio de un ejemplo. Para ello, consideremos:

$$a \to b \wedge c \leftrightarrow d \vee e \cdots \implies a \boxed{\to b} \boxed{\wedge c} \boxed{\leftrightarrow d} \boxed{\vee e} \cdots$$

Se puede observar cómo para leer la fórmula se puede hacer a partir del primer elemento y de sucesivos pares *(conectiva, fórmula)*, que corresponde precisamente al comportamiento de `expressionAux`, procesándose los elementos leídos una vez no queden más por leer. Si no tuviésemos en cuenta ninguna prioridad para las conectivas y asociásemos las fórmulas por la izquierda, entonces sería muy fácil su procesamiento sin más que ir aplicando la conectiva del par correspondiente al elemento resultante y la siguiente tupla partiendo como base del primer elemento leído. Sin embargo, sí hemos de tener en cuenta ese orden de prioridad, por lo que la lista de los elementos y el elemento base son procesados por la función `finalize`, cuyo funcionamiento es similar al comentado anteriormente pero teniendo en cuenta el orden de prioridad y, en caso de igual prioridad, asociando por la derecha.

En el mismo código se puede observar la parte de la función que añade los paréntesis para la lectura y devuelve una tupla *(Formula, Cadena leída, Mensaje Error)*, lo que permite declarar las fórmulas exactamente igual a como se hace en el formalismo de la lógica proposicional considerado. Por ejemplo:

$$a \wedge (\neg b \vee c) \to a \implies \texttt{" a \& (¬ b | c) -> a "}$$

Tras la definición de fórmulas individuales, LogicUS representa los conjuntos de fórmulas como listas.

```
type alias LPSet = List FormulaLP
```

Código 4.7: Tipo LPSet en LogicUS

Por ejemplo, el conjunto:

$$M = \{(p \wedge q) \vee (p \wedge r), (p \wedge r) \vee (\neg p \wedge q) \rightarrow \neg q, (p \leftrightarrow q) \wedge (p \rightarrow \neg q) \wedge p\}$$

se define como:

```
f1 = fplReadExtraction <| fplReadFromString "p & q | p & r"
f2 = fplReadExtraction <| fplReadFromString " p & r | ¬ p & q -> ¬ q"
f3 = fplReadExtraction <| fplReadFromString "(p <-> q) & (q -> p) & p "

m = [f1, f2, f3]
```

Código 4.8: Definición de LPSets en LogicUS

También se proporcionan dentro del marco de la sintaxis herramientas para representación de los árboles de formación tanto en formato de cadena como en formato DOT, pero la definición de las mismas no es de gran interés (veremos después otros algoritmos con el manejo de grafos más interesantes) por lo que no las presentaremos explícitamente, al igual que las funciones de representación de las fórmulas y conjuntos tanto en formato de cadena como en formato Latex. Todas las funciones expuestas en los módulos pueden consultarse en el *LogicUS.PL.SintaxSemantics* del paquete publicado.

Tras las cuestiones relacionadas con la implementación sintáctica pasaremos en el siguiente punto a abordar la implementación de los elementos semánticos de la Lógica Proposicional.

Semántica Proposicional en LógicUS

Los *valores de verdad* corresponden a 1 (verdadero) y 0 (falso). Elm ya provee esos valores booleanos en el tipo básico `Bool`, por lo que no es necesario realizar ninguna definición alternativa para ellos. La definición de las *funciones de verdad* asociadas a las conectivas corresponden directamente a la aplicación de dichas funciones en la evaluación de las fórmulas (la veremos un poco más adelante). Pasemos entonces a presentar la implementación de las interpretaciones.

Una *interpretación* (restringida al ámbito de una fórmula, o conjunto de fórmulas, concreta) es una asociación de un valor de verdad a cada variable proposicional presente en la fórmula. Esta asociación se puede implementar directamente con los diccionarios como `Dict String Bool`. Sin embargo, teniendo en cuenta que los valores booleanos solo son dos, existe una posibilidad más compacta, donde una interpretación se representa por medio de una lista de símbolos proposicionales de forma que se considerarán verdaderas en la interpretación aquellas variables que aparezcan en la lista y se considerarán falsas aquellas que no aparezcan. Por ejemplo, la interpretación $\{p : 1, q : 0, r : 0, s : 1\}$ se puede representar por `["p", "s"]`, mientras que una representación como `[]` se correspondería a una interpretación en la que todas las variables son asociadas con 0/Falso.

Para trasladar esta interpretación en la evaluación de las fórmulas y conjuntos basta tener en cuenta que la evaluación de fórmulas se puede realizar de forma recursiva tomando los valores de verdad de los átomos según la interpretación y aplicando las funciones de verdad:

- **Evaluación de átomos**. Un átomo será verdadero si y solo sí su correspondiente símbolo proposicional pertenece a la interpretación (lista), y será falso en caso contrario.
- **Evaluación de conectivas**. Aplicando las correspondientes funciones de verdad asociadas a las diversas conectivas lógicas, de forma que:

$$v(\neg F) = \neg v(F) \qquad v(F \wedge G) = v(F) \wedge v(G) \qquad v(F \vee G) = v(F) \vee v(G)$$
$$v(F \rightarrow G) = \neg v(F) \vee v(G) \qquad v(F \leftrightarrow G) = (v(F) \wedge v(G)) \vee (\neg v(F) \wedge \neg v(G))$$

Elm, como el resto de lenguajes habituales, dispone de los operadores booleanos `and` y `or`, por lo que implementar la función es casi una traducción directa de las funciones proporcionadas por el propio lenguaje. En el *código 4.9* se muestra la implementación de la función de evaluación para fórmulas proposicionales. La evaluación de los conjuntos se corresponde con la conjunción de las evaluaciones de las fórmulas del conjunto según la interpretación, de forma que un conjunto de fórmulas se considera verdadero si y sólo si todas sus fórmulas lo son. Elm tiene funciones de verificación para saber si todos los elementos de una lista cumplen una condición, y también si alguno lo cumple: `List.all` y `List.any`, respectivamente. En el *código 4.9* también se presenta la implementación de la función de evaluación de conjuntos.

```
type alias Interpretation = List PSymb

fplValuation : FormulaLP -> Interpretation -> Bool
fplValuation f i =
  case f of
    Atom symb -> List.member symb i
    Neg g -> not (valuation g i)
    Conj g h -> valuation g i &&  valuation h i
    Disj g h -> valuation g i ||  valuation h i
    Impl g h -> not (valuation g i) || valuation h i
    Equi g h ->
      (valuation g &&  valuation h i) ||
        ( not(valuation g i) &&  not (valuation h i))
    Insat -> False
    Taut -> True

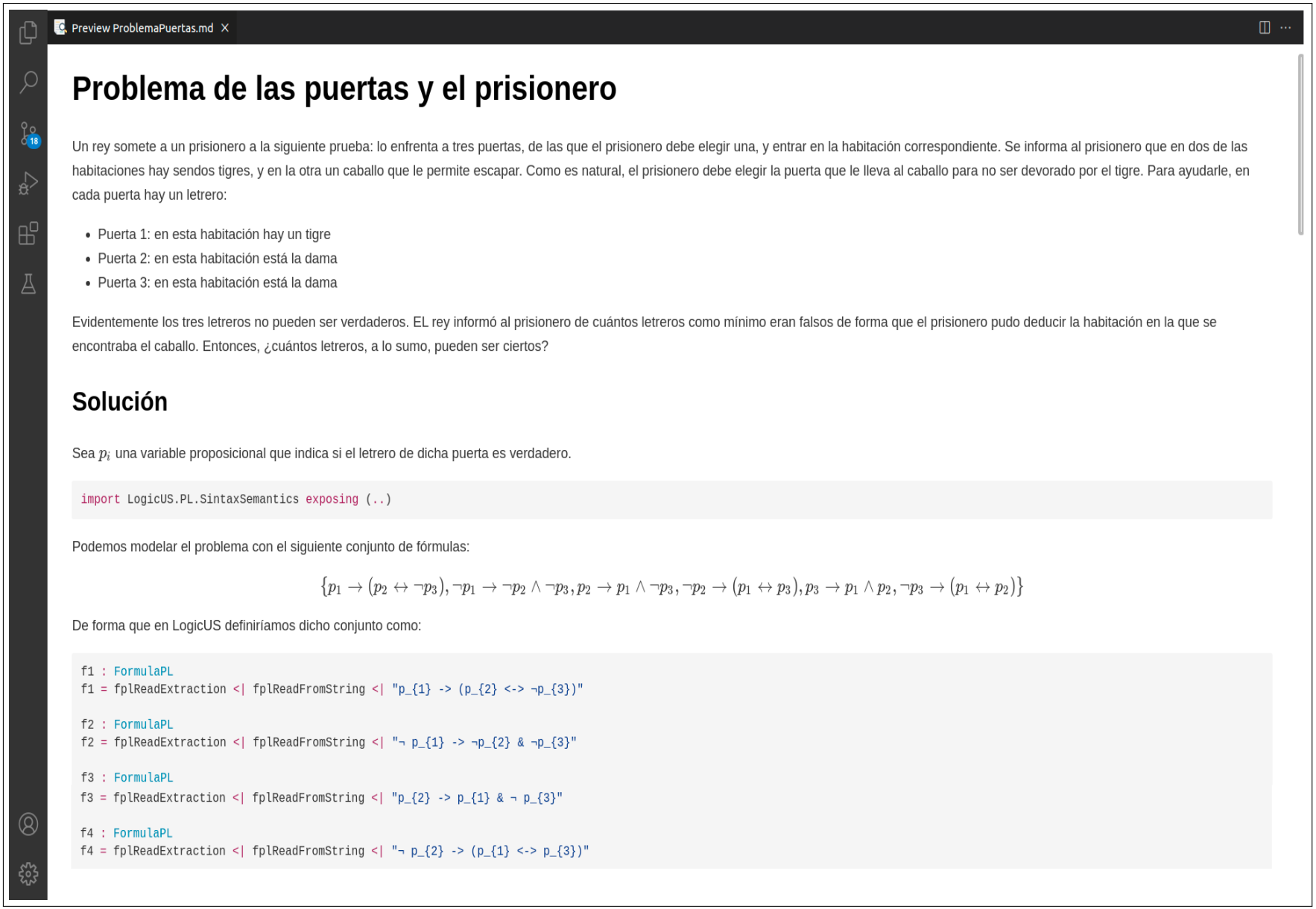

splValuation : SetPL -> Interpretation -> Bool
splValuation fs i =
  List.all (\x -> fplValuation x i) fs
```

Código 4.9: Evaluación de fórmulas en LogicUS

Vista la evaluación, la construcción de la tabla de verdad, la obtención de modelos, contramodelos y la verificación de satisfactibilidad, tautología y consecuencia lógica se pueden reducir a evaluar (por fuerza bruta) cada una de las posibles interpretaciones. Ahora bien, éstas pueden ser calculadas, dada la representación adoptada para las mismas, como los posibles subconjuntos del conjunto de símbolos proposicionales de una fórmula o conjunto de fórmulas, que se corresponden con 2^n casos distintos (con n el número de variables proposicionales consideradas), por lo que puede ser altamente ineficiente desde un punto de vista computacional. Aún así, en el módulo se proporcionan todas las funciones comentadas y los métodos de representación correspondientes, y pueden consultarse todas las funciones expuestas en el paquete *LogicUS.PL.SintaxSemantics* publicado.

Aún así, presentamos un ejemplo que permita ilustrar todos estos aspectos: el problema clásico del prisionero, formulado y resuelto haciendo uso de *LogicUS* en la *figura 4.3*

Vista la implementación de la base de la Lógica Proposicional en LogicUS (Elm) pasaremos en los siguientes apartados a presentar distintas aproximaciones para la resolución del problema *SAT* mostrando las implementaciones de las funciones principales para cada una de ellas.


Preview ProblemaPuertas.md
Problema de las puertas y el prisionero
Un rey somete a un prisionero a la siguiente prueba: lo enfrenta a tres puertas, de las que el prisionero debe elegir una, y entrar en la habitación correspondiente. Se informa al prisionero que en dos de las habitaciones hay sendos tigres, y en la otra un caballo que le permite escapar. Como es natural, el prisionero debe elegir la puerta que le lleva al caballo para no ser devorado por el tigre. Para ayudarle, en cada puerta hay un letrero:
Puerta 1: en esta habitación hay un tigre
Puerta 2: en esta habitación está la dama
Puerta 3: en esta habitación está la dama
Evidentemente los tres letreros no pueden ser verdaderos. EL rey informó al prisionero de cuántos letreros como mínimo eran falsos de forma que el prisionero pudo deducir la habitación en la que se encontraba el caballo. Entonces, ¿cuántos letreros, a lo sumo, pueden ser ciertos?
Solución
Sea p_i una variable proposicional que indica si el letrero de dicha puerta es verdadero.
import LogicUS.PL.SintaxSemantics exposing (..)
Podemos modelar el problema con el siguiente conjunto de fórmulas:
$$\{p_1 \to (p_2 \leftrightarrow \neg p_3), \neg p_1 \to \neg p_2 \land \neg p_3, p_2 \to p_1 \land \neg p_3, \neg p_2 \to (p_1 \leftrightarrow p_3), p_3 \to p_1 \land p_2, \neg p_3 \to (p_1 \leftrightarrow p_2)\}$$
De forma que en LogicUS definiríamos dicho conjunto como:
f1 : FormulaPL
f1 = fplReadExtraction <| fplReadFromString <| "p_{1} -> (p_{2} <-> ¬p_{3})"
f2 : FormulaPL
f2 = fplReadExtraction <| fplReadFromString <| "¬ p_{1} -> ¬p_{2} & ¬p_{3}"
f3 : FormulaPL
f3 = fplReadExtraction <| fplReadFromString <| "p_{2} -> p_{1} & ¬ p_{3}"
f4 : FormulaPL
f4 = fplReadExtraction <| fplReadFromString <| "¬ p_{2} -> (p_{1} <-> p_{3})"

Preview ProblemaPuertas.md

```
f5 : FormulaPL
f5 = fplReadExtraction <| fplReadFromString <| "p_{3} -> p_{1} & ¬p_{2}"

f6 : FormulaPL
f6 = fplReadExtraction <| fplReadFromString <| "¬ p_{3} -> (p_{1} <-> p_{2})"

u : SetPL
u = [f1,f2,f3,f4,f5, f6]
```

Que corresponde justamente con el que hemos propuesto.

```
repr_u : String
repr_u = "$$" ++ (splToMathString u) ++ "$$"
```

$$
\begin{array}{l}
(p_1 \rightarrow (p_2 \leftrightarrow \neg p_3)) \\
(\neg p_1 \rightarrow (\neg p_2 \wedge \neg p_3)) \\
(p_2 \rightarrow (p_1 \wedge \neg p_3)) \\
(\neg p_2 \rightarrow (p_1 \leftrightarrow p_3)) \\
(p_3 \rightarrow (p_1 \wedge \neg p_2)) \\
(\neg p_3 \rightarrow (p_1 \leftrightarrow p_2))
\end{array}
$$

Ahora, dado que se dice cuántas como mínimo son falsas, tal que el problema sea inambiguamente resoluble, entonces veamos qué ocurre si al menos 1 es falsa. Entonces tenemos las interpretaciones tal que o una sola es falsa, o dos son falsas, o las tres son falsas:

```
-- p1 es falsa
i1 : Interpretation
i1 = [("p",[2]), ("p",[3])]

--p2 es falsa
i2 : Interpretation
i2 = [("p",[1]), ("p",[3])]

--p3 es falsa
i3 : Interpretation
i3 = [("p",[1]), ("p",[2])]

-- p1 y p2 son falsas
i4 : Interpretation
i4 = [("p",[3])]

-- p1 y p3 son falsas
i5 : Interpretation
i5 = [("p",[2])]
```

Preview ProblemaPuertas.md

```
-- p2 y p3 son falsas
i6 : Interpretation
i6 = [("p",[1])]

-- p1, p2 y p3 son falsas
i7 : Interpretation
i7 = []
```

Ahora podríamos evaluar el conjunto respecto a dichas interpretaciones, evaluando cada fórmula y tomando la conjunción de las evaluaciones. Por ejemplo para `i1`:

```
vf11 : Bool
vf11 = fplValuation f1 i1
```

True

```
vf21 : Bool
vf21 = fplValuation f2 i1
```

False

No hace falta seguir evaluando las demás fórmulas, la evaluación del conjunto sería falsa, dada la propiedad $A \wedge F = F$. En efecto:

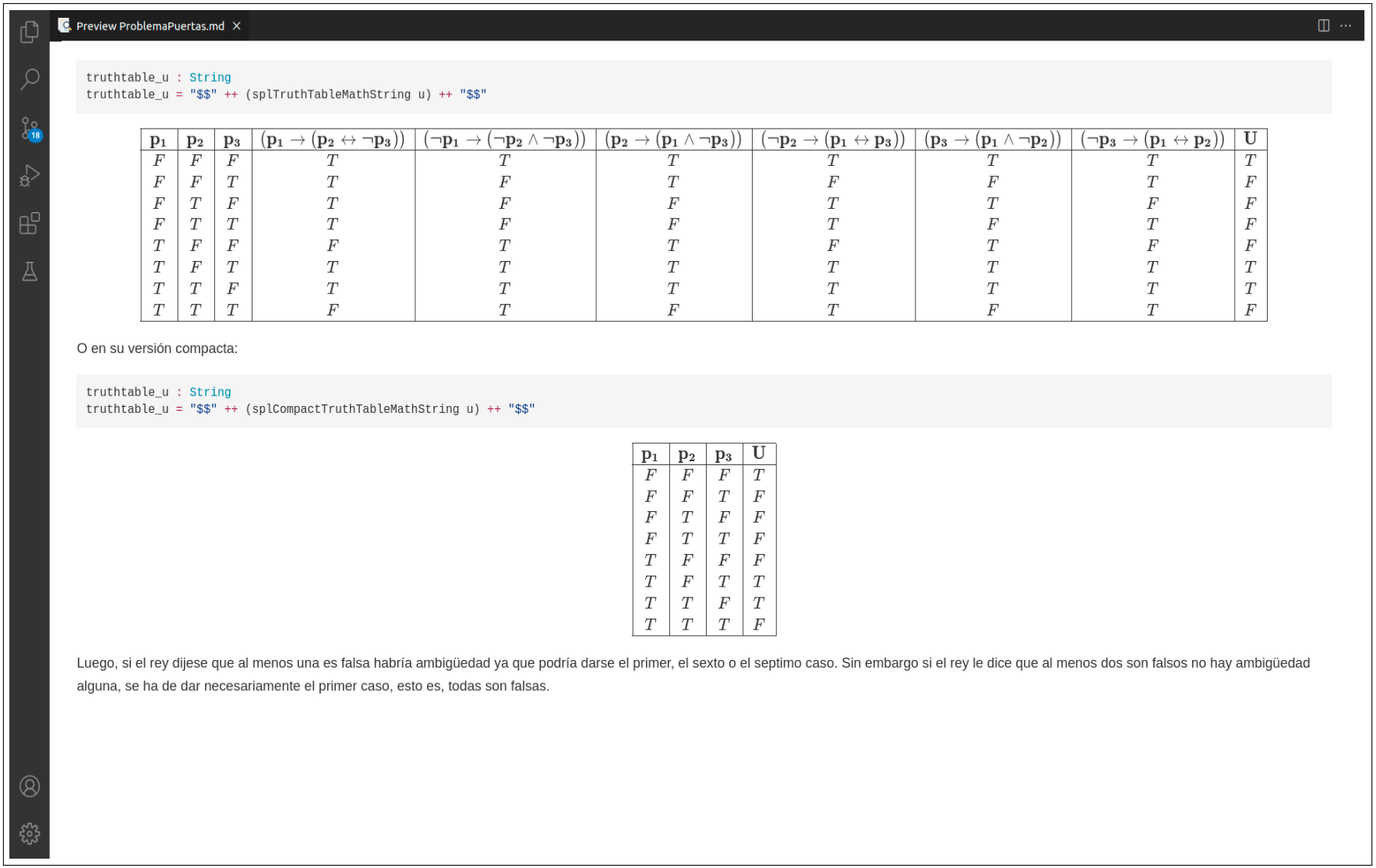
```
vu1 : Bool
vu1 = splValuation u i1
```

False

Podríamos seguir con el resto, pero esto es prácticamente el cálculo de la tabla de verdad (excepto para el caso todos verdaderos que es claramente falso). Representemos la tabla de verdad para el conjunto:

```
truthtable_u : String
truthtable_u = "$$" ++ (splTruthTableMathString u) ++ "$$"
```

$\mathbf{p_1}$	$\mathbf{p_2}$	$\mathbf{p_3}$	$(\mathbf{p_1} \to (\mathbf{p_2} \leftrightarrow \neg \mathbf{p_3}))$	$(\neg \mathbf{p_1} \to (\neg \mathbf{p_2} \wedge \neg \mathbf{p_3}))$	$(\mathbf{p_2} \to (\mathbf{p_1} \wedge \neg \mathbf{p_3}))$	$(\neg \mathbf{p_2} \to (\mathbf{p_1} \leftrightarrow \mathbf{p_3}))$	$(\mathbf{p_3} \to (\mathbf{p_1} \wedge \neg \mathbf{p_2}))$	$(\neg \mathbf{p_3} \to (\mathbf{p_1} \leftrightarrow \mathbf{p_2}))$	$\mathbf{U}$
F	F	F	T	T	T	T	T	T	T
F	F	T	T	F	T	F	F	T	F
F	T	F	T	F	F	T	T	F	F
F	T	T	T	F	F	T	F	T	F
T	F	F	F	T	T	F	T	F	F
T	F	T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T	T	T
T	T	T	F	T	F	T	F	T	F



p_1	p_2	p_3	$(p_1 \to (p_2 \leftrightarrow \neg p_3))$	$(\neg p_1 \to (\neg p_2 \wedge \neg p_3))$	$(p_2 \to (p_1 \wedge \neg p_3))$	$(\neg p_2 \to (p_1 \leftrightarrow p_3))$	$(p_3 \to (p_1 \wedge \neg p_2))$	$(\neg p_3 \to (p_1 \leftrightarrow p_2))$	U
F	F	F	T	T	T	T	T	T	T
F	F	T	T	F	T	F	F	T	F
F	T	F	T	F	F	T	T	F	F
F	T	T	T	F	F	T	F	T	F
T	F	F	F	T	T	F	T	F	F
T	F	T	T	T	T	T	T	T	T
T	T	F	T	T	T	T	T	T	T
T	T	T	F	T	F	T	F	T	F

O en su versión compacta:

p_1	p_2	p_3	U
F	F	F	T
F	F	T	F
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	T
T	T	F	T
T	T	T	F

Luego, si el rey dijese que al menos una es falsa habría ambigüedad ya que podría darse el primer, el sexto o el septimo caso. Sin embargo si el rey le dice que al menos dos son falsos no hay ambigüedad alguna, se ha de dar necesariamente el primer caso, esto es, todas son falsas.

Figura 4.3: Ejemplo de uso del módulo LogicUS.PL.SintaxSemantics
Fuente propia. Creada con litvis

4.2.4. LogicUS.PL.SemanticTableaux

En este módulo se exponen las funciones y tipos necesarios para la realización de los tableros semánticos. En la construcción del tablero resultan de interés los tipos de fórmulas y las reglas de derivación de las mismas.

Comencemos por los tipos de fórmulas. Para la resolucíón por Tableros Semánticos consideramos cuatro clases de fórmulas, las de tipo α (fórmulas con carácter conjuntivo), tipo β (fórmulas de carácter disyuntivo), tipo dN (doble negación) y literales L. Estas categorías son precisamente las que se han considerado en la implementación (añadiendo I para la fórmula insatisfactible $\perp$).

Se define el tipo Formula, `FormulaPLType`, con estas cuatro clases (constructores), y la función `fplType` que devuelve el tipo correspondiente. Para su definición se hace uso del emparejamiento de patrones (*Pattern Matching*) que Elm incluye entre sus características (ver *código 4.10*).

Una vez definidos los tipos, hemos de definir las acciones de las reglas. Con el fin de unificar todas bajo un mismo marco de operación que permita después hacer uso de una sola función a la hora de construir los tableros, definimos `fplComponents`, que obtiene las componentes de la fórmula que resultarían de aplicar la regla correspondiente. A partir de estas componentes se puede construir el tablero sin más que aplicar las reglas según el caso de la fórmula, tal y como veremos a continuación, dado que vamos a exponer de forma detallada la implementación de la construcción del tablero.

```
type FormulaPLType  =
    L | DN | A | B | I | T

fplType : FormulaPL -> FormulaPLType
fplType f =
    case f of
        Atom _ -> L

        Neg (Atom _) -> L

        Neg (Neg _) -> DN

        Neg (Conj _ _) -> B

        Neg (Disj _ _) -> A

        Neg (Impl _ _) -> A

        Neg (Equi _ _) -> B

        Neg Insat -> T

        Neg Taut -> I

        Conj _ _ -> A

        Disj _ _ -> B

        Impl _ _ -> B

        Equi _ _ -> A

        Insat -> I

        Taut -> T
```

Código 4.10: Clases α, β, dN, L, I en LogicUS

En el *algoritmo 2.2* se expone una versión iterativa del algoritmo de Tableros Semánticos. Sin embargo, en los lenguajes funcionales los bucles iterativos no existen explícitamente sino que es necesario hacer uso, o bien de funciones de orden superior (`map`, `fold`, etc.), o bien de algún mecanismo de recursión.

Para ello, el procedimiento, formalmente descrito en el *algoritmo 4.1*, recibirá un conjunto de fórmulas y un número que corresponde al identificador del nodo que le corresponde, inicialmente 0 (podríamos almacenar los nodos en cada momento pero ello requeriría tener que estar comprobando el número de nodos en cada paso). Además, vamos a devolver el grafo (por el momento) como una estructura con dos listas:

- una para los nodos, en la que cada elemento tendrá el *id* (único), un número según el *tipo* de nodo (0:nodo interior, 1:hoja satisfactible, -1:hoja insatisfactible) y el conjunto de fórmulas correspondiente a dicho nodo.
- otra para las aristas, que están formadas por el *id* del nodo origen (padre), el *id* de nodo destino (hijo), el *tipo* de regla que se aplica, y los índices de las fórmulas en el conjunto (en realidad lista) sobre los que se aplica.

Algoritmo 4.1: Tablero Semántico para conjuntos de fórmulas proposicionales

TableroSemánticoAux(U)**:**

Input : U: Un conjunto de fórmulas proposicionales; *nid*: El número de nodo que corresponde al nodo actual

Output: (T_n, T_e): Una tupla con un conjunto con los nodos y otro con las aristas del tablero.

1. Si U_n contiene un par de fórmulas complementarias devolver $(T_n, T_e) = (\{(nid, -1, U)\}, \emptyset)$
2. Si no, si en U_n existe $F_i \equiv \neg\neg G$ (tipo dN) entonces:
   - 2.1 Hacer $(T'_n, T'_e) = \mathit{TableroSemánticoAux}(U \backslash \{F\} \cup \{G\}, nid + 1)$
   - 2.2 Devolver $(T_n, T_e) = (T'_n \cup \{(nid, 0, U)\}, T'_e \cup \{(nid, nid + 1, dN, [i])\})$
3. Si no, si en U_n existe $F_i \equiv G \wedge H$ (tipo α) entonces:
   - 3.1 Hacer $(T'_n, T'_e) = \mathit{TableroSemánticoAux}(U \backslash \{F\} \cup \{G\}, nid + 1)$
   - 3.2 Devolver $(T_n, T_e) = (T'_n \cup \{(nid, 0, U)\}, T'_e \cup \{(nid, nid + 1, \alpha, [i])\})$
4. Si no, si en U_n existe $F_i \equiv G \vee H$ (tipo β) entonces:
   - 4.1 Hacer $(, (T'_e)^{(1)}) = \mathit{TableroSemánticoAux}(U \backslash \{F\} \cup \{G\}, nid + 1)$
   - 4.2 Hacer $nextId = nid + \left|(T'_n)^{(1)}\right| + 1$
   - 4.3 Hacer $((T'_n)^{(2)}, (T'_e)^{(2)}) = \mathit{TableroSemánticoAux}(U \backslash \{F\} \cup \{H\}, nextId)$
   - 4.4 Devolver $(T_n, T_e) = ((T'_n)^{(1)} \cup (T'_n)^{(2)} \cup \{(nid, 0, U)\}, T'_e \cup \{(nid, nid + 1, dN, [i])\}$
5. Si no, entonces U es un conjunto de literales sin complementarios luego devolver $(T_n, T_e) = (\{(nid, 1, U)\}, \emptyset)$

end

La implementación, que se muestra en el *código 4.11*, se sigue directamente de este algoritmo teniendo en cuenta los siguientes detalles:

- `splSearchDN`, `splSearchAlpha`, `splSearchBeta`, `splSearchContradiction` devuelven la fórmula y la lista de índices de la fórmula según el tipo correspondiente. Si no existe, devuelven `Nothing`.
- `splExpandDN`, `splExpandAlpha` y `splExpandBeta` realizan la acción de eliminar la variable correspondiente y generar el descendiente o descendientes (en caso de β), eliminando además las tautologías, que son irrelevantes (dada la propiedad $A \wedge \top \equiv \top$).
- `splSemanticTableauBuilder` se ha definido como función local (*nested*) de la función principal `semanticTableau` que únicamente elimina del conjunto original las fórmulas repetidas (si las hay), ejecuta el algoritmo partiendo del $nid = 0$ y, finalmente, crea un elemento de tipo `Graph` a partir de los nodos y aristas obtenidos. Este objeto permite generar una representación tanto en formato cadena como en formato DOT (visualizable con un renderizador Viz), que aprovecharemos para definir los mecanismos de representación de los tableros.

Una cuestión interesante de este algoritmo es que el método de Tableros Semánticos no sólo resuelve la satisfactibilidad de un conjunto sino que también permite la extracción de modelos a partir de las hojas abiertas, donde cada una de ellas da lugar a un conjunto de modelos en los que los literales negativos definen las variables consideradas falsas, mientras que los literales positivos definen aquellas consideradas verdaderas (las que no aparecen dan lugar a distintos modelos, asumiendo cualquier de los valores posibles).

Aunque la función que realiza el procesamiento de transformación de los literales está definida en el módulo *LogicUS.Pl.SintaxSemantics*, hemos preferido presentarla en este momento en vez de presentarla en el primero de los apartados sin un contexto adecuado.

```
semanticTableau : SetPL -> PLSemanticTableau
semanticTableau fs =
let
splSemanticTableauBuilder xs nid =
case splSearchContradiction xs of
Just _ ->
( [ Graph.Node nid ( -1, xs ) ], [] )

Nothing ->
let
currentNode = Graph.Node nid ( 0, xs )
in
case splSearchDN xs of
Just ( i, f ) ->
let
( nodes, edges ) =
splSemanticTableauBuilder (splExpandDN xs f) (nid + 1)
in
( currentNode :: nodes, Graph.Edge nid (nid + 1) ( DN, [ i ] ) :: edges )

Nothing ->
case splSearchAlpha xs of
Just ( i, f ) ->
let
( nodes, edges ) =
splSemanticTableauBuilder (splExpandAlpha xs f) (nid + 1)
in
( currentNode :: nodes,
Graph.Edge nid (nid + 1) ( A, [ i ] ) :: edges )

Nothing ->
case splSearchBeta xs of
Just ( i, f ) ->
let
expansion = splExpandBeta xs f
in
let
alt1 = Tuple.first expansion
alt2 = Tuple.second expansion
in
let
( nodes1, edges1 ) =
splSemanticTableauBuilder alt1 (nid + 1)
in
let
nextid = nid + List.length nodes1 + 1
in
let
( nodes2, edges2 ) =
splSemanticTableauBuilder alt2 nextid
in
( currentNode :: (nodes1 ++ nodes2),
[ Graph.Edge nid (nid + 1) ( B, [ i ] ),
Graph.Edge nid nextid ( B, [ i ] ) ] ++ edges1 ++ edges2 )

Nothing ->
( [ Graph.Node nid ( 1, xs ) ], [] )
in
let
( ns, es ) =
splSemanticTableauBuilder (uniqueConcatList [] fs) 0
in
Graph.fromNodesAndEdges ns es
```

Código 4.11: Construcción del Tablero Semático en LogicUS

La función `interpretationsFromSymbolsAndLiterals` toma una lista de literales y un conjunto de símbolos y calcula las distintas interpretaciones siguiendo el procedimiento anterior, para ello, y dada la definición compacta de interpretación, calcula todos los subconjuntos (`powerset`) del subconjunto de símbolos que no aparecen en la lista de literales (variables cuyo valor es irrelevante) y añade los símbolos que aparecen positivos en la lista de literales.

Usando la función anterior para el cálculo de los modelos a partir del tablero, es suficiente recorrer las hojas abiertas asociando a cada una de ellas las interpretaciones correspondientes, y considerando todas en una única lista eliminando las interpretaciones repetidas (ver *código 4.12*).

```
interpretationsFromSymbolsAndLiterals : List PSymb -> List Literal -> List Interpretation
interpretationsFromSymbolsAndLiterals symbs literals =
  let
    symbsLiterals =
      splSymbols literals

    trueSymbs =
      List.filter (\x -> List.member (Atom x) literals) symbs
  in
  let
    optSymbs =
      powerset <| List.filter (\x -> not <| List.member x symbsLiterals) symbs
  in
    List.map (\ls -> (List.sort << LE.unique) (ls ++ trueSymbs)) optSymbs

semanticTableauModels : PLSemanticTableau -> List Interpretation
semanticTableauModels st =
let
  symbs =
  (Maybe.withDefault (Node 0 ( 0, [] )) <| List.head <| Graph.nodes st).label |>
    Tuple.second |> PL_SS.splSymbols

  openLeaves =
    List.foldr
      (\x ac ->
        if Tuple.first x.label == 1 then
          Tuple.second x.label :: ac

        else
          ac
      )
      []
      (Graph.nodes st)
in
List.sort <| LE.unique <| List.concat <|
  List.map (\ls -> PL_SS.interpretationsFromSymbolsAndLiterals symbs ls) openLeaves
```

Código 4.12: Extracción de modelos del tablero semático en LogicUS

En el código previo aparecen dos funciones que ya hemos comentado y descrito en detalle cuando expusimos el lenguaje Elm y que, como dijimos, eran de gran interés en los lenguajes declarativos, como son la función `map` y la función `foldl`. Recuérdese, que la función `map` aplica una función a todos los elementos de la lista (o conjunto, o diccionario, ...), mientras que el plegado (*foldl*) emula lo que serían los bucles iterativos con acumulador partiendo de un acumulador inicial (de cualquier tipo según el valor que se quiera calcular), una lista (o conjunto, o diccionario, ...) y una función sobre los elementos. Estas funciones, u otras similares, serán ampliamente utilizadas durante la descripción del trabajo de implementación que estamos presentando.

Para cerrar este apartado, presentaremos la función de representación de tableros, que será prácticamente equivalente para otras representaciones gráficas del desarrollo de los algoritmos que se harán en futuros apartados. Como las implementaciones para las funciones de representación de los tableros en formato cadena y en formato DOT son prácticamente equivalentes, mostramos únicamente, expuesta en el *código 4.13*, la implementación en detalle de la función `sematicTableauToDOTString`.

El tipo `Graph n e` corresponde a un grafo que puede contener cualquier tipo de objeto tanto en los nodos como en las aristas. Sin embargo, la función `Graph.toString` provee un mecanismo para generar dicha representación, tomando, además del grafo, dos funciones:

- una de tipo (`n -> Maybe String`), que da la representación de las etiquetas de los nodos,

- y otra (`e -> Maybe String`), que da la representación de las etiquetas de las aristas.

Nótese que ambas funciones son parciales, de forma que aquellas etiquetas (de nodos o aristas) para las cuales las respectivas funciones no estén definidas, no constarán de una etiqueta en la representación.

Análogamente, la función `Graph.DOT.outputWithStyles` recibe los argumentos comentados y uno más (propio de la notación GraphViz: color y forma de los nodos y aristas, etc) de forma que se genera una cadena en formato DOT que es visualizable en un renderizador Viz.

Para poder llevar a cabo las implementaciones realizadas a partir de estas funciones, hemos de considerar un detalle (que será equivalente en el resto de los métodos basados en grafos que se presentan en los siguiente módulos): en la representación de los tableros normalmente a las ramas cerradas se les suele añadir un nodo etiquetado con $\times$ y a las abiertas etiquetado con $\bigcirc$. Para ello, antes de generar el tablero final parece obvio que tendremos que añadir a cada hoja el descendiente correspondiente, según la consistencia (marcada con 1) o inconsistencia (marcada con -1) de la misma.

Nótese que esto se podría haber llevado a cabo directamente en la construcción del tablero, pero tratándose de una mera cuestión de representación hemos preferido trasladarlo a esta función.

Por ello, para cada hoja (marcada con $i = 1$ o $i = -1$) generamos un nuevo nodo etiquetado con $(2 \cdot i, [])$ (tomar $2 * i$ es solo para diferenciarlas de las originales y poder generar su representación como $\bigcirc$ o $\times$) y lo asociamos con la hoja de referencia, etiquetando la arista como:

- $(I, [i_1, i_2])$ si la hoja es cerrada, donde i_1, i_2 corresponden a los índices de las fórmulas complementarias, o un único índice si la insatisfactibilidad se debiese a la aparición de `Insat`,
- o $(L, [])$ en caso de que se trate de una hoja abierta.

Finalmente, para las funciones de representación:

- Asignaremos a cada nodo la representación del conjunto de fórmulas asociado al mismo. Y a los nodos añadidos (con la lista vacía como conjunto asociado) asiganremos $\bigcirc$ o $\times$ según su satisfactibilidad (1 o -1).

- Asignaremos a cada arista el símbolo correspondiente según el tipo de regla dN, α, β, I, L y acompañado de la representación como cadena de la lista de índices, si procede.

Con el fin de ilustrar el procedimiento de tableros semánticos en el paquete LogicUS presentamos un ejemplo de aplicación del mismo a la reducción de satisfactibilidad y extracción de modelos (*figura 4.4*).

Vista la implementación de tableros semánticos en LogicUS, pasamos a abordar la implementación de la transformación de fórmulas en formas normales, que será la base de los dos siguientes módulos que se presentarán tras el mismo.

```
semanticTableauToDOT : PLSemanticTableau -> String
semanticTableauToDOT t =
    let
        toStringNode =
            \( i, fs2 ) ->
                Just <|
                    if i == -2 then "×"
                    else if i == 2 then "○"
                    else PL_SS.splToString fs2

        toStringEdge =
            \( ftype, is ) ->
                Just <|
                    case ftype of
                        L -> "L"

                        DN -> "dN (" ++ String.join ", "
                                    (List.map (\i -> String.fromInt (i + 1)) is) ++ ")"

                        A ->
                            "α (" ++ String.join ", "
                                (List.map (\i -> String.fromInt (i + 1)) is) ++ ")"

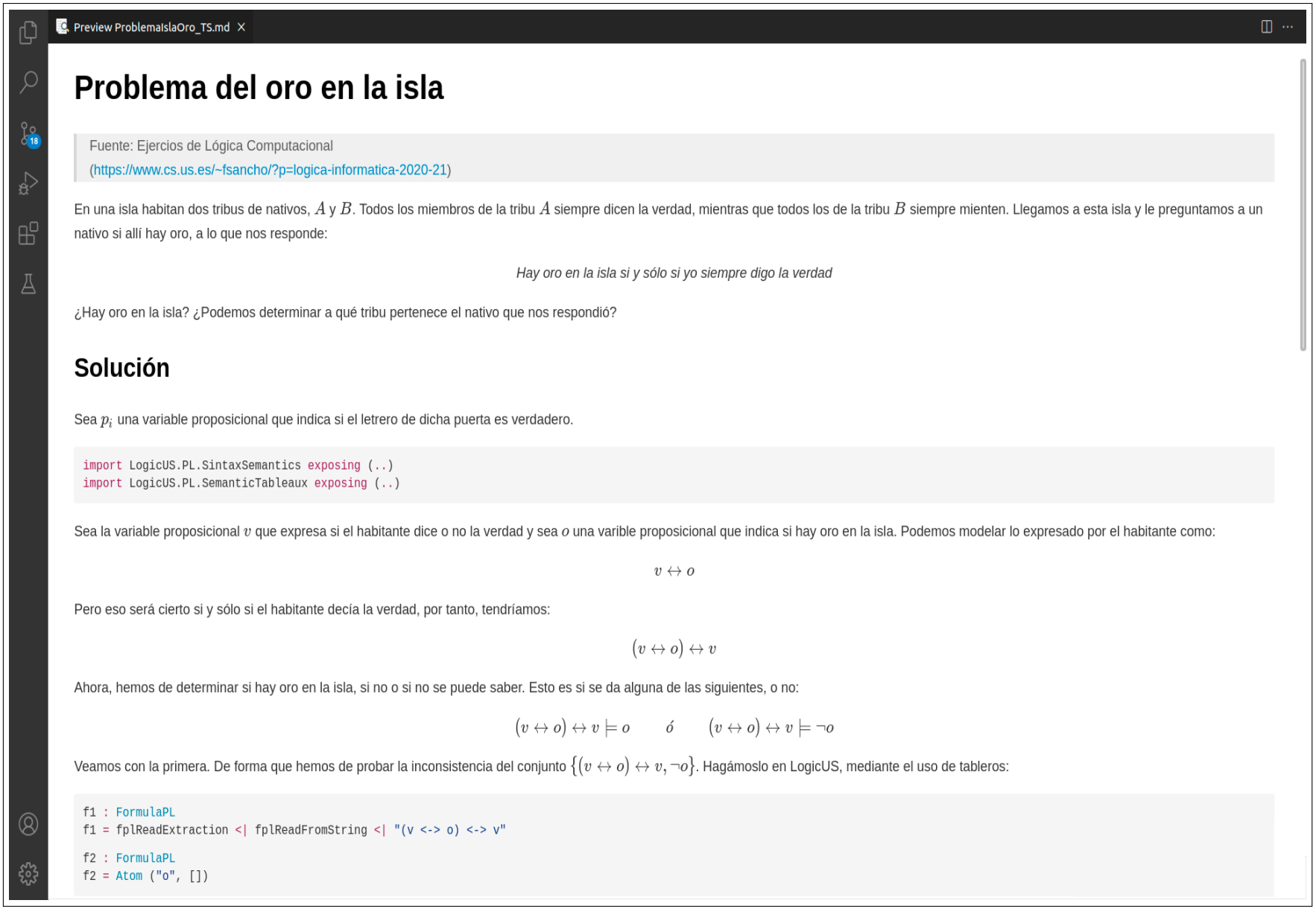
                        B ->
                            "β (" ++ String.join ", "
                                (List.map (\i -> String.fromInt (i + 1)) is) ++ ")"

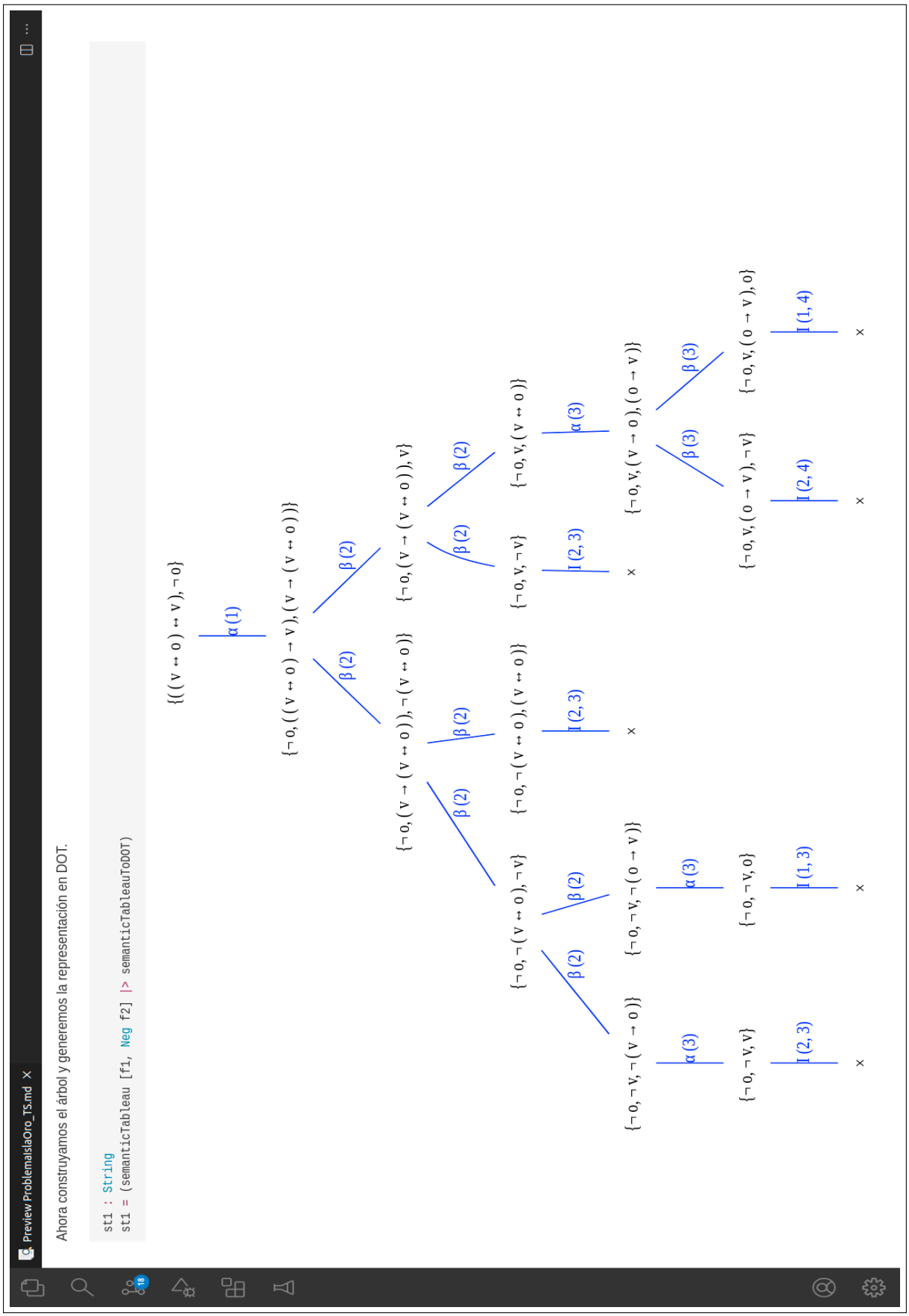
                        I ->
                            "I (" ++ String.join ", "
                                (List.map (\i -> String.fromInt (i + 1)) is) ++ ")"

                        T ->
                            "T"

        myStyles =
            { defaultStyles | node = "shape=box, color=black",
                edge = "dir=none, color=blue, fontcolor=blue" }
    in
    let
        newLeaves =
            List.indexedMap
                (\j n ->
                    let
                        nid = n.id
                        ( i, fs2 ) = n.label
                    in
                    ( Graph.Node (Graph.size t + j) ( 2 * i, [] )
                    , Graph.Edge nid
                        (Graph.size t + j) <|
                            if i == 1 then( L, [] )
                            else (I, Maybe.withDefault [] (splSearchContradiction fs2))
                    )
                )
                (List.filter (\n -> Tuple.first n.label /= 0) <| Graph.nodes t)
    in
    Graph.DOT.outputWithStyles myStyles toStringNode toStringEdge <|
        Graph.fromNodesAndEdges
            (Graph.nodes t ++ List.map Tuple.first newLeaves)
            (Graph.edges t ++ List.map Tuple.second newLeaves)
```

Código 4.13: Función de representación DOT del tablero semático en LogicUS


Preview ProblemaIslaOro_TS.md
Problema del oro en la isla
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
En una isla habitan dos tribus de nativos, A y B. Todos los miembros de la tribu A siempre dicen la verdad, mientras que todos los de la tribu B siempre mienten. Llegamos a esta isla y le preguntamos a un nativo si allí hay oro, a lo que nos responde:
Hay oro en la isla si y sólo si yo siempre digo la verdad
¿Hay oro en la isla? ¿Podemos determinar a qué tribu pertenece el nativo que nos respondió?
Solución
Sea p_i una variable proposicional que indica si el letrero de dicha puerta es verdadero.
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.PL.SemanticTableaux exposing (..)
Sea la variable proposicional v que expresa si el habitante dice o no la verdad y sea o una varible proposicional que indica si hay oro en la isla. Podemos modelar lo expresado por el habitante como:
$$v \leftrightarrow o$$
Pero eso será cierto si y sólo si el habitante decía la verdad, por tanto, tendríamos:
$$(v \leftrightarrow o) \leftrightarrow v$$
Ahora, hemos de determinar si hay oro en la isla, si no o si no se puede saber. Esto es si se da alguna de las siguientes, o no:
$$(v \leftrightarrow o) \leftrightarrow v \models o \qquad ó \qquad (v \leftrightarrow o) \leftrightarrow v \models \neg o$$
Veamos con la primera. De forma que hemos de probar la inconsistencia del conjunto $\{(v \leftrightarrow o) \leftrightarrow v, \neg o\}$. Hagámoslo en LogicUS, mediante el uso de tableros:
f1 : FormulaPL
f1 = fplReadExtraction <| fplReadFromString <| "(v <-> o) <-> v"
f2 : FormulaPL
f2 = Atom ("o", [])


Preview ProblemaIslaOro_TS.md
Ahora construyamos el árbol y generemos la representación en DOT.
st1 : String
st1 = (semanticTableau [f1, Neg f2] |> semanticTableauToDOT)
{((V ↔ O) ↔ V), ¬O}
α (1)
{¬O, ((V ↔ O) → V), (V → (V ↔ O))}
β (2)
β (2)
{¬O, (V → (V ↔ O)), ¬(V ↔ O)}
{¬O, (V → (V ↔ O)), V}
β (2)
β (2)
β (2)
β (2)
{¬O, ¬(V ↔ O), ¬V}
{¬O, ¬(V ↔ O), (V ↔ O)}
{¬O, V, ¬V}
{¬O, V, (V ↔ O)}
I (2, 3)
I (2, 3)
α (3)
β (2)
β (2)
α (3)
{¬O, ¬V, ¬(V → O)}
{¬O, ¬V, (O → V)}
×
×
{¬O, V, (V → O), (O → V)}
α (3)
β (3)
β (3)
{¬O, ¬V, V}
{¬O, ¬V, O}
{¬O, V, (O → V), ¬V}
{¬O, V, (O → V), O}
I (2, 3)
I (1, 3)
I (2, 4)
I (1, 4)
×
×
×
×

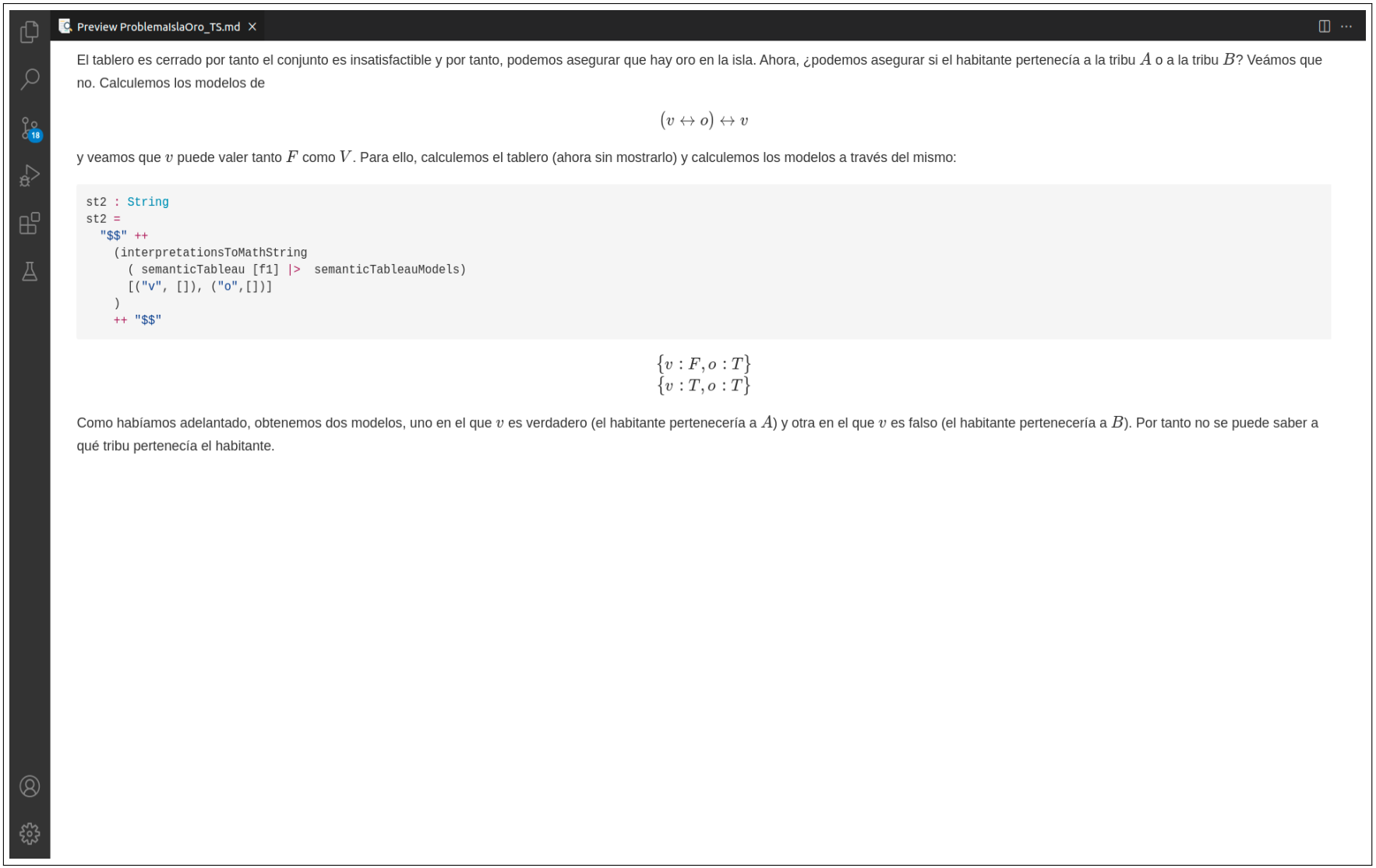


Figura 4.4: Ejemplo de uso del módulo LogicUS.PL.SemanticTableaux
Fuente propia. Creada con litvis

4.2.5. LogicUS.NormalForms

Formas Normales

En esta sección presentaremos la implementación de la transformación de fórmulas en fórmulas equivalentes en forma normal. Recuérdese que en el *algoritmo 2.3* expusimos el método de transformación a formas normales bajo la aplicación de 4 reglas básicas:

1. Eliminación de las equivalencias: $(A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$.
2. Eliminación de las implicaciones $A \rightarrow B \equiv \neg A \vee B$.
3. Interiorización de las negaciones (De Morgan): $\neg(A[\wedge/\vee]B) \equiv \neg A[\vee/\wedge]\neg B)$ y $\neg\neg A = A$. (FNN)
4. Aplicación de la ley distributiva (la adecuada según la forma normal buscada):
   - FNC: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
   - FND: $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

El módulo implementa exactamente estos pasos en distintas funciones, cuya composición da lugar al cálculo de las formas normales correspondientes. Las dos primeras reglas son fácilmente implementables sin más que aplicar recursivamente la regla correspondiente, por lo que no las mostraremos. Sí mostraremos la implementación de la función `fplToNNF` que implementa en una misma función las tres primeras reglas, calculando la Forma Normal Negativa. También presentaremos la implementación de la función `fplInteriorizeAllDisj` correspondiente a la interiorización de las disyunciones y la `fplToCNF` (como composición de las dos anteriores). La implementación de las funciones `fplInteriorizeAllConj` y `fplToDNF` es equivalente y no será expuesta explícitamente.

La función `fplToNNF` implementa la regla de interiorización de negaciones y está definida mediante dos funciones mutuamente recursivas. La función principal se encarga de ir interiorizando las negaciones en la fórmula principal y en aquellas subfórmulas que no corresponden a una negación, mientras que la función interna se encarga precisamente de aplicar las leyes de De Morgan. Nótese que, además, las dos primeras reglas se aplican directamente a través de las reglas de equivalencia siguientes:

$$A \rightarrow B \equiv \neg A \vee B \qquad A \leftrightarrow B \equiv (\neg A \vee B) \wedge (\neg B \vee A)$$
$$\neg(A \rightarrow B) \equiv A \wedge \neg B \qquad \neg(A \leftrightarrow B) \equiv (A \wedge B) \vee (B \wedge \neg A)$$

```
fplToNNFAux : FormulaPL -> FormulaPL
fplToNNFAux p =
  case p of
      Atom x ->
          Neg (Atom x)

      Neg x ->
          fplToNNF x

      Conj x y ->
          Disj (fplToNNFAux x) (fplToNNFAux y)

      Disj x y ->
          Conj (fplToNNFAux x) (fplToNNFAux y)

      Impl x y ->
          Conj (fplToNNF x) (fplToNNFAux y)

      Equi x y ->
          Disj (Conj (fplToNNF x) (fplToNNFAux y)) (Conj (fplToNNFAux x) (fplToNNF y))

      Insat ->
          Taut

      Taut ->
          Insat
```

```
fplToNNF : FormulaPL -> FormulaPL
fplToNNF f =
    case f of
        Atom x ->
            Atom x

        Neg x ->
            fplToNNFAux x

        Conj x y ->
            Conj (fplToNNF x) (fplToNNF y)

        Disj x y ->
            Disj (fplToNNF x) (fplToNNF y)

        Impl x y ->
            Disj (fplToNNFAux x) (fplToNNF y)

        Equi x y ->
            Conj (Disj (fplToNNFAux x) (fplToNNF y)) (Disj (fplToNNFAux y) (fplToNNF x))

        Insat ->
            Insat

        Taut ->
            Taut
```

Código 4.14: Función de paso a FNN en LogicUS

Presentada la función `fplToNNF`, pasamos a presentar la función de interiorización de disyunciones, `fplInteriorizeAllDisj`. Debe indicarse que esta función es únicamente aplicable sobre funciones en FNN, por lo que es parcial y devolverá un elemento de tipo `Maybe FormulaPL`. Se ha implementado como una función recursiva que se basa en la aplicación de la ley distributiva $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, donde se han considerando tres casos relevantes:

- Si se tiene la fórmula $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$, entonces se toma la función $(A \vee B) \wedge (A \vee C)$ y se aplica de nuevo `fplInteriorizeAllDisj` sobre la misma. De igual modo para $(B \wedge C) \vee A$, dado el carácter conmutativo de la disyunción.
- Si se tiene la fórmula $A \wedge (B \vee C)$, se aplica de nuevo `fplInteriorizeAllDisj` sobre las subfórmulas de la conjunción misma.
- Si se tiene el caso $A \vee B$ (y no se ha tomado el primer caso), entonces primero se interiorizan las disyunciones A y B, y solo en el caso de que alguna de las dos (o ambas) sean conjunciones habría de aplicarse de nuevo la interiorización de la disyunción.

Una vez implementada dicha función, mostrada en el *código 4.15*, basta componer las dos funciones comentadas previamente `fplToNNF` y `fplInteriorizeAllDisj` para obtener la FNC implementada en `fplToCNF`. Nótese que `fplInteriorizeAllDisj` devuelve un elemento `Maybe`, pero dado que está compuesta con `fplToNNF` podemos asegurar que el resultado será de tipo `Just`, y por tanto podemos extraer f sin pérdida de generalidad con `Maybe.withDefault Insat`(si fuese `Nothing` devolvería `Insat`). De forma análoga para la FND en la función `fplToDNF`.

Finalmente, se ofrecen también dos funciones para la reducción de la satisfactibilidad y validez de las fórmulas a través de las Formas Normales Disyuntiva y Conjuntiva, respectivamente. Para ello es suficiente con aplicar las propiedades:

- Una fórmula en FND, $F = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} L_{ij}$, es satisfactible si y sólo si alguna de las conjunciones no contiene literales complementarios.

- Una fórmula en FNC, $F = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} L_{ij}$, es tautología (o lógicamente válida) si y sólo si todas las disyunciones contienen un par de literales complementarios.

Las funciones `dnfAsLiteralSets` y `cnfAsLiteralsSets` proporcionan métodos para representar las fórmulas en FND y FNC, respectivamente, como un conjunto de conjuntos de literales. No vamos a presentar esta implementación ya que es análoga a la que veremos próximamente en la transformación clausal. De hecho, `cnfAsLiteralsSets` proporciona precisamente un conjunto de cláusulas (aunque en LogicUS toman una representación distinta, por lo que se presentan de forma separada).

A través de estas funciones se proporcionan los métodos de reducción de satisfactibilidad, validez (y también extracción de modelos), siguiendo los teoremas previos. Para ello, la función auxiliar `hasContraryLiterals` indica si un conjunto tiene un par de literales complementarios o no. En esta función se hace uso de la evaluación perezosa para que, si en algún momento se encuentra un literal cuyo complementario pertenece también al conjunto de literales, la función devuelva `True` sin necesidad de realizar la evaluación completa. Para la extracción de modelos, basta utilizar los conjuntos de literales de la forma FND que no contengan literales complementarios y, a partir de ellos, generar todos los modelos, recurriendo a la función `interpretationsFromSymbolsAndLiterals` de igual forma a como se realizaba con las hojas abiertas del Tablero Semántico. En el *código 4.16* se presentan las funciones previamente descritas cuya implementación es directa a partir de las propiedades y cuestiones consideradas.

Antes de presentar el siguiente módulo sobre la definición de cláusulas proposicionales, se expone en la *figura 4.5* un caso de uso de las funciones descritas.

```
fplInteriorizeAllDisj : FormulaPL -> Maybe FormulaPL
fplInteriorizeAllDisj f =
    case f of
        Atom _ ->
            Just f

        Disj (Conj f1 f2) g ->
            fplInteriorizeAllDisj <| Conj (Disj f1 g) (Disj f2 g)

        Disj g (Conj f1 f2) ->
            fplInteriorizeAllDisj <| Conj (Disj g f1) (Disj g f2)

        Conj f1 f2 ->
            Maybe.map2 Conj (fplInteriorizeAllDisj f1) (fplInteriorizeAllDisj f2)

        Disj f1 f2 ->
            let
                g1 =
                    fplInteriorizeAllDisj f1

                g2 =
                    fplInteriorizeAllDisj f2
            in
            case g1 of
                Just (Conj x1 y1) ->
                    case g2 of
                        Just (Conj x2 y2) ->
                            fplInteriorizeAllDisj <| Disj (Conj x1 y1) (Conj x2 y2)

                        Just x2 ->
                            fplInteriorizeAllDisj <| Disj (Conj x1 y1) x2

                        _ ->
                            Nothing

                Just x1 ->
                    case g2 of
                        Just (Conj x2 y2) ->
                            fplInteriorizeAllDisj <| Disj x1 (Conj x2 y2)
```

```
                        Just x2 ->
                            Just <| Disj x1 x2

                        _ ->
                            Nothing

                _ ->
                    Nothing

        Neg (Atom _) ->
            Just f

        Insat ->
            Just Insat

        Taut ->
            Just Taut

        _ ->
            Nothing


fplToCNF : FormulaPL -> FormulaPL
fplToCNF f =
    Maybe.withDefault Insat <| fplInteriorizeAllDisj <| fplToNNF f

fplToDNF : FormulaPL -> FormulaPL
fplToDNF f =
    Maybe.withDefault Insat <| fplInteriorizeAllConj <| fplToNNF f
```

Código 4.15: Función de paso a CNF y DNF en LogicUS

```
hasContraryLiterals : List FormulaPL -> Bool
hasContraryLiterals ls =
    case ls of
        [] ->
            False

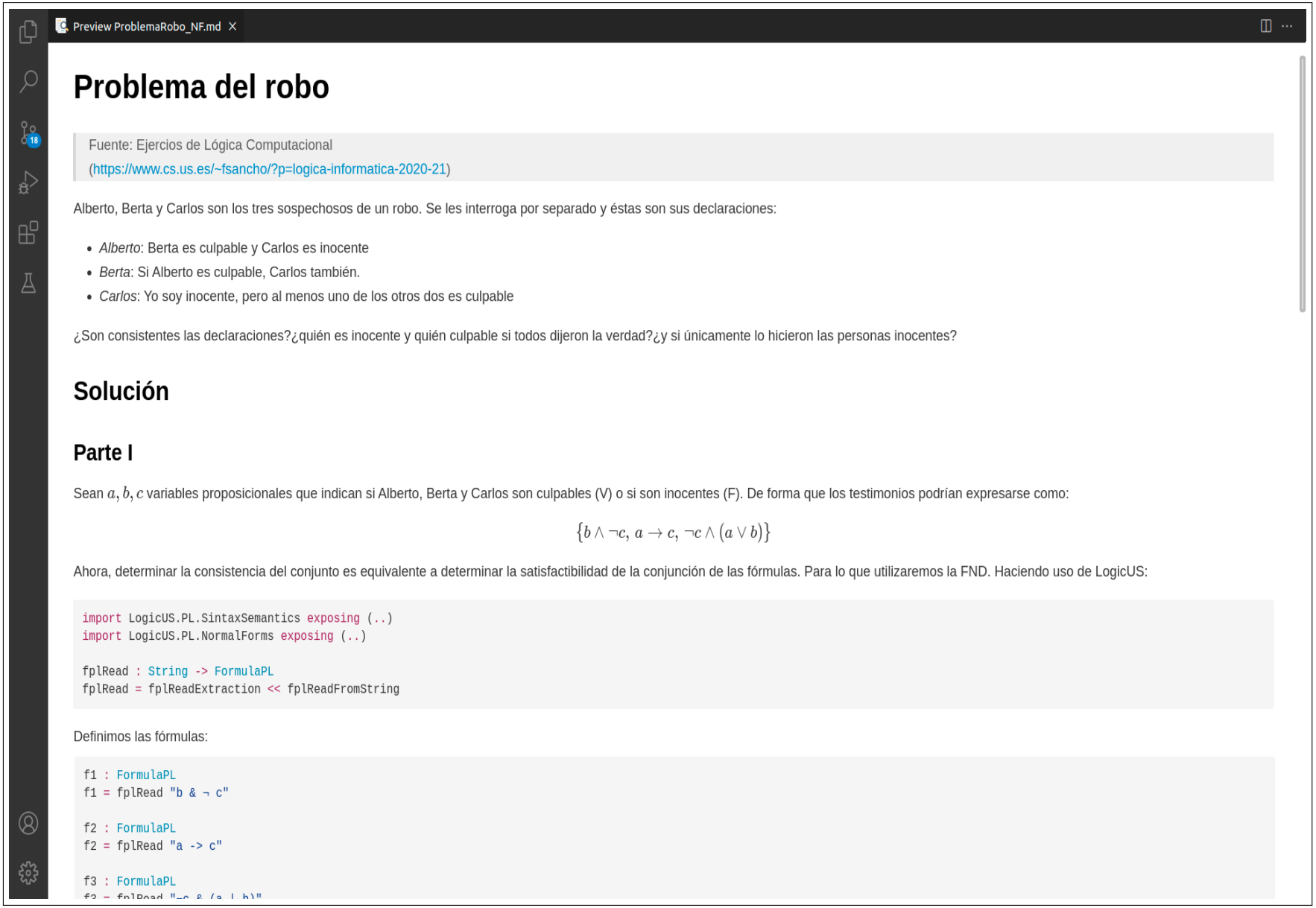
        x :: xs ->
            List.member (PL_SS.fplNegation x) xs || hasContraryLiterals xs

fplSatisfiabilityDNF : FormulaPL -> Bool
fplSatisfiabilityDNF f =
    let
        fls =
            Maybe.withDefault [] <| dnfAsLiteralSets <| fplToDNF f
    in
    List.any (not << hasContraryLiterals) fls

fplModelsDNF : FormulaPL -> List Interpretation
fplModelsDNF f =
    let
        fls =
            Maybe.withDefault [] <| dnfAsLiteralSets <| fplToDNF f
    in
    uniqueConcatList []
        <| List.concat
            <| List.map
                (PL_SS.interpretationsFromSymbolsAndLiterals (PL_SS.fplSymbols f))
                (List.filter (not << hasContraryLiterals) fls)

fplValidityCNF : FormulaPL -> Bool
fplValidityCNF f =
    let
        fls =
            Maybe.withDefault [] <| cnfAsLiteralSets <| fplToCNF f
    in
    List.all hasContraryLiterals fls
```

Código 4.16: Satisfactibilidad, validez y modelos con formas normales en LogicUS


Preview ProblemaRobo_NF.md
Problema del robo
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Alberto, Berta y Carlos son los tres sospechosos de un robo. Se les interroga por separado y éstas son sus declaraciones:
Alberto: Berta es culpable y Carlos es inocente
Berta: Si Alberto es culpable, Carlos también.
Carlos: Yo soy inocente, pero al menos uno de los otros dos es culpable
¿Son consistentes las declaraciones?¿quién es inocente y quién culpable si todos dijeron la verdad?¿y si únicamente lo hicieron las personas inocentes?
Solución
Parte I
Sean a, b, c variables proposicionales que indican si Alberto, Berta y Carlos son culpables (V) o si son inocentes (F). De forma que los testimonios podrían expresarse como:
$$\{b \wedge \neg c,\ a \to c,\ \neg c \wedge (a \vee b)\}$$
Ahora, determinar la consistencia del conjunto es equivalente a determinar la satisfactibilidad de la conjunción de las fórmulas. Para lo que utilizaremos la FND. Haciendo uso de LogicUS:
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.PL.NormalForms exposing (..)
fplRead : String -> FormulaPL
fplRead = fplReadExtraction << fplReadFromString
Definimos las fórmulas:
f1 : FormulaPL
f1 = fplRead "b & ¬ c"
f2 : FormulaPL
f2 = fplRead "a -> c"
f3 : FormulaPL

$$(b \wedge \neg c), (a \rightarrow c), (\neg c \wedge (a \vee b))$$

Entonces ahora para determinar la satisfactibilidad podemos hacerlo a través de la función `fplSatisfiabilityDNF` y la conjunción de las fórmulas del conjunto:

```
g : FormulaPL
g = splConjunction [f1, f2, f3]
```

que corresponde a:

$$(((b \wedge \neg c) \wedge (a \rightarrow c)) \wedge (\neg c \wedge (a \vee b)))$$

De forma que veamos si g es satisfactible. Vamos a hacerlo pasos a paso, en vez de utilizar directamente `fplSatisfiabilityDNF`:

1. Hallamos una FND de g:

```
g1 : FormulaPL
g1 = fplToDNF g
```

$$(((((b \wedge \neg c) \wedge \neg a) \wedge (\neg c \wedge a)) \vee (((b \wedge \neg c) \wedge \neg a) \wedge (\neg c \wedge b))) \vee ((((b \wedge \neg c) \wedge c) \wedge (\neg c \wedge a)) \vee (((b \wedge \neg c) \wedge c) \wedge (\neg c \wedge b))))$$

O como conjunto de conjuntos de literales:

```
g1_ls : Maybe (List SetPL)
g1_ls = dnfAsLiteralSets g1
```

$$\{\neg a, b, \neg c, a\}, \{\neg a, b, \neg c\}, \{b, c, \neg c, a\}, \{b, c, \neg c\}$$

De forma que la fórmula, y por ende, el conjunto es consistente ya que existe al menos un conjunto (de hecho sólo existe 1) que no posee literales complementarios. Obsérvese también que dado que se ha obtenido un único conjunto sin literales complementarios en el que además aparecen todos los símbolos proposicionales entonces éste proporciona el único modelo para la fórmula. De forma que es fácil deducir que, si todos dijeron la verdad, la culpable fue Berta.

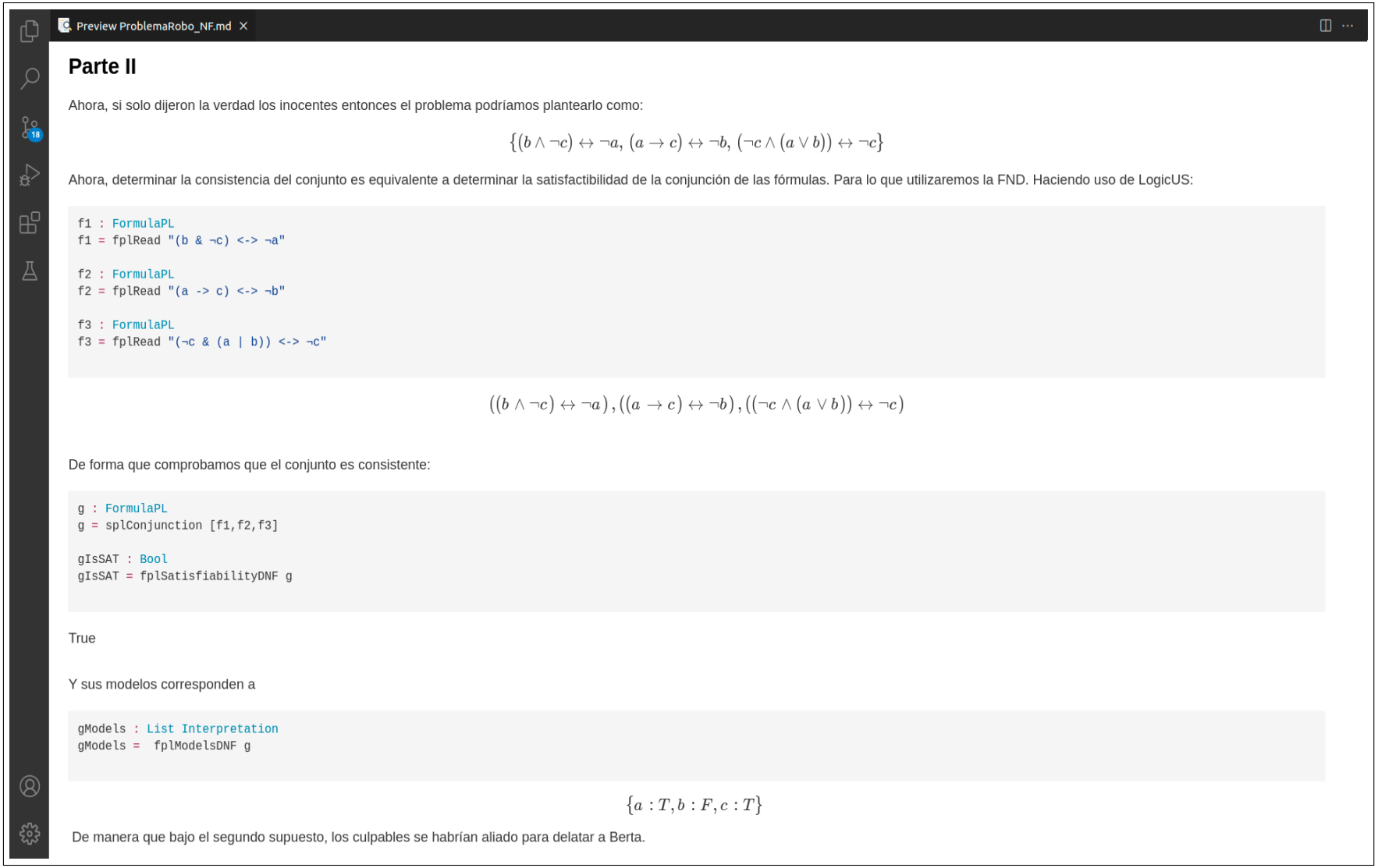


Figura 4.5: Ejemplo de uso del módulo LogicUS.PL.NormalForms
Fuente propia. Creada con litvis

4.2.6. LogicUS.Clauses

Si bien en el punto anterior intordujimos las formas normales y presentamos el cálculo de la FNC, base del cálculo de cláusas, no incluímos en el mismo la presentación de las mismas ya que no forman parte de dicho módulo. Estos es así debido a que la representación adoptada para las mismas es independiente de los literales que corresponden a fórmulas. De forma que vamos a presentar, análogamente a como lo hicimos para las fórmulas, los aspectos sintácticos y semánticos de las cláusulas.

Dado que Elm no permite crear funciones cuyos elementos de entrada sean de un subtipo concreto, es preferible adoptar otra representación para los literales. Nótese que un literal corresponde a un átomo negado o no, de forma que puede representarse mediante una tupla en el que el primer elemento corresponde al símbolo del átomo (de tipo `String`) y el segundo a un elemento que indique si el literal es positivo (1) o negativo (0) (de tipo `Bool`). De forma que las cláusulas estarían definidas como conjuntos de literales.

Sin embargo, Elm no contempla el tipo *Bool* dentro de la clase `comparable` por lo que, al no ser posible definir conjuntos sobre los literales propuestos, los conjuntos de literales son definidos como listas en las que deberemos ir comprobando la unicidad de sus elementos. Para facilitar esta tarea, éstos aparecerán ordenados de forma alfabética y anteponiendo los positivos a los negativos en caso de igualdad en el símbolo. Estos conjuntos de literales serán interpretados (tal y como marca la lógica formal) como disyunciones de literales. Así mismo, se definen los conjuntos de cláusulas como listas de cláusulas, principalmente por el motivo previamente expuesto. Estos conjuntos son tomados como conjunciones de acuerdo a lo expuesto en el ámbito teórico.

En el *código 4.17* se presentan todas las implementaciones relacionadas. Nótese que, aunque no lo hemos explicitado, existe otro tipo básico en Elm, correspondiente al tipo `Order`, que permite establecer una relación de orden en un tipo de elemento, y que puede ser utilizada para ordenar una lista por medio de la función `sortWith`.

```
type alias ClausePLLiteral = ( PSymb, Bool )

type alias ClausePL = List ClausePLLiteral

type alias ClausePLSet = List ClausePL

compareClausePLLiterals : ( PSymb, Bool ) -> ( PSymb, Bool ) -> Order
compareClausePLLiterals ( symb1, sign1 ) ( symb2, sign2 ) =
  case ( sign1, sign2 ) of
    ( True, False ) -> LT
    ( False, True ) -> GT
    \_ -> compare symb1 symb2

cplSort : ClausePL -> ClausePL
cplSort cs = List.sortWith compareClausePLLiterals cs
```

Código 4.17: Definición de Cláusulas en LogicUS

En cuanto al aspecto semántico, dada la interpretación de las cláusulas y conjuntos comentada previamente, y teniendo en cuenta lo visto en cuanto a las formas normales, resulta fácil establecer los conceptos de evaluación de conjuntos de cláusulas, satisfactibilidad, validez, modelos, etc. Un aspecto que no se ha visto previamente es la simplificación de conjuntos de cláusulas mediante la *subsunción*. Se dice que una cláusula c_1 subsume a otra c_2 si c_1 está enteramente contenida en c_2. Nótese que en dicho caso si c_1 es satisfactible también lo es c_2 dado el carácter disyuntivo de las cláusulas. Esto, permite simplificar los conjuntos clausales eliminando aquellas cláusulas que sean subsumidas por otras. En este sentido, también es interesante eliminar las cláusulas correspondientes a tautologías (dado el carácter conjuntivo de los conjuntos y la propiedad $A \wedge \top \equiv A$) esto es realizable manteniéndose la equivalencia.

La evaluación de las cláusulas respecto de una interpretación (dada como una lista de símbolos proposicionales) puede hacerse sencillamente comprobando si alguno de los símbolos de los literales

positivos pertenecen a la interpretación o si alguno de los negativos no se incluyen en la misma. Para el cálculo de las posibles interpretaciones basta tomar el conjunto de símbolos de las cláusulas y calcular todos los posibles subconjuntos (*powerset*), de forma que su evaluación permite calcular de forma sencilla (aunque no eficiente) los modelos y contramodelos del conjunto.

En el *código 4.18* se expone las implementaciones correspondientes a algunos de los aspectos semánticos comentados.

```
cplSubsumes : ClausePL -> ClausePL -> Bool
cplSubsumes c1 c2 =
    List.all (\x -> List.member x c2) c1

csplRemoveSubsumedClauses : ClausePLSet -> ClausePLSet
csplRemoveSubsumedClauses cs =
    List.foldl
        (\c ac ->
            if List.any (\x -> cplSubsumes x c) ac then
                ac

            else
                List.filter (not << cplSubsumes c) ac ++ [ c ]
        )
        []
        (List.map cplSort cs)

cplIsTautology : ClausePL -> Bool
cplIsTautology c =
    List.any (\( psymb, sign ) -> List.member ( psymb, not sign ) c) c

csplRemoveTautClauses : ClausePLSet -> ClausePLSet
csplRemoveTautClauses cs =
    List.filter (not << cplIsTautology) <| List.map cplSort cs

csplSymbols : ClausePLSet -> List PSymb
csplSymbols cs =
    List.sort <|
        List.foldl
            (\c ac -> uniqueConcatList  ac (List.sort <| List.map Tuple.first c))
            []
            cs

csplInterpretations : ClausePLSet -> List Interpretation
csplInterpretations cs =
    List.sort <| powerset <| csplSymbols cs

cplValuation : ClausePL -> Interpretation -> Bool
cplValuation c i =
    List.any
        (\( symb, sign ) ->
            if sign then
                List.member symb i

            else
                not <| List.member symb i
        )
        c

csplValuation : ClausePLSet -> Interpretation -> Bool
csplValuation cs i =
    List.all (\c -> cplValuation c i) cs

csplModels : ClausePLSet -> List Interpretation
csplModels cs =
    List.filter (\i -> csplValuation cs i) <| csplInterpretations cs

csplIsTaut : ClausePLSet -> Bool
csplIsTaut cs =
    List.all (\c -> cplIsTautology c) cs
```

```
csplIsSat : ClausePLSet -> Bool
csplIsSat cs =
    not <| List.isEmpty <| csplModels cs

csplIsInsat : ClausePLSet -> Bool
csplIsInsat cs =
    List.isEmpty <| csplModels cs
```

Código 4.18: Semántica de Cláusulas en LogicUS

Además de todo lo expuesto, se proveen los mecanismos necesarios para definir la cláusulas a través de cadenas siguiendo la expresión regular:

```
{((¬)?[a-z][a-zA-Z]*(_{[0-9]+(,[0-9]+)*})?)
    (,((¬)?[a-z][a-zA-Z]*(_{[0-9]+(,[0-9]+)*})?))*)?}
```

que corresponde a literales (variables o negación de variables), separadas por comas y encerradas por llaves.

Entre las implementaciones ofrecidas también hay una función de transformación a cláusulas de las fórmulas mediante una FNC equivalente. Para ello la función `cplFromCNF` establece el mecanismo de transformación de FNC a cláusulas:

- `cplFromCNF` es una función parcial que está únicamente definida para fórmulas en FNC, luego si en la fórmula existen implicaciones, equivalencias dobles negaciones, o una conjunción interior a una disyunción entonces la fórmula no estaría en FNC y, en consecuencia, la función devolvería `Nothing`.

- Si la fórmula es de tipo conjuntivo entonces se devuelve un conjunto con la lista de las cláusulas derivadas de las componentes de la mismas, calculadas mediante una llamada recursiva a la misma función, eliminando las repetidas.

- Si la fórmula es de tipo disyuntivo entonces se unen las cláusulas obtenidas de cada una de las componentes en una única, eliminando los literales repetidos y ordenándolos según el orden establecido para los literales.

- Si la fórmula corresponde a un átomo o a su negación entonces se devuelve un conjunto con una única cláusula con el literal expresado como una tupla con la primera de sus componentes el símbolo y la segunda un booleano (verdadero si es un átomo, falso si es la negación).

- Si la fórmula corresponde a la fórmula válida ($\top$) se devuelve un conjunto vacío y si corresponde a la insatisfactible ($\bot$) se devuelve un conjunto con la cláusula vacía.

Dichas acciones se presentan detalladas en el *código 4.19* junto a las funciones generales de transformación a cláusulas de fórmulas y conjuntos proposicionales `fplToClauses` y `splToClauses`, respectivamente.

Además de las funciones comentadas, también se ofrecen funciones de representación, tanto en formato cadena como en formato Latex, para su correcta visualización, aunque su implementación no goza de un especial interés por lo que no expondrán en detalle.

```
cplFromCNF : FormulaPL -> Maybe ClausePLSet
cplFromCNF f =
    let
        cplFromCNFAux g =
            case g of
                Atom symb ->
                    Just [ [ ( symb, True ) ] ]

                Neg (Atom symb) ->
                    Just [ [ ( symb, False ) ] ]

                Disj g1 g2 ->
                    Maybe.map (\c -> [ c ]) <|
                        Maybe.map cplSort <|
                            Maybe.map2 uniqueConcatList
                                (Maybe.map List.concat <| cplFromCNFAux g1)
                                (Maybe.map List.concat <| cplFromCNFAux g2)

                Insat ->
                    Just [ [] ]

                Taut ->
                    Just []

                _ ->
                    Nothing
    in
    case f of
        Conj f1 f2 ->
            Maybe.map2 uniqueConcatList (cplFromCNF f1) (cplFromCNF f2)

        Atom symb ->
            Just [ [ ( symb, True ) ] ]

        Neg (Atom symb) ->
            Just [ [ ( symb, False ) ] ]

        Disj f1 f2 ->
            Maybe.map (\c -> [ c ]) <|
                Maybe.map cplSort <|
                    Maybe.map2 uniqueConcatList
                        (Maybe.map List.concat <| cplFromCNFAux f1)
                        (Maybe.map List.concat <| cplFromCNFAux f2)

        Insat ->
            Just [ [] ]

        Taut ->
            Just []

        _ ->
            Nothing

fplToClauses : FormulaPL -> ClausePLSet
fplToClauses f =
    Maybe.withDefault [ [] ] <| cplFromCNF <| PL_NF.fplToCNF f

splToClauses : SetPL -> ClausePLSet
splToClauses fs =
    List.foldl (\f ac -> uniqueConcatList ac <| fplToClauses f) [] <| fs
```

Código 4.19: Transformación de fórmulas en cláusulas en LogicUS

Para finalizar, en la *figura 4.6* presentamos el mismo ejemplo de uso presentado con formas normales pero con la resolución mediante el uso de cláusulas.

En los dos siguientes puntos se mostrará la implementación de los algoritmos DPLL y Resolución Proposicional, que harán uso de las cláusulas para determinar la (in)satisfactibilidad de un conjunto de fórmulas (a través de cláusulas) y en el caso de DPLL también proporcionará modelos del mismo.


Preview ProblemaRobo_CL2.md
Problema del robo
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Alberto, Berta y Carlos son los tres sospechosos de un robo. Se les interroga por separado y éstas son sus declaraciones:
Alberto: Berta es culpable y Carlos es inocente
Berta: Si Alberto es culpable, Carlos también.
Carlos: Yo soy inocente, pero al menos uno de los otros dos es culpable
¿Son consistentes las declaraciones?¿quién es inocente y quién culpable si todos dijeron la verdad?¿y si únicamente lo hicieron las personas inocentes?
Solución
Sean a, b, c variables proposicionales que indican si Alberto, Berta y Carlos son culpables (V) o si son inocentes (F). De forma que los testimonios podrían expresarse como:
$$\{b \wedge \neg c,\ a \rightarrow c,\ \neg c \wedge (a \vee b)\}$$
Ahora, determinar la consistencia del conjunto es equivalente a determinar la satisfactibilidad de la conjunción de las fórmulas. Para lo que utilizaremos la FND. Haciendo uso de LogicUS:
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.PL.Clauses exposing (..)
fplRead : String -> FormulaPL
fplRead = fplReadExtraction << fplReadFromString
Definimos las fórmulas:
f1 : FormulaPL
f1 = fplRead "b & ¬ c"
f2 : FormulaPL
f2 = fplRead "a -> c"
f3 : FormulaPL
f3 = fplRead "¬c & (a | b)"

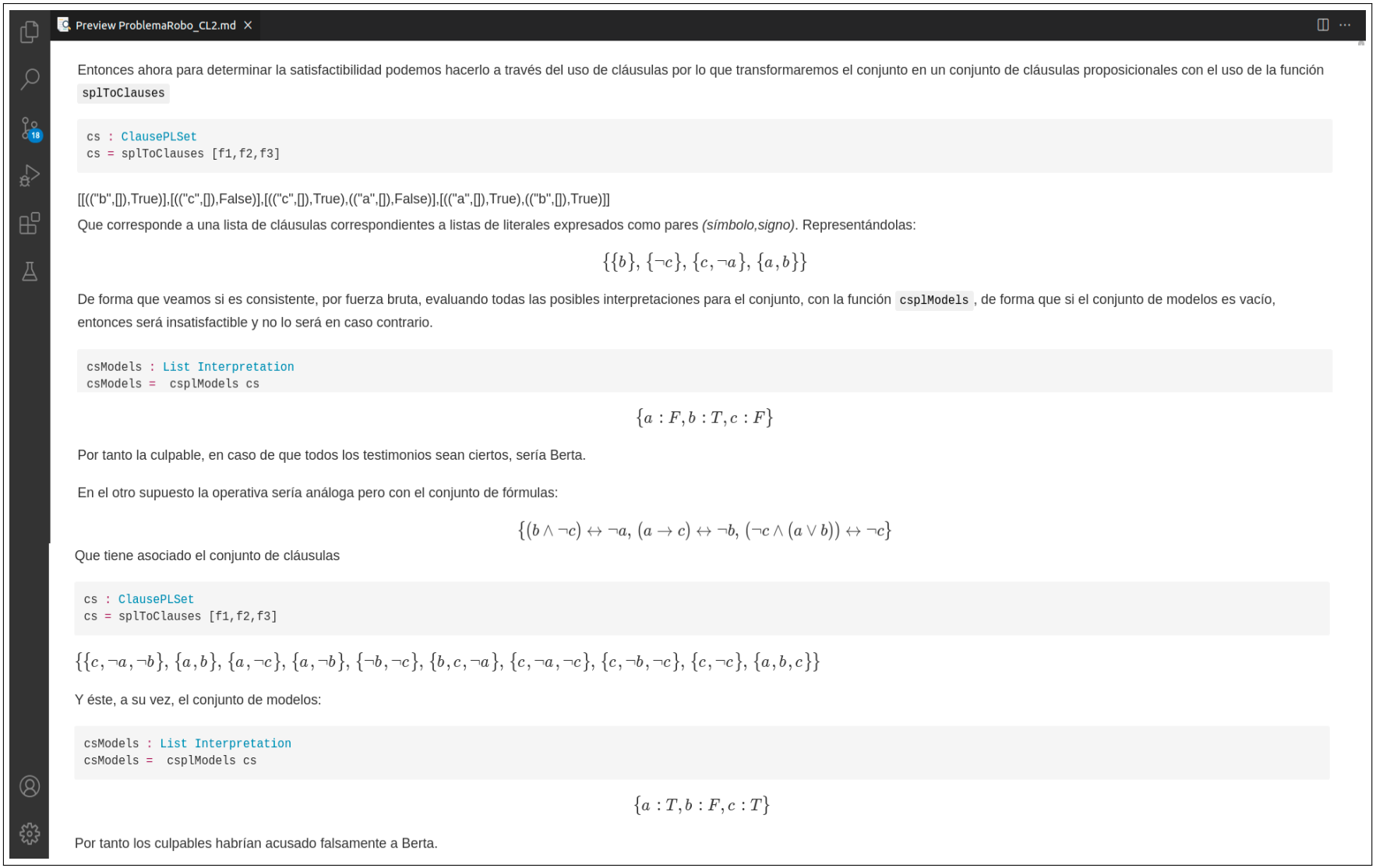


Figura 4.6: Ejemplo de uso de LogicUS.PL.Clauses
Fuente propia. Creada con litvis

4.2.7. LogicUS.PL.DPLL

En este módulo se implementa el algoritmo David-Putnam-Logemann-Loveland (DPLL) en formato de tableau (árbol), así como la obtención de modelos a partir del mismo y la representación tanto en formato cadena como en formato DOT (visualizable con Viz).

En el *algoritmo 2.4* presentamos una versión recursiva de este algoritmo que será usada directamente para trasladar las definiciones sin necesidad de realizar una conversión previa. Aunque sí lo vamos a presentar en formato de tableau en el *algoritmo 4.2*.

A partir de ahí, el paso a la programación declarativa es casi inmediato sin más que especificar la heurística en la elección del literal en caso de que el conjunto no contenga cláusulas unitarias. En nuestra implementación es elegido el literal con más ocurrencias en las cláusulas del conjunto. Además de ello, en cada paso del algoritmo son eliminados del conjunto considerado las cláusulas subsumidas por otras dentro del conjunto (haciendo uso de la función `csplRemoveSubsumedClauses`).

Algoritmo 4.2: Algoritmo Davis–Putnam–Logemann–Loveland (Tableau)

DPLL (U)**:**

Input : U: Un conjunto de cláusulas proposicionales
Output: T : Un árbol que indica el desarrollo del algoritmo.

1. Hacer r la raíz de T y etiquetarla con $r = U$
2. Si U contiene la cláusula vacía, marcar el nodo cerrado y devolver T.
3. Si U es vacío, marcar el nodo abierto y devolver T
4. Si no si U posee unidades, tomar una de ellas $\{L\}$, hacer $T' = DPLL(Propagate(L, U)$ y acoplar T' a r como único descendiente.
5. Si no, tomar un literal L presente en U y hacer:
   - $(U_1, U_2) = Divide(L, U)$
   - $T' = DPLL(U_1)$, $T'' = DPLL(U_2)$
   - Añadir T' y T'' a r como descendientes.

end

Prop (L, U)**:**

Input : L: Un literal, U: Un conjunto de cláusulas proposicionales.
Output: U': Un conjunto de cláusulas en el que no aparece L

1. Hacer $U' = U$
2. Eliminar de U' aquellas cláusulas en las que aparece L.
3. Eliminar de todas las cláusulas de U' la negación de L.
4. Devolver U'

end

Divide (L, U)**:**

Input : L: Un literal, U: Un conjunto de cláusulas proposicionales.
Output: (U', U''): Un conjunto de cláusulas en el que no aparece L

1. Hacer $U' = Propagate(L, U)$ y $U'' = Propagate(\neg L, U)$
2. Devolver (U', U'')

end

La función `dpll`, cuya implementación se presenta en el código, define precisamente la construcción del tablero. En dicha función destacamos algunos detalles:

- `propagation`: Recorre el conjunto de cláusulas mediante un plegado (`foldl`) de forma que, si la cláusula contiene el literal, entonces dicha cláusula no es añadida al acumulador, en otro caso, se añade la cláusula eliminando la negación del literal si ésta la contuviese. Finalmente, se eliminan las cláusulas subsumidas por otras.
- `psymbsOcurrFreq`: Calcula las ocurrencias de cada uno de los literales en el conjunto considerado. Para ello, se obtienen todas las ocurrencias de todos los literales del conjunto (concatenación de todas las cláusulas), se agrupan según el símbolo de los literales (haciendo uso de la función `gatherEquals`) y se asigna a cada símbolo la longitud de la lista de los valores agrupados, que es exactamente el número de ocurrencias de cada símbolo. De esta forma podemos obtener el literal con más ocurrencias simplemente como el máximo de la lista por el segundo elemento (`LE.maximumBy Tuple.second psymbsOcurrFreq`).
- Se sigue la misma numeración de los nodos que en el método de Tableros Semánticos, tomando como valor del descendiente una unidad más para el primer descendiente, y en caso de dos descendientes se calcula primero la rama del primer descendiente, y se toma el número del último nodo de la rama más uno para el valor del segundo descendiente (se corresponde con una búsqueda en profundidad). Sería fácil adaptar esta numeración para poder hacer uso del paralelismo.
- El algoritmo `dpllAux` calcula una lista con los nodos etiquetados con los conjuntos de cláusulas correspondientes, y otra con las aristas del árbol etiquetadas con los literales considerados en la propagación. A continuación, el grafo se crea simplemente haciendo uso de la función `fromNodesAndEdges`.

Junto con la función de cálculo del tablero DPLL (*código 4.20*) también se dispone de la función `dpllTableauModels`, que calcula los modelos del conjunto de cláusulas dados los símbolos de referencia (dado que es posible que en la primera simplificación alguno haya desaparecido como consecuencia de la simplificación del conjunto por subsunción y eliminación de tautologías). Para ello, se toman las hojas abiertas y se calculan los caminos, de forma que las etiquetas de las aristas dan precisamente un conjunto de literales a partir del cual se extraen los modelos de la hoja. Para la reconstrucción de las ramas, la función `getDPLLPathToWithpredecessors` busca desde la hoja los sucesivos padres hasta llegar a la raíz, y almacena en una lista los literales de las aristas consideradas en ese recorrido.

```
dpll : List ClausePL -> DPLLTableau
dpll cs =
    let
        dpllAux clauses nid =
            let
                propagation ( lsymb, lsign ) =
                    List.foldl
                        (\x ac ->
                            if List.member ( lsymb, lsign ) x then
                                ac

                            else
                                ac ++ [ List.filter (\( ysymb, _ ) -> ysymb /= lsymb) x ]
                        )
                        []
                        clauses
                        |> PL_CL.csplRemoveSubsumedClauses
            in
            if List.isEmpty clauses then
                ( [ Node nid ( 1, clauses ) ], [] )

            else if List.any (\c -> List.isEmpty c) clauses then
                ( [ Node nid ( -1, clauses ) ], [] )

            else
                case List.head <| List.filter (\x -> List.length x == 1) clauses of
                    Just [ l ] ->
                        let
```

```
                            new_clauses =
                                propagation l
                        in
                        let
                            ( nodes, edges ) =
                                dpllAux new_clauses (nid + 1)
                        in
                        (Node nid ( 0, clauses ) :: nodes, Edge nid (nid + 1) l :: edges)

                    _ ->
                        let
                            psymbsOcurrFreq =
                                List.concat clauses
                                    |> LE.gatherEqualsBy Tuple.first
                                    |> List.map
                                        (\( x, xs ) -> ( Tuple.first x, List.length xs ))
                        in
                        case LE.maximumBy Tuple.second psymbsOcurrFreq of
                            Just ( lsymb, _ ) ->
                                let
                                    new_clauses1 =
                                        propagation ( lsymb, True )

                                    new_clauses2 =
                                        propagation ( lsymb, False )
                                in
                                let
                                    ( nodes1, edges1 ) =
                                        dpllAux new_clauses1 (nid + 1)
                                in
                                let
                                    next_id =
                                        nid + List.length nodes1 + 1
                                in
                                let
                                    ( nodes2, edges2 ) =
                                        dpllAux new_clauses2 next_id
                                in
                                ( Node nid ( 0, clauses ) :: (nodes1 ++ nodes2),
                                [ Edge nid (nid + 1) ( lsymb, True ),
                                Edge nid next_id ( lsymb, False ) ] ++ edges1 ++ edges2 )

                            Nothing ->
                                ( [ Node nid ( -1, clauses ) ], [] )

        new_cs =
            PL_CL.csplRemoveSubsumedClauses <| PL_CL.csplRemoveTautClauses <| cs
    in
    let
        ( nodes, edges ) =
            dpllAux new_cs 0
    in
    Graph.fromNodesAndEdges nodes edges



dpllTableauModels : List PSymb -> DPLLTableau -> List Interpretation
dpllTableauModels refSymbs dt =
    let
        openLeaf =
            List.foldl
                (\x ac ->
                    if Tuple.first x.label == 1 then
                        ac ++ [ x.id ]

                    else
                        ac
                )
                []
            <|
                Graph.nodes dt
```

```
        predecessors =
            IntDict.fromList <| List.map (\x -> ( x.to, ( x.from, x.label ) ))
                <| Graph.edges dt

        symbs =
            LE.unique <|
                ((Maybe.withDefault (Node 0 ( 0, [] ))
                    <| List.head
                    <| Graph.nodes dt
                 ).label
                    |> Tuple.second |> PL_CL.csplSymbols) ++ refSymbs
    in
    let
        getDPLLPathToWithpredecessors nid =
            case IntDict.get nid predecessors of
                Nothing ->
                    []

                Just ( anid, l ) ->
                    getDPLLPathToWithpredecessors anid ++ [ PL_CL.clauseLitToLiteral l ]
    in
    List.sort <|
        LE.unique <|
            List.concat <|
                List.map
                    (\nid ->
                        PL_SS.interpretationsFromSymbolsAndLiterals
                            symbs
                            (getDPLLPathToWithpredecessors nid)
                    ) openLeaf
```

Código 4.20: Algoritmo DPLL en LogicUS

En el módulo también se exponen los mecanismos apropiados para la representación de los tableros DPLL mediante las funciones `dpllTableauToString` y `dpllTableauToDOT`.

```
dpllTableauToDOT : DPLLTableau -> String
dpllTableauToDOT g =
  let
    myStyles =
      { defaultStyles | node = "shape=box, color=white, fontcolor=black",
          edge = "dir=none, color=blue, fontcolor=blue" }

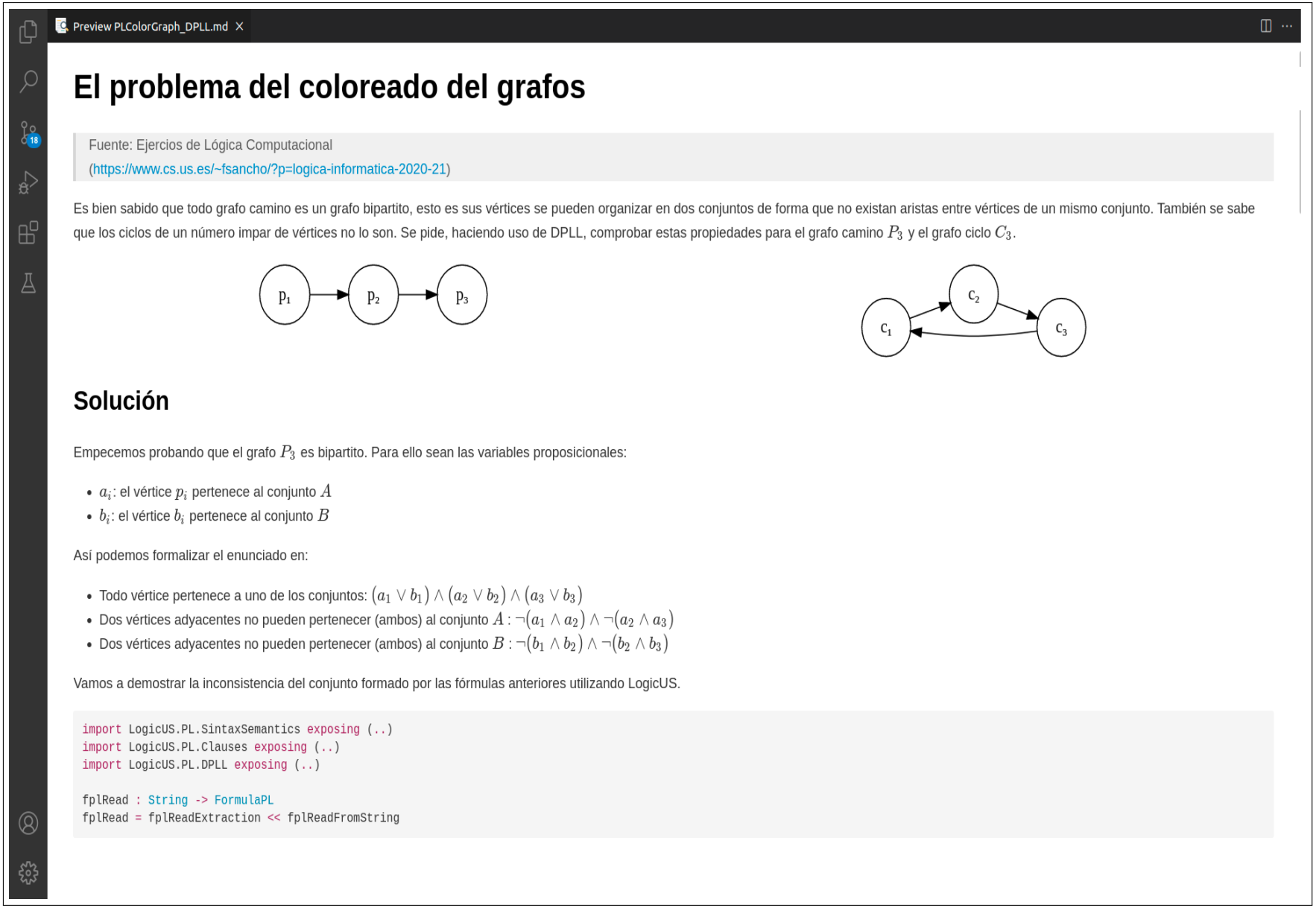
    toStringNode =
      \( i, cs ) ->
        case i of
          0 ->
            Just <| String.join ", " <| List.map PL_CL.cplToString cs

          1 ->
            Just "○"
          _ ->
            Just "□"

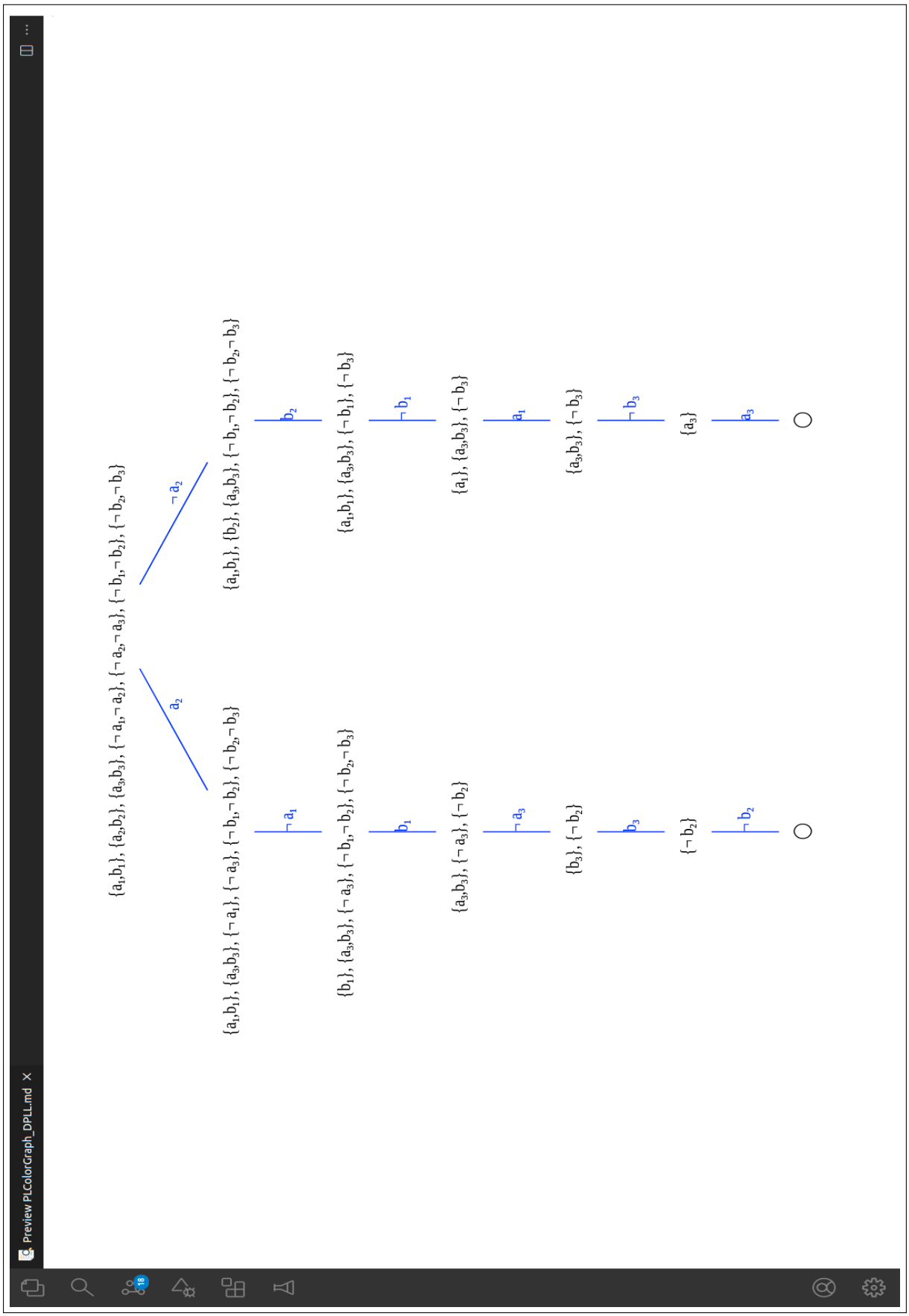
    toStringEdge =
      \l -> (Just << PL_SS.fplToString << PL_CL.clauseLitToLiteral) l
  in
  String.replace "\"" ">" <| String.replace "=\"" "=<"
    <| Graph.DOT.outputWithStyles myStyles toStringNode toStringEdge g
```

Código 4.21: Representación de DPLL Tableaux

Finalmente, en la *figura 4.7* se presenta un ejemplo que ilustra el uso de las funciones del módulo.


Preview PLColorGraph_DPLL.md
El problema del coloreado del grafos
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Es bien sabido que todo grafo camino es un grafo bipartito, esto es sus vértices se pueden organizar en dos conjuntos de forma que no existan aristas entre vértices de un mismo conjunto. También se sabe que los ciclos de un número impar de vértices no lo son. Se pide, haciendo uso de DPLL, comprobar estas propiedades para el grafo camino P_3 y el grafo ciclo C_3.
p1
p2
p3
c1
c2
c3
Solución
Empecemos probando que el grafo P_3 es bipartito. Para ello sean las variables proposicionales:
a_i: el vértice p_i pertenece al conjunto A
b_i: el vértice b_i pertenece al conjunto B
Así podemos formalizar el enunciado en:
Todo vértice pertenece a uno de los conjuntos: $(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge (a_3 \vee b_3)$
Dos vértices adyacentes no pueden pertenecer (ambos) al conjunto A : $\neg(a_1 \wedge a_2) \wedge \neg(a_2 \wedge a_3)$
Dos vértices adyacentes no pueden pertenecer (ambos) al conjunto B : $\neg(b_1 \wedge b_2) \wedge \neg(b_2 \wedge b_3)$
Vamos a demostrar la inconsistencia del conjunto formado por las fórmulas anteriores utilizando LogicUS.
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.PL.Clauses exposing (..)
import LogicUS.PL.DPLL exposing (..)
fplRead : String -> FormulaPL
fplRead = fplReadExtraction << fplReadFromString


Preview PLColorGraph_DPLL.md
Definimos las fórmulas:
f1 : FormulaPL
f1 = fplRead "(a_{1} | b_{1}) & (a_{2} | b_{2}) & (a_{3} | b_{3})"
f2 : FormulaPL
f2 = fplRead "¬(a_{1} & a_{2}) & ¬(a_{2} & a_{3})"
f3 : FormulaPL
f3 = fplRead "¬(b_{1} & b_{2}) & ¬(b_{2} & b_{3})"
De forma que le corresponde el conjunto de cláusulas:
cs : ClausePLSet
cs = splToClauses [f1,f2,f3]
$\{\{a_1, b_1\}, \{a_2, b_2\}, \{a_3, b_3\}, \{\neg a_1, \neg a_2\}, \{\neg a_2, \neg a_3\}, \{\neg b_1, \neg b_2\}, \{\neg b_2, \neg b_3\}\}$
Que corresponde a una lista de cláusulas correspondientes a listas de literales expresados como pares (símbolo,signo). Representándolas:
Ahora, aplicando el algoritmo DPLL y representándolo en formato DOT
csTableau : String
csTableau = dpll cs |> dpllTableauToDOT
"digraph G {\n rankdir=TB\n graph []\n node [shape=box, color=white, fontcolor=black]\n edge [dir=none, color=blue, fontcolor=blue]\n\n 0 -> 1 [label=<a2>]\n 0 -> 7 [label=<¬ a2>]\n 1 -> 2 [label=<¬ a1>]\n 2 -> 3 [label=<b1>]\n 3 -> 4 [label=<¬ a3>]\n 4 -> 5 [label=<b3>]\n 5 -> 6 [label=<¬ b2>]\n 7 -> 8 [label=<b2>]\n 8 -> 9 [label=<¬ b1>]\n 9 -> 10 [label=<a1>]\n 10 -> 11 [label=<¬ b3>]\n 11 -> 12 [label=<a3>]\n\n 0 [label=<{a1,b1}, {a2,b2}, {a3,b3}, {¬ a1,¬ a2}, {¬ a2,¬ a3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}>]\n 1 [label=<{a1,b1}, {a3,b3}, {¬ a1}, {¬ a3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}>]\n 2 [label=<{b1}, {a3,b3}, {¬ a3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}>]\n 3 [label=<{a3,b3}, {¬ a3}, {¬ b2}>]\n 4 [label=<{b3}, {¬ b2}>]\n 5 [label=<{¬ b2}>]\n 6 [label=<◯>]\n 7 [label=<{a1,b1}, {b2}, {a3,b3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}>]\n 8 [label=<{a1,b1}, {a3,b3}, {¬ b1}, {¬ b3}>]\n 9 [label=<{a1}, {a3,b3}, {¬ b3}>]\n 10 [label=<{a3,b3}, {¬ b3}>]\n 11 [label=<{a3}>]\n 12 [label=<◯>]\n}"
Representándolo:


Preview PLColorGraph_DPLL.md
{a1,b1}, {a2,b2}, {a3,b3}, {¬ a1,¬ a2}, {¬ a2,¬ a3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}
a2
¬ a2
{a1,b1}, {a3,b3}, {¬ a1}, {¬ a3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}
{a1,b1}, {b2}, {a3,b3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}
¬ a1
b2
{b1}, {a3,b3}, {¬ a3}, {¬ b1,¬ b2}, {¬ b2,¬ b3}
{a1,b1}, {a3,b3}, {¬ b1}, {¬ b3}
b1
¬ b1
{a3,b3}, {¬ a3}, {¬ b2}
{a1}, {a3,b3}, {¬ b3}
¬ a3
a1
{b3}, {¬ b2}
{a3,b3}, {¬ b3}
b3
¬ b3
{¬ b2}
{a3}
¬ b2
a3

Preview PLColorGraph_DPLL.md

Hagámos lo mismo para el grafo C_3. Para ello sean las variables proposicionales:

- a_i: el vértice c_i pertenece al conjunto A
- b_i: el vértice c_i pertenece al conjunto B

Así podemos formalizar el enunciado en:

- Todo vértice pertenece a uno de los conjuntos: $(a_1 \vee b_1) \wedge (a_2 \vee b_2) \wedge (a_3 \vee b_3)$
- Dos vértices adyacentes no pueden pertenecer (ambos) al conjunto A : $\neg(a_1 \wedge a_2) \wedge \neg(a_2 \wedge a_3) \wedge \neg(a_3 \wedge a_1)$
- Dos vértices adyacentes no pueden pertenecer (ambos) al conjunto B : $\neg(b_1 \wedge b_2) \wedge \neg(b_2 \wedge b_3) \wedge \neg(b_3 \wedge b_1)$

Vamos a demostrar la inconsistencia del conjunto formado por las fórmulas anteriores utilizando LogicUS.

Definimos las fórmulas:

```
f1 : FormulaPL
f1 = fplRead "(a_{1} | b_{1}) & (a_{2} | b_{2}) & (a_{3} | b_{3})"

f2 : FormulaPL
f2 = fplRead "¬(a_{1} & a_{2}) & ¬(a_{2} & a_{3}) & ¬(a_{3} & a_{1})"

f3 : FormulaPL
f3 = fplRead "¬(b_{1} & b_{2}) & ¬(b_{2} & b_{3}) & ¬(b_{3} & b_{1})"
```

De forma que le corresponde el conjunto de cláusulas:

```
cs : ClausePLSet
cs = splToClauses [f1,f2,f3]
```

$$\{\{a_1, b_1\}, \{a_2, b_2\}, \{a_3, b_3\}, \{\neg a_1, \neg a_2\}, \{\neg a_2, \neg a_3\}, \{\neg a_1, \neg a_3\}, \{\neg b_1, \neg b_2\}, \{\neg b_2, \neg b_3\}, \{\neg b_1, \neg b_3\}\}$$

Que corresponde a una lista de cláusulas correspondientes a listas de literales expresados como pares *(símbolo,signo)*. Representándolas:

Ahora, aplicando el algoritmo DPLL y representándolo en formato DOT

```
csTableau : String
csTableau =  dpll cs |> dpllTableauToDOT
```

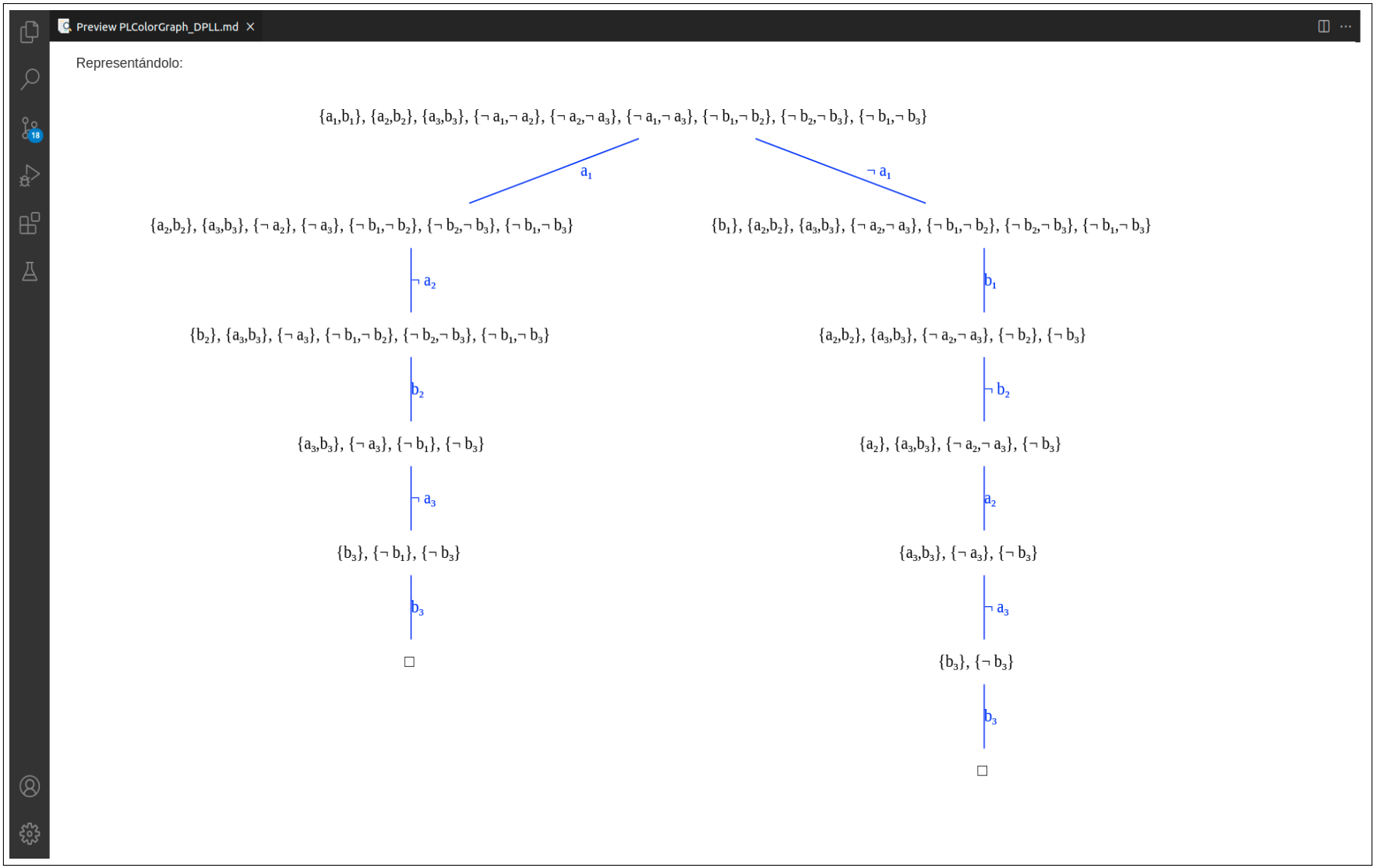


Figura 4.7: Ejemplo de uso de LogicUS.PL.DPLL
Fuente propia. Creada con litvis

4.2.8. LogicUS.PL.Resolution

En este apartado vamos a presentar la implementación del algoritmo de Resolución Proposicional. Distinguimos dos posibles vías: la Resolución por Saturación (Regular), y la resolución como método de búsqueda con estrategias. Ambas vías partiendo de la base de conjuntos de cláusulas.

4.2.9. Resolución por Saturación (Regular)

El *algoritmo 2.5* muestra un mecanismo general de resolución, el algoritmo por saturación, que no toma una sola cláusula sino que realiza todas las posibles resolventes entre las cláusulas existentes en cada momento, lo cual lo hace altamente ineficiente. Aunque este algoritmo se encuentra implementado, vamos a mostrar la versión de Saturación Regular, que toma, además del conjunto de cláusulas, una lista con el orden en el que se exploran las variables, de forma que en cada paso se realizan todas las posibles resolventes pero únicamente por la variable considerada, lo que mejora notablemente la eficiencia del algoritmo (*algoritmo 4.3*).

Algoritmo 4.3: Algoritmo de Resolución Proposicional

Resolution (U)**:**

Input : U: Un conjunto de clausulas proposicionales, vs: Una lista con las variables proposicionales ordenadas según el orden de eliminación.
Output: *Bool* : Que indica si el conjunto es satisfactible o insatisfactible

1. Si $\square \in U$: devolver True (insatisfactible)
2. Si no si $U = \emptyset$: devolver False (satisfactible)
3. Si no:
   - 3.1 Hacer $v = pop(vs)$
   - 3.2 Hacer $(U', U'') = (\{c \in U / v \in symbs(c)\}, \{c \in U / v \notin symbs(c)\})$
   - 3.3 Hacer $U''' = removeIrrelevants(U' \cup allResolventsBySymb(U'', v))$
   - 3.4 Devolver $Resolution(U''',)$

end

La implementación realizada se corresponde fielmente al algoritmo descrito teniendo en cuenta los siguientes detalles:

- `csplRegularResolution` devuelve, además de la insatisfactibilidad del conjunto de cláusulas, una lista con los conjuntos de cláusulas obtenidas en cada paso del algoritmo.
- Aunque se especifique un orden para las variables, la lista dada es completada con las variables presentes en el conjunto de cláusulas y no presentes en la lista de variables.
- En cada paso se eliminan las cláusulas subsumidas y las tautológicas a fin de mejorar la eficiencia del algoritmo.
- La función `csplAllResolventsByPsymb` calcula todas las resolventes entre todas las cláusulas de un conjunto por una variable. Recuérdese que dos cláusulas son resolubles por una variable p si una de ellas contiene p y la otra $\neg p$. Aquellas cláusulas que no contengan el literal no participarán en ninguna resolvente, por lo que resulta ineficiente dárselos a la función `csplAllResolventsByPsymb`. Nótese además que en el siguiente paso hemos de eliminar todas las cláusulas que hacen uso de p, que son justamente las que participan en la resolución. Por ello, primero se divide el conjunto en dos subconjuntos según tengan o no la variable considerada. Aquellas que la tienen se pasan a la

función `csplAllResolventsByPsymb`, mientras que aquellas que no la tienen se unen posteriormente al resultado que devuelve esta función.

```
csplRegularResolution : List PSymb -> List ClausePL -> ( Bool, List ClausePLSet )
csplRegularResolution vars clauses =
    let
        cs =
            (PL_CL.csplRemoveSubsumedClauses << PL_CL.csplRemoveTautClauses) clauses
    in
    let
        new_vars =
            uniqueConcatList [] (vars ++ PL_CL.csplSymbols cs)
    in
    Maybe.withDefault ( False, [] ) <| csplRegularResolutionAux [ cs ] new_vars cs


csplRegularResolutionAux :
    List ClausePLSet -> List PSymb -> ClausePLSet -> Maybe ( Bool, List ClausePLSet )
csplRegularResolutionAux hist vars clauses =
    case clauses of
        [] ->
            Just ( False, hist ++ [ clauses ] )

        [ [] ] ->
            Just ( True, hist ++ [ clauses ] )

        _ ->
            case vars of
                v :: vs ->
                    let
                        ( u1, u2 ) =
                            List.partition (List.all (\x -> Tuple.first x /= v)) clauses
                    in
                    csplRegularResolutionAux (hist ++ [ clauses ]) vs
                        <| (PL_CL.csplRemoveSubsumedClauses
                            <| PL_CL.csplRemoveTautClauses
                                <| u1 ++ csplAllResolventsByPsymb u2 v

                [] ->
                    Nothing
```

Código 4.22: Algoritmo de Resolución regular en LogicUS

Resolución como método de búsqueda

Otra aproximación en el uso de la resolución proposicional es su combinación con estrategias o algoritmos de búsqueda, de forma que se proporciona un mecanismo sólido y completo (orientado a la refutación). Una forma común para la representación de estos métodos es a través de listas, árboles (como DPLL) o digrafos acíclicos (como los que presentaremos en este apartado). Es común el uso de las listas tanto para implementar estas operaciones como para la representación de los digrafos, lo que permite una representación compacta de los conjuntos de cláusulas y además almacena información estructural sobre qué cláusulas se resolvieron para derivar cada resolvente. Nuestra implementación sigue precisamente dicha estructura.

Como hemos señalado, y como suele ser habitual en las implementaciones clásicas de procesos de búsqueda, basaremos el proceso en el manejo de dos listas, una con los nodos ya explorados (tradicionalmente denotados por *cerrados*) y otra con los nodos a la espera de ser explorados (tradicionalmente denotados por *abiertos*). El proceso de búsqueda es llevado a cabo mediante los siguientes pasos:

1. Se toma el primer elemento de la lista de *abiertos*, y que se considera ordenada de acuerdo con una heurística. En la implementación se propone una heurística combinada de Best First Search (tomando la longitud de la cláusula como medida de bondad) junto con otras estrategias de resolución:

- `csplSCFResolution`: Utiliza únicamente la heurística *Shortest Clause First (SCF)*.
- `csplSCFLinearResolution`: Utiliza la heurística *SCF* con una búsqueda lineal, esto es, tomando siempre una de las cláusulas generadas en el paso anterior como cláusula participante en la resolvente. Este caso se puede asemejar a un proceso de *Backtracking*.
- `csplSCFPositiveResolution`: Utiliza la heurística *SCF* con resolución positiva.
- `csplSCFNegativeResolution`: Utiliza la heurística *SCF* con resolución negativa.
- `csplSCFUnitaryResolution`: Utiliza la heurística *SCF* con resolución unidad. Nótese que esta estrategia no es refutacionalmente completa, en general, pero sí lo es circunscrita a un ámbito corriente de aplicación, correspondiente a las Cláusulas de Horn.
- `csplSCFByEntriesResolution`: Utiliza la heurística *SCF* con resolución por entradas. Nótese que esta estrategia no es refutacionalmente completa, en general, pero sí lo es circunscrita a un ámbito de las Cláusulas de Horn.

En ningún caso se mostrará el proceso completo de búsqueda ya que es probable que existan procesos intermedios que no hayan resultado satisfactorios y que no sean relevantes para el usuario. De hecho, si el proceso no consigue llegar a la cláusula vacía (el conjunto seria satisfactible, al menos en las estrategias refutacionalmente completas) entonces el grafo devuelto corresponde únicamente a los nodos del conjunto, dado que ninguna de las vías exploradas es satisfactoria.

2. Se calculan las resolventes de dicha cláusula con las cláusulas de cerrados y abiertos con las que no se haya hecho resolución previamente y las cláusulas derivadas son añadidas en la lista de abiertos, asignando a cada cláusula un identificador único (número mayor o igual que 1). Dos puntualizaciones:

   - En algunos casos, como `csplSCFResolution`, esa resolución se puede reducir a la resolución con únicamente los cerrados sin pérdida de completitud. En otros casos, para no realizar la resolución con cláusulas con las que ya se había realizado previamente, se mantiene un diccionario en el que, para cada uno de los nodos (cláusulas), se mantiene la lista de nodos (cláusulas) con las que ha sido resuelta previamente.
   - Nótese que si la cláusula derivada de una resolvente pertenece ya a la lista de *abiertos* o de *cerrados* entonces no es tenida en consideración. Esta restricción es, en realidad, más fuerte basada en el concepto de subsunción clausal. Si la cláusula obtenida es subsumida por otra (de *abiertos* o *cerrados*) entonces no es añadida a la lista de abiertos. En aquellos casos en los que únicamente se resuelve con cláusulas de cerrados, de la lista de abiertos también son eliminadas en cada paso las cláusulas subsumidas por otras. Sin embargo, cuando se hace resolución respecto a ambas listas, esto no es posible ya que puede que se elimine algún nodo (cláusula) que haya intervenido en el proceso de resolución anterior.

3. Si se obtiene la cláusula vacía, entonces el conjunto es insatisfactible y se recupera el proceso de resolución seguido hasta la obtención de dicha cláusula. Para ello los nodos, además de estar identificados por un número único, no están etiquetados únicamente con una cláusula sino que lo están con lo que denominamos `ResolutionItem` (*RI*), correspondiente a un *record* con las propiedades: `c` (la cláusula), `p1` (idientificador de una de las cláusulas de la resolvente), `la otra cláusula de la resolvente`, `l1` (el literal considerado en la resolvente para la primera cláusula), `l2` (el literal considerado en la resolvente para la segunda cláusula, que debe corresponder a la negación de l1). Así, el proceso de reconstrucción (llevado a cabo en la función `recoverResolutionPath`) conlleva los pasos mostrados en el *algoritmo 4.4*

4. En caso de que se haya llegado a la cláusula vacía se devuelve `True` y el grafo asociado al conjunto de nodos y aristas proporcionado por la función anterior. En caso contrario, se devuelve `False` y el grafo con nodos las cláusulas originales, sin aristas. Nótese que en el primer caso es posible que algunas de las cláusulas originales no se hayan utilizado en el proceso de resolución, los nodos referentes a dichas cláusulas son añadidos por cuestiones puramente semánticas (explicativas).

Algoritmo 4.4: Algoritmo de Resolución Proposicional

RecoverResolutionPath(U)**:**

Input : i: el identificador de un nodo ($i\ in N\backslash\{0\}$),
$items$: Un diccionario con los *RIs* generados durante el proceso de búsqueda, indexados por el número del nodo.

Output: $nodes$:una lista con los nodos del grafo de resolución (parcial),
$edges$: una lista con las aristas del grafo de resolución (parcial)

1. Si $i \notin items$: devolver $(nodes = \emptyset, edges = \emptyset)$
2. Si no, sea $\{c, p1, p2, l1, l2\}$ el RI asociado al nodo i.
3. Hacer $(nodes_1, edges_1) = RecoverResolutionPath(p1, items)$, $(nodes_2, edges_2) = RecoverResolutionPath(p2, items)$
3. Hacer n_i el nodo de id i y etiquetarlo con c. Hacer e_{i1} la arista $(p1, i)$ etiquetada con $l1$; y e_{i2} la arista $(p2, i)$ etiquetada con $l2$.
4. Devolver $(nodes = nodes_1 \cup nodes_2 \cup \{n_i\}, edges = edges_1 \cup edges_2 \cup \{e_{i1}, e_{i2}\})$

end

NOTA: Los RIs referentes a las cláusulas iniciales poseen como `p1` *y* `p2` *al nodo 0 (inexistente) y como literales asociados los literales positivo y negativo asociados al símbolo de la cadena vacía, respectivamente, (no son utilizados).*

Del *código 4.23* al *código 4.26* se muestra la implementación detallada del algoritmo de *Resolución SCF* que comentaremos en profundidad, pero que sigue el esquema ya presentado. Complementariamente, del *código 4.27* al *código 4.30* mostraremos detalladamente la implementación de la resolución lineal que es ligeramente distinta al esquema presentado en la anterior. El resto de las estrategias poseen una implementación análoga, por lo que no se expondrán explícitamente.

Implementación en LogicUS de la resolución SCF

Según el esquema general presentado, hemos de mostrar los siguientes pasos:

Esquema principal

- Si no quedan nodos por explorar ($abiertos = \emptyset$) entonces devolver una lista de nodos y aristas vacías (satisfactible). En otro caso tomar la primera cláusula.
- Si la cláusula es vacía entonces devolver los nodos y aristas proporcionados por la reconstrucción del camino de resolución (`recoverResolutionPath`).
- En otro caso, realizar la resolución con los nodos cerrados. En otras estrategias es necesario hacerlo también con las abiertas, pero en este caso basta hacerlo con los cerrados ya que en algún momento posterior las abiertas que no son subsumidas serán resueltas con las cerradas. Para ello se recurre a la función auxiliar `resolventsWithClosedSCFResolutionAux` que comentaremos posteriormente.
- Pasar la cláusula considerada a *cerrados*, identificándola con el id correspondiente (nid) y actualizar abiertos con las nuevas cláusulas generadas de la resolución con cerrados, ordenándolas por la longitud. Veremos por qué este orden es importante en la función `openedUpdationSCFResolutionAux`.
- Repetir el proceso con los nuevos cerrados y abiertos, actualizando $nid \leftarrow nid + 1$.

```
csplSCFResolutionAux :
    List ( Int, ResolutionItem ) -> List ( Int, ResolutionItem ) -> Int ->
        ( List (Node ClausePL), List (Edge ClausePLLiteral) )

csplSCFResolutionAux closed opened nid =
    case opened of
        [] ->
            ( [], [] )

        ( _, ri ) :: xs ->
            if List.isEmpty ri.c then
                let
                    refDict =
                        Dict.fromList <| closed ++ [ ( nid + 1, ri ) ]
                in
                recoverResolutionPath (nid + 1) refDict

            else
                let
                    r_closed =
                        resolventsWithClosedSCFResolutionAux closed (nid + 1) ri.c
                in
                let
                    new_closed =
                        closed ++ [ ( nid + 1, ri ) ]

                    new_opened =
                        openedUpdationSCFResolutionAux
                            xs
                            (List.sortBy (\x -> Tuple.first x)
                                <| List.map (\x -> ( List.length x.c, x )) r_closed)
                in
                csplSCFResolutionAux new_closed new_opened (nid + 1)
```

Código 4.23: Algoritmo de Resolución SCF en LogicUS (I)

```
resolventsWithClosedSCFResolutionAux :
  List ( Int, ResolutionItem ) -> Int -> ClausePL -> List ResolutionItem
resolventsWithClosedSCFResolutionAux closed id c =
  List.foldl
    (\( i, ri ) ac ->
      List.foldl
        (\( cj, l1, l2 ) ac2 ->
          if not <| PL_CL.cplIsTautology cj
            || List.any (\x -> PL_CL.cplSubsumes x.c cj) ac
            || List.any (\( _, x ) -> PL_CL.cplSubsumes x.c cj) closed
          then
           { c = cj, p1 = id, l1 = l1, p2 = i, l2 = l2 } ::
            List.filter (\x -> not <| PL_CL.cplSubsumes cj x.c) ac2

          else
            ac2
        )
        ac
        (cplAllResolvents ri.c c)
    )
    []
    closed
```

Código 4.24: Algoritmo de Resolución SCF en LogicUS (II)

Resolución con CERRADOS (`resolventsWithClosedSCFResolutionAux`)

La función opera del siguiente modo:

- Recibe el conjunto actual de cerrados, la cláusula considerada en la resolución y el id ($nid + 1$) de la misma.
- Se realiza un proceso iterativo mediante plegado sobre la lista de *cerrados*, de forma que en cada paso se considera la cláusula c_i de cerrados y se realiza:
  1. Se calculan todas las resolventes entre c y c_i.
  2. Se realiza un recorrido sobre las resolventes de forma que para cada una de ellas:
     - Si es tautología o es subsumida por alguna otra cláusula (de una resolución de una pareja anterior guardadas en el acumulador, o de cerrados) entonces es desechada.
     - En otro caso, es añadida al acumulador eliminando de él aquellas cláusulas subsumidas por la recién incorporada.

De esta forma, y tal y como se muestra en el *código 4.24*, se obtienen las nuevas resolventes que serán incorporadas a la lista de abiertos (si procede).

Actualización de ABIERTOS (`resolventsWithClosedSCFResolutionAux`)
Equivalente en la resolución positiva, negativa, por entradas y unitaria. La función opera del siguiente modo:

- Recibe la lista de *abiertos* actual y las nuevas incorporaciones. Ambas corresponden a listas de tuplas `(Int, ResolutionItem)` en el que el entero corresponde a la longitud de la cláusula del correspondiente RI. Ambas listas están ordenadas por la primera componente de la tupla (la longitud de la cláusula).
- Se realiza un proceso iterativo mediante plegado, con un doble acumulador, sobre la lista de incorporaciones. El doble acumulador se inicia con la primera componente vacía y la segunda la lista de abiertos antigua (veremos más adelante la razón). De forma que en cada paso se considera el elemento `(li, ri)` y se realiza:
  1. Se toman de la lista de antiguos abiertos por revisar (segunda componente del acumulador) los elementos cuya primera componente (la longitud) es menor o igual que la de la clausula considerada. Y se añaden al acumulador final (primera componente).
  2. Ahora,
     - Si algún elemento del acumulador final subsume a la cláusula considerada, entonces ésta es desechada y se actualiza la lista de antiguos elementos por revisar, eliminando los que han sido incorporados al acumulador final.
     - Si no, la cláusula es añadida al acumulador final y se actualiza la lista de antiguos elementos por revisar eliminando los que han sido incorporados y los que son subsumidos por la cláusula considerada (si los hubiere).
- Al final del proceso basta concatenar ambas componentes del acumulador para obtener la nueva lista de abiertos.

De esta forma se obtienen las nuevas resolventes que serán incorporadas a la lista de abiertos (si procede).

```
resolventsWithClosedSCFResolutionAux :
    List ( Int, ResolutionItem ) -> Int -> ClausePL -> List ResolutionItem
resolventsWithClosedSCFResolutionAux closed id c =
 List.foldl (\( i, ri ) ac ->
        List.foldl
            (\( cj, l1, l2 ) ac2 ->
                if not <| PL_CL.cplIsTautology cj
                    || List.any (\x -> PL_CL.cplSubsumes x.c cj) ac
                    || List.any (\( _, x ) -> PL_CL.cplSubsumes x.c cj) closed
                then
```

```
                    { c = cj, p1 = id, l1 = l1, p2 = i, l2 = l2 } ::
                        List.filter (\x -> not <| PL_CL.cplSubsumes cj x.c) ac2

                else
                    ac2
            )
            ac
            (cplAllResolvents ri.c c)
    )
    []
    closed
```

Código 4.25: Algoritmo de Resolución SCF en LogicUS (III)

Función principal (`csplSCFResolution`)

Esta función simplemente inicializa los valores de *abiertos* con el conjunto original de cláusulas, eliminando las subsumidas y las tautologías y estableciendo los valores previamente comentados para `p1`, `p2`, `l1` y `l2`, así como la longitud de la cláusula y ordenando la lista por ella; y de *cerrados* correspondiente a la lista vacía y el id del primer nodo (0). Con esto, ejecuta el algoritmo (mediante una llamada a `csplSCFResolutionAux`) y construye un grafo a partir de la lista de nodos y la lista de aristas obtenidas, añadiendo nuevos nodos para las cláusulas originales no presentes en la resolución, aunque realmente son irrelevantes.

```
csplSCFResolution : List ClausePL -> ( Bool, ResolutionTableau )
csplSCFResolution clauses =
    let
        cs =
            PL_CL.csplRemoveEqClauses clauses
    in
    let
        new_cs =
            List.sortBy (\x -> Tuple.first x)
                <| List.map
                    (\x ->
                        (
                            List.length x,
                            {c=x, p1=0, p2 =0, l1=( "", True ), l2=( "", False )}
                        )
                    )
                    (PL_CL.csplRemoveSubsumedClauses <| PL_CL.csplRemoveTautClauses cs)
    in
    let
        ( nodes, edges ) =
            csplSCFResolutionAux [] new_cs 0
    in
    let
        nid_max =
            Maybe.withDefault 0 <| List.maximum <| List.map (\x -> x.id) <| nodes

        nodes_clauses =
            List.map (\x -> x.label) <| nodes
    in
    let
        final_nodes =
            List.map (\x -> Node x.id ( List.member x.label cs, x.label )) nodes ++
                (List.indexedMap
                    (\i x -> Node (nid_max + i + 1) ( True, x ))
                    (List.filter (\x -> not (List.member x nodes_clauses)) cs)
                )

    in
    ( edges /= [], Graph.fromNodesAndEdges final_nodes edges )
```

Código 4.26: Algoritmo de Resolución SCF en LogicUS (IV)

Implementación en LogicUS de la resolución SCF-Lineal

Esquema principal

- Si no quedan nodos por explorar ($abiertos = \emptyset$), entonces devolver una lista de nodos y aristas vacías (satisfactible). En otro caso, tomar la primera cláusula.
- Si la cláusula es vacía, devolver los nodos y aristas proporcionados por la reconstrucción del camino de resolución (`recoverResolutionPath`).
- En otro caso, realizar la resolución con los nodos cerrados y con los nodos abiertos. Para ello, se realiza primero con los abiertos recurriendo a la función `resolventsWithOpenedSCFLinearResolutionAux`, se eliminan las subsumidas por cláusulas de *cerrados* y se añaden a las obtenidas de la resolución con *cerrados* a través de la función `resolventsWithClosedSCFLinearResolutionAux`. Finalmente, se eliminan las cláusulas que son subsumidas por otras del conjunto de resolventes y se ordenan por su longitud.
- Pasar la cláusula considerada a *cerrados*, identificándola con el id correspondiente, actualizar el diccionario de cláusulas de cerrados resueltas, y para cada resolvente hacer una llamada recursiva, de forma que unas ramas no pueden tener relación con resolventes de ese mismo paso consiguiendo que la resolución sea lineal. Además nótese que las instrucciones `filter` y `map` hacen que el cálculo sea paralelizable, así como la evaluación parcial y perezosa permiten que en cuanto alguno cumpla los requisitos pare la ejecución y devuelva dicho elemento.

```
csplSCFLinearResolutionAux :
  List ( Int, ResolutionItem ) -> Dict Int (List Int) -> List ( Int, ResolutionItem )
    -> Int -> ( List (Node ClausePL), List (Edge ClausePLLiteral) )
csplSCFLinearResolutionAux closed resDone opened nid =
  case opened of
    [] ->
      ( [], [] )

    ( id, rid ) :: xs ->
      if List.isEmpty rid.c then
        let
          refDict =
            Dict.fromList <| closed ++ opened
        in
        recoverResolutionPath id refDict

      else
        let
          resolvents_i =
            (filterSubsumedResolutionItems<<List.sortBy (\x ->List.length x.c))
              (resolventsWithClosedSCFLinearResolutionAux closed resDone id rid.c)
              ++ (List.filter
                (\ri -> not <|
                  List.any
                    (\( _, x ) -> PL_CL.cplSubsumes x.c ri.c) closed)
                    (resolventsWithOpenedSCFLinearResolutionAux xs id rid.c)
              )
        in
        let
          newClosed =
            closed ++ [ ( id, rid ) ]

            newResDone =
              Dict.insert id (List.map Tuple.first <| closed ++ xs) <|
                Dict.map (\_ v -> v ++ [ id ]) resDone
        in
          Maybe.withDefault ( [], [] ) << List.head) <|
            List.filter (not << List.isEmpty << Tuple.first) <|
              List.map (\ri -> csplSCFLinearResolutionAux newClosed newResDone
                (( nid + 1, ri ) :: xs) (nid + 1)) resolvents_i
```

Código 4.27: Algoritmo de Resolución Lineal SCF en LogicUS (I)

Resolución con CERRADOS y ABIERTOS (`resolventsWithClosedSCFLinearResolutionAux`)

La implementación es completamente análoga a la anteriormente presentada para el caso SCF (no lineal) sin más que añadir un filtro sobre los cerrados que evite la resolución con aquellas cláusulas con las que ya se haya resuelto previamente. Este filtro es implementado suprimiendo las cláusulas cuyas listas de índices del diccionario de resueltos previamente contienen el índice de la cláusula considerada en la resolución.

```
(List.filter (\ ( i, _ ) -> not <| List.member id <| Maybe.withDefault [] <|
Dict.get i resDone) <| closed)
```

La implementación para el caso de abiertos es también completamente análoga a la anteriormente presentada para *cerrados* (sin el filtro), pero considerando ahora la lista de abiertos en lugar de la lista de cerrados.

En otras resoluciones se pueden establecer filtros adicionales (tanto en abiertos como en cerrados) como que una de las dos cláusula sea positiva, negativa, unitaria o corresponda a las entradas, pero las implementaciones son esencialmente iguales a la presentada. De hecho, los filtros corresponderían a añadir mediante la conjunción `&&` el predicado de resolución previamente mostrado (`not <| List.member id <| Maybe.withDefault [] <| Dict.get i resDone`):

- `csplSCFPositiveResolution`: `(cplIsPositive ci || cplIsPositive ci_c)` (donde `Ci` corresponde a la cláusula considerada y `ci_c` a cada una de las cláusulas del conjunto (*cerrados* o *abiertos*).
- `csplSCFNegativeResolution`: `(cplIsNegative ci || cplIsNegative ci_c)` (donde `Ci` corresponde a la cláusula considerada y `ci_c` a cada una de las cláusulas del conjunto (*cerrados* o *abiertos*).
- `csplSCFUnitaryResolution`: `(li == 1 || li_c == 1)` (donde `li` corresponde a la longitud de la cláusula considerada y `li_c` a la longitud de cada una de las cláusulas del conjunto (*cerrados* o *abiertos*).
- `csplSCFByEntriesResolution`: `(ri.p1 == 0 || ri_c.p1 == 0)` (donde `ri` corresponde al RI de la cláusula considerada y `ri_c` a cada una de las cláusulas del conjunto (*cerrados* o *abiertos*).

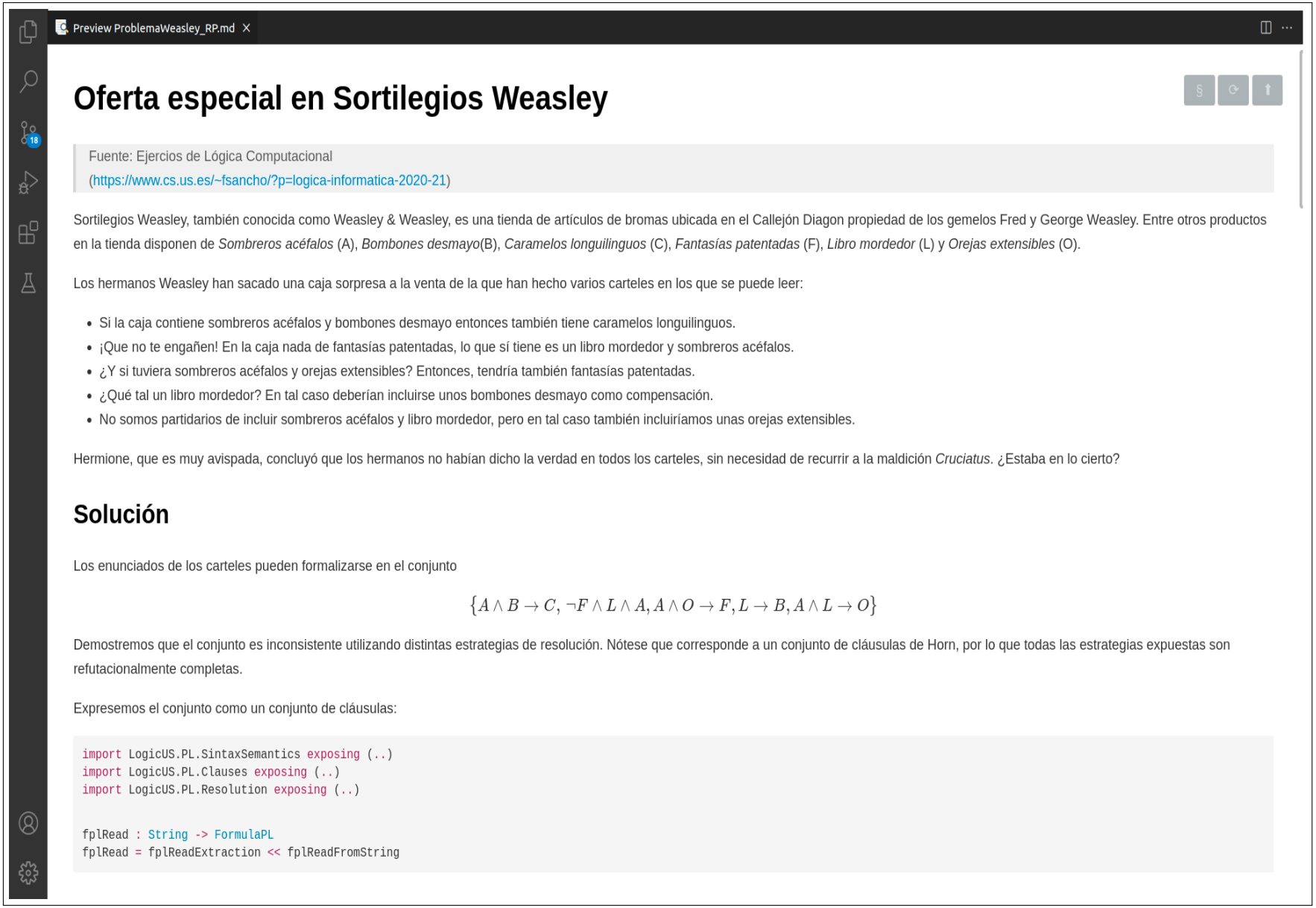
Función principal (`csplSCFLinearResolution`)

Es similar a la presentada en `csplSCFResolution`, incorporando la inicialización del diccionario (vacío) y de los ids de los nodos iniciales y también a las implementadas en el resto de soluciones, sin más que ajustar la llamada a la función auxiliar correspondiente.

A diferencia del anterior, el conjunto inicial de nodos (sobre el que se aplican también la simplificación por subsunción y tautología) asigna a cada cláusula inicial según el orden por la longitud de las cláusulas los primeros números naturales (a partir del 1) haciendo uso de la función `indexedMap` que toma una lista y le aplica una función que puede depender tanto del valor del elemento como de la posición.

Como en casos anteriores, se han implementado también las funciones de representación correspondientes, tanto en formato de cadena como en formato DOT.

Presentamos a continuación un ejemplo de uso que permite ilustrar los distintos mecanismos de resolución expuestos:


Preview ProblemaWeasley_RP.md
Oferta especial en Sortilegios Weasley
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Sortilegios Weasley, también conocida como Weasley & Weasley, es una tienda de artículos de bromas ubicada en el Callejón Diagon propiedad de los gemelos Fred y George Weasley. Entre otros productos en la tienda disponen de Sombreros acéfalos (A), Bombones desmayo(B), Caramelos longuilinguos (C), Fantasías patentadas (F), Libro mordedor (L) y Orejas extensibles (O).
Los hermanos Weasley han sacado una caja sorpresa a la venta de la que han hecho varios carteles en los que se puede leer:
Si la caja contiene sombreros acéfalos y bombones desmayo entonces también tiene caramelos longuilinguos.
¡Que no te engañen! En la caja nada de fantasías patentadas, lo que sí tiene es un libro mordedor y sombreros acéfalos.
¿Y si tuviera sombreros acéfalos y orejas extensibles? Entonces, tendría también fantasías patentadas.
¿Qué tal un libro mordedor? En tal caso deberían incluirse unos bombones desmayo como compensación.
No somos partidarios de incluir sombreros acéfalos y libro mordedor, pero en tal caso también incluiríamos unas orejas extensibles.
Hermione, que es muy avispada, concluyó que los hermanos no habían dicho la verdad en todos los carteles, sin necesidad de recurrir a la maldición Cruciatus. ¿Estaba en lo cierto?
Solución
Los enunciados de los carteles pueden formalizarse en el conjunto
$$\{A \wedge B \to C, \neg F \wedge L \wedge A, A \wedge O \to F, L \to B, A \wedge L \to O\}$$
Demostremos que el conjunto es inconsistente utilizando distintas estrategias de resolución. Nótese que corresponde a un conjunto de cláusulas de Horn, por lo que todas las estrategias expuestas son refutacionalmente completas.
Expresemos el conjunto como un conjunto de cláusulas:
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.PL.Clauses exposing (..)
import LogicUS.PL.Resolution exposing (..)
fplRead : String -> FormulaPL
fplRead = fplReadExtraction << fplReadFromString

Preview ProblemaWeasley_RP.md

Definimos las fórmulas:

```
f1 : FormulaPL
f1 = fplRead "a & b -> c"

f2 : FormulaPL
f2 = fplRead "¬f & l & a"

f3 : FormulaPL
f3 = fplRead "a & o -> f"

f4 : FormulaPL
f4 = fplRead "l -> b"

f5 : FormulaPL
f5 = fplRead "a & l -> o"
```

Entonces, el conjunto de cláusulas correspondería a:

```
cs : ClausePLSet
cs = splToClauses [f1,f2,f3, f4, f5]
```

$$\{\{c, \neg a, \neg b\}, \{\neg f\}, \{l\}, \{a\}, \{f, \neg a, \neg o\}, \{b, \neg l\}, \{o, \neg a, \neg l\}\}$$

Resolución regular

```
cs_reg : (Bool, List (List ClausePL))
cs_reg = csplRegularResolution (csplSymbols cs) cs
```

True

De forma que el proceso correspondería a :

$$\{\{c, \neg a, \neg b\}, \{\neg f\}, \{l\}, \{a\}, \{f, \neg a, \neg o\}, \{b, \neg l\}, \{o, \neg a, \neg l\}\}$$
$$\{\{\neg f\}, \{l\}, \{b, \neg l\}, \{c, \neg b\}, \{f, \neg o\}, \{o, \neg l\}\}$$
$$\{\{\neg f\}, \{l\}, \{f, \neg o\}, \{o, \neg l\}, \{c, \neg l\}\}$$
$$\{\{\neg f\}, \{l\}, \{f, \neg o\}, \{o, \neg l\}\}$$
$$\{\{l\}, \{o, \neg l\}, \{\neg o\}\}$$
$$\{\{\neg o\}, \{o\}\}$$
$$\{\square\}$$

Preview ProblemaWeasley_RP.md
Resolución lineal SCF
cs_lin : (Bool, ResolutionTableau)
cs_lin = csplSCFLinearResolution cs
True
De forma que el proceso correspondería a :
{l}
{o,¬ a,¬ l}
{¬ f}
{f,¬ a,¬ o}
{a}
{c,¬ a,¬ b}
{b,¬ l}
¬ f
f
{¬ a,¬ o}
a
o
¬ a
{¬ o}
a
l
¬ o
{¬ a,¬ l}
¬ a
{¬ l}
¬ l
□

Figura 4.8: Ejemplo de uso de LogicUS.PL.Resolution
Fuente propia. Creada con litvis

4.2.10. LogicUS.PL.CSP

En este punto presentaremos la implementación realizada para la definición y resolución de CSP SAT, a través de la definición de fórmulas parametrizadas haciendo uso de los operadores de conjunción y disyunción paramétricos.

Las fórmulas parametrizadas permiten definir fórmulas mucho más grandes en las que los símbolos incluyen parámetros como índices, que son sustituidos según los valores definidos en los cuantificadores. Estas fórmulas son especialmente interesantes cuando queremos trabajar con conjuntos de fórmulas asociados a problemas reales como los Problemas de Satisfacción de Restricciones (*Constraint Satisfaction Problems - CSP*). Así, un ejemplo clásico, y que trataremos después con LogicUS es el problema de las 8 Reinas, que consiste en determinar una colocación de 8 reinas en un tablero de ajedrez 8×8 de forma que ninguna de ellas sea atacada por otra. De forma que, por ejemplo, la restricción de ausencia de dos reinas en una misma diagonal principal puede modelarse como:

$$\bigwedge_{i\in\{1\ldots8\}} \bigwedge_{j\in\{1\ldots8\}} \left(P_{i,j} \rightarrow \bigwedge_{\substack{k \in \{-7\ldots7\} \\ s.t.\ \psi_1}} \neg P_{i+k,j+k} \right)$$

$$\psi_1 \equiv (k \neq 0) \wedge (1 \leq i+k \leq 8) \wedge (1 \leq j+k \leq 8)$$

En dichas fórmulas se identifican algunas cuestiones interesantes que hemos de tener en cuenta al definir la estructura de las fórmulas:

- Cada operador paramétrico define un parámetro (o varios considerándolo como varios operadores del mismo tipo sobre cada una de las variables) que tiene asociado un universo de discurso (rango de variación del parámetro) y un conjunto de restricciones sobre el valor que puede tomar el parámetro, esto es una expresión booleana.

- Tanto en los índices como en las condiciones pueden participar expresiones aritméticas sobre los parámetros.

Teniendo esto en cuenta se definen las fórmulas paramétricas (`BigFPL`) con la siguiente estructura: ´

```
type alias Param =
  { name : String
    , values : List Int
  }

type BigFPL
  = Atom String (List A_Expr)
  | Neg BigFPL
  | Conj BigFPL BigFPL
  | Disj BigFPL BigFPL
  | Impl BigFPL BigFPL
  | Equi BigFPL BigFPL
  | BAnd (List Param) B_Expr BigFPL
  | BOr (List Param) B_Expr BigFPL
  | Insat
  | Taut
```

Código 4.28: Estructura de las Fórmulas parametrizadas en LogicUS

Aunque podríamos definir las fórmulas a través de los constructores, si la tarea ya podía ser ardua con la fórmulas proposicionales clásicas, con las fórmulas paramétricas esto se hace inviable. Por ello, se ha implementado un Parser que permita definir las fórmulas directamente a través de cadenas siguiendo el formato:

- Las fórmulas atómicas (variables proposicionales) corresponden a variables proposicionales, compuestas por cadenas de texto, con caracteres en mayúscula, y opcionalmente pueden ser sub-indexadas por una serie de índices que corresponden a expresiones aritméticas. Dichos índices se especifican entre los símbolos `_{` y `}`; y separados por comas. Ejemplos de fórmulas atómicas son: `P`, `Q_{i, j}`, `AL_{i + k, i-k}`.

- Una expresión aritmética puede corresponder a un número entero, a una variable especificada como una cadena de caracteres en minúscula seguida, opcionalmente, por algunos dígitos numéricos (`x`, `i1`, `y33`), o la combinación binaria de dos expresiones aritméticas a través de los operadores infijos: `+` (suma), `-` (resta), `*` (producto), `//` (división entera), `%` (módulo).

- Las fórmulas pueden definirse como fórmulas atómicas o como asociaciones de ellas mediante conectivos binarios infijos: `&` (conjunción), `|` (disyunción), `->` (implicación), `<->` (equivalencia); el conectivo unitario: `¬` (negación); o dos conectivos grandes que siguen el formato:

  BigOp PARAMETERS CONDITION BigFPL

  - El BigOp puede corresponder a `!&` (BigAnd) o `!|` (BigOr).

  - La lista de parámetros que establece asociaciones entre el identificador del parámetro y su universo de variación en la forma identificador universo o identificador (universo) (según los casos descritos debajo). Estas asociaciones están encerradas entre corchetes `[`, `]` y separadas por comas. Tenga en cuenta que, por ejemplo, `!&[i(1:8), j(1:8)]{T}(...)` sería equivalente a `!&[i(1:8)]{T} !&[j(1:8)]{T} (...)`

  - El universo de variación de un parámetro se puede especificar mediante un conjunto de valores enteros, expresados entre llaves, p.e `i{-1,1,3,5}` o como un rango expresado como `(ll:up)` con `ll` el límite inferior y `up` el superior, p.e. `j(1:8)`.

  - La condición se establece a partir de una expresión booleana expresada entre llaves y que puede corresponder a: la condición verdadera `T`, la expresión falsa `F`, una comparación entre expresiones aritméticas. Estas expresiones comparativas se expresan entre corchetes `[e1` *comp* `e2 ]` e incluyen: `=` (igual), `!=` (distinto), `>=` (mayor o igual), `<=` (menor o igual), `>` (mayor estricto), `<` (menor estricto); o con expresiones booleanas creadas a partir de operadores booleanos `AND` (conjunción), `OR` (disyunción), `NOT` (negación), además del uso de paréntesis para establecer prioridades.

Además, es recomendable tratar de extraer los operadores paramétricos a la cabeza de la fórmula, no para la facilidad de lectura del parser, sino para la comodidad en la escritura de la fórmula. Veámoslo con el ejemplo de las 8 Reinas como:

```
!& [i(1:8)]{T}
    !&[j(1:8)]{T}
        (P_{i,j} ->
            !& [k(-7:7)]{[k!=0]AND[i+k>=1]AND[j+k>=1]AND[i+k<=8]AND[j+k<=8]}
                (¬P_{i+k,j+k})
        )
```

Extrayendo los operadores paramétricos y agrupándolos en un mismo operador (no siempre se puede) quedaría como:

```
!&[i(1:8),j(1:8),k(-7:7)]
    {[k!=0]AND[i+k>=1]AND[j+k>=1]AND[i+k<=8]AND[j+k<=8]}
        (P_{i,j} -> ¬P_{i+k,j+k})
```

Téngase en cuenta, además, que para que las fórmulas estén bien formadas, han de cumplir dos requisitos básicos:

- Un mismo parámetro no puede estar definido múltiples veces, ni en el mismo ni en distintos operadores.
- Los parámetros que intervengan en alguna expresión han de estar previamente definidos en el mismo operador o en uno predecesor (más externo).

Con esto ya podríamos definir fórmulas parametrizadas a partir de cadenas. De hecho, el módulo expone únicamente este mecanismo de definición, con el fin de evitar que se puedan definir fórmulas incorrectas a través de los propios constructores.

Además de los mecanismos de definición el módulo provee dos mecanismos fundamentales de operación: la expansión de fórmulas, que permite convertir las fórmula paramétricas en fórmulas proposicionales 'puras', y la resolución de satisfactibilidad de conjuntos de fórmulas paramétricas y cláusulas (*SAT Solver*), que permite establecer la satisfactibilidad y obtener un modelo (si existe), de ese conjunto de cláusulas.

Veámos primero el mecanismo de expansión de fórmulas. El mismo, se basa en sustituir los operadores paramétricos por la conjunción o disyunción (según el operador) de las fórmulas que resultan de sustituir los valores posibles de los parámetros. Para ello, se utiliza un mecanismo de sustitución y evaluación de expresiones (tanto aritméticas para el cálculo de índices, como booleanas para el filtrado de valores). En caso de que la fórmula no esté bien formada el método de expansión devolverá la fórmula insatisfactible.

En concreto, el método de expansión está definido como un método recursivo con una memoria (diccionario) que almacena el estado, asignación de los valores a los parámetros, en cada instante y del que se toman dichos valores a la hora de evaluar las expresiones. Ahora bien, cuando se encuentra con un operador paramétrico realiza todas las posibles sustituciones, creando todos los posibles estados (a través del producto cartesiano de los posibles valores) y eliminando (mediante filtrado) aquellos que no sean válidos (no cumplen la condición del operador). La implementación precisa de dicho método se encuentra detallada en el *código 4.29*, en el que se puede observar el flujo de operación descrito.

```
bfplToFPL : BigFPL -> FormulaPL
bfplToFPL f =
  Maybe.withDefault PL_SS.Insat <| bfplExpansionAux Dict.empty <| f

bfplExpansionAux : Dict String Int -> BigFPL -> Maybe FormulaPL
bfplExpansionAux var_val f =
  case f of
        Atom p is ->
            Maybe.map (PL_SS.Atom << Tuple.pair p)
              <| ME.combine <|
                List.map (\i -> Aux_AE.evaluateAExpr i var_val) is

    Neg p ->
      Maybe.map PL_SS.Neg (bfplExpansionAux var_val p)

    Conj p q ->
      Maybe.map2 PL_SS.Conj (bfplExpansionAux var_val p) (bfplExpansionAux var_val q)

    Disj p q ->
      Maybe.map2 PL_SS.Disj (bfplExpansionAux var_val p) (bfplExpansionAux var_val q)

    Impl p q ->
      Maybe.map2 PL_SS.Impl (bfplExpansionAux var_val p) (bfplExpansionAux var_val q)

    Equi p q ->
      Maybe.map2 PL_SS.Equi (bfplExpansionAux var_val p) (bfplExpansionAux var_val q)

    BAnd li c p ->
      let
        names = List.map .name li
```

```
        values = List.map .values li
      in
      let
        valid_subs =
          List.filter (\s -> Maybe.withDefault False <| Aux_BE.evaluateBExpr c s)
            <| List.map (\x -> Dict.union var_val <| Dict.fromList <| LE.zip names x)
                <| LE.cartesianProduct
                  <| values
      in
      Maybe.map PL_SS.splConjunction
        <| ME.combine
          <| List.map (\s -> bfplExpansionAux s p) valid_subs

    BOr li c p ->
      let
        names = List.map .name li

        values =
          List.map .values li
      in
      let
        valid_subs =
          List.filter (\s -> Maybe.withDefault False <| Aux_BE.evaluateBExpr c s)
            <| List.map (\x -> Dict.union var_val <| Dict.fromList <| LE.zip names x)
              <| LE.cartesianProduct
                <| values
      in
      Maybe.map PL_SS.splDisjunction
        <| ME.combine
          <| List.map (\s -> bfplExpansionAux s p) valid_subs

    Insat ->
    Just PL_SS.Insat

    Taut ->
    Just PL_SS.Taut
```

Código 4.29: Expansión de fórmulas parametrizadas en LogicUS

Tratemos ahora el algoritmo de resolución para fórmulas paramétricas y en general cláusulas. En realidad, el algoritmo implementado trabaja únicamente con cláusulas y corresponde a un algoritmo de Bcaktracking. En el algoritmo BT original habría que establecer un orden para los valores pero en este caso son sólo dos valores (verdadero, o falso) para cada una de las variables, y dicho cálculo se realiza en paralelo. En efecto, el algoritmo corresponde a DPLL pero en cada paso si hay cláusulas unitarias se toman todas las variables (en vez de sólo 1) y para elegir la variable (en caso de que no haya cláusulas unitarias) se toma aquella que participa en más cláusulas (ya sea positiva o negativamente), para ello se mantiene una memoria actualizada con el número de cláusulas en las que participa cada variable proposicional (tanto positiva como negativamente). Dicho algoritmo se encuentra implementado en el *código 4.30* y el *código 4.31*.

Como en otros módulos también se proveen los mecanismos de representación para las fórmulas parametrizadas, tanto en formato cadena como en formato Latex. Para este último se proveen dos métodos, el primero incluye en la fórmula todos los elementos (también las condiciones), y el segundo genera dos cadenas, una con la representación de la fórmula y otra con la representación de los predicados.

Como final de la descripción del módulo proporcionamos un ejemplo de uso del mismo (en la *figura 4.9*), tratando, ya que lo hemos comentado previamente, el Problema de Satisfacción de Restricciones asociado al problema de las 8 Reinas.

```
sbfplsolver : List BigFPL -> ( Bool, List PSymb )
sbfplsolver fs =
  let
    cls =
      PL_CL.csplRemoveSubsumedClauses
        <| List.concat
          <| List.map (fplToClauses << bfplToFPL) fs
  in
  case solverAux [] (varsClauses cls) cls of
    Nothing -> ( False, [] )
    Just y -> ( True, y )

solver : List ClausePL -> ( Bool, List PSymb )
solver cls =
  case solverAux [] (varsClauses cls) cls of
    Nothing -> ( False, [] )
    Just y -> ( True, y )

solverAux : List PSymb -> Dict PSymb ( Int, Int ) -> List ClausePL -> Maybe (List PSymb)
solverAux asig av_vars cls =
  if List.member [] cls then
    Nothing

  else
    case getVarsUnitaryClauses cls of
      x :: xs ->
        let
          new_asig =
            asig ++ (List.map Tuple.first <| List.filter Tuple.second (x :: xs))

          ( new_av_vars, new_cls ) =
            updationAvailable (x :: xs) av_vars cls
        in
          solverAux new_asig new_av_vars new_cls

      [] ->
        let
          best_var =
          LE.maximumBy (\( _, ( pc, nc ) ) -> pc + nc) <|
            List.filter (\( _, ( pc, nc ) ) -> pc + nc > 0) <|
              Dict.toList av_vars
        in
        case best_var of
          Just ( bv, ( pc, nc ) ) ->
            let
              ( new_av_vars1, new_cls1 ) =
                updationAvailable [ ( bv, True ) ] av_vars cls

              ( new_av_vars2, new_cls2 ) =
                updationAvailable [ ( bv, False ) ] av_vars cls
            in
            if nc > pc then
              ME.or
                (solverAux asig new_av_vars2 new_cls2)
                (solverAux new_asig1 new_av_vars1 new_cls1)

            else
              ME.or
                (solverAux new_asig1 new_av_vars1 new_cls1)
                (solverAux asig new_av_vars2 new_cls2)

          Nothing ->
            Just asig
```

Código 4.30: SAT Solver en LogicUS I

```
updationAvailable :
  List ( PSymb, Bool ) -> Dict PSymb ( Int, Int ) -> List ClausePL ->
    ( Dict PSymb ( Int, Int ), List ClausePL )
updationAvailable used_vars cur_av_vars cur_cls =
  List.foldl
    (\c ( ac1, ac2 ) ->
      if List.any (\l -> List.member l used_vars) c then
        ( Dict.map
          (\a ( pc, nc ) ->
            if List.member ( a, True ) c then
              ( pc - 1, nc )

            else if List.member ( a, False ) c then
              ( pc, nc - 1 )

            else
              ( pc, nc )
          )
          ac1
        , ac2)

      else
        ( ac1,
        ac2 ++ [
          List.filter
            (\ ( a, s ) -> not <| List.member ( a, not s ) used_vars)
            c
          ]
        )
    )
    ( List.foldl
      (\ ( a, _ ) ac -> Dict.remove a ac)
      cur_av_vars used_vars
    , []
    )
    cur_cls

evaluateAExpr : A_Expr -> Dict String Int -> Maybe Int
evaluateAExpr expr vals =
  case expr of
    Number i -> Just i

    Var s -> Dict.get s vals

    Add e1 e2 -> Maybe.map2 (+) (evaluateAExpr e1 vals) (evaluateAExpr e2 vals)

    Dif e1 e2 -> Maybe.map2 (-) (evaluateAExpr e1 vals) (evaluateAExpr e2 vals)

    Mul e1 e2 -> Maybe.map2 (*) (evaluateAExpr e1 vals) (evaluateAExpr e2 vals)

    Div e1 e2 -> Maybe.map2 (//) (evaluateAExpr e1 vals) (evaluateAExpr e2 vals)

    Mod e1 e2 -> Maybe.map2 modBy (evaluateAExpr e2 vals) (evaluateAExpr e1 vals)

evaluateBExpr : B_Expr -> Dict String Int -> Maybe Bool
evaluateBExpr expr vals =
  case expr of
    T -> Just True

    F -> Just False

    And e1 e2 -> Maybe.map2 (&&) (evaluateBExpr e1 vals) (evaluateBExpr e2 vals)

    Or e1 e2 -> Maybe.map2 (||) (evaluateBExpr e1 vals) (evaluateBExpr e2 vals)

    Not e -> Maybe.map not (evaluateBExpr e vals)

    Cond c -> evalCond c vals
```

Código 4.31: SAT Solver en LogicUS II


Preview PL_NREINAS_CSP.md
Problema de las 8 Reinas
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
El Problema de las N reinas es un problema clásico dentro de los Problemas de Satisfacción de Restricciones. El problema fue originalmente propuesto en 1848 por el ajedrecista Max Bezzel para el tablero de ajedrez y considerando 8 Reinas. Durante años, muchos matemáticos, incluyendo a Gauss y Cantor, han trabajado en él y lo han generalizado a N-reinas. El problema se enuncia como:
Hallar, si existe, una distribución de N reinas en un tablero de ajedrez $N \times N$ de manera no haya dos reinas que se ataquen entre sí.
En este ejercicio trabajaremos el problema original considerando un tablero de ajedrez clásico 8×8 y 8 reinas a colocar en el mismo. Se pide formalizar el problema y resolviéndolo haciendo uso de fórmulas proposicionales (parametrizadas).
Solución
Cualquiera que conozca el ajedrez sabe que dos reinas se atacan entre sí si ocupan la misma fila, la misma columna o la misma diagonal. Por tanto para que el problema tenga solución se debe cumplir que para cada pareja de reinas se tiene que ambas ocupan filas, columnas y diagonales (principal y secundaria) distintas. Formalicémos el problema haciendo uso de fórmulas parametrizadas y resolvámoslo haciendo uso de LogicUS.
Bien, consideremos el tablero
p81 p82 p83 p84 p85 p86 p87 p88
p71 p72 p73 p74 p75 p76 p77 p78
p61 p62 p63 p64 p65 p66 p67 p68
p51 p52 p53 p54 p55 p56 p57 p58
p41 p42 p43 p44 p45 p46 p47 p48
p31 p32 p33 p34 p35 p36 p37 p38
p21 p22 p23 p24 p25 p26 p27 p28
p11 p12 p13 p14 p15 p16 p17 p18

Preview PL_NREINAS_CSP.md

```
import LogicUS.PL.CSP exposing (..)
import LogicUS.PL.SintaxSemantics exposing (..)

bfplRead : String -> BigFPL
bfplRead = bfplReadExtraction << bfplReadFromString
```

Bien, ténganse las variables proposicionales $P_{i,j}$ con $i, j \in \{1, \ldots, 8\}$ tal que $P_{i,j}$ vale 1 si una reina está colocada en la posición $p_{i,j}$.

Entonces las restricciones pueden modelarse como:

- Dos reinas no ocupan una misma fila. Esto es en toda fila se cumple que si una casilla de la fila tiene una reina entonces las demás no pueden tener reina:

```
f1 : BigFPL
f1 = bfplRead "!&[i(1:8)]{T} (!&[j(1:8)]{T} (P_{i,j} -> !&[k(1:8)]{[k!=j]}(¬P_{i,k})))"
```

$$\bigwedge_{i \in \{1..8\}} \bigwedge_{j \in \{1..8\}} \left(P_{i,j} \rightarrow \bigwedge_{\substack{k \in \{1..8\} \\ [k \neq j]}} \neg P_{i,k} \right)$$

- Dos reinas no ocupan una misma columna. Esto es en toda columna se cumple que si una casilla de la columna tiene una reina entonces las demás no pueden tener reina:

```
f2 : BigFPL
f2 = bfplRead "!&[j(1:8)]{T} (!&[i(1:8)]{T} (P_{i,j} -> !&[k(1:8)]{[k!=i]}(¬P_{k,j})))"
```

$$\bigwedge_{j \in \{1..8\}} \bigwedge_{i \in \{1..8\}} \left(P_{i,j} \rightarrow \bigwedge_{\substack{k \in \{1..8\} \\ [k \neq i]}} \neg P_{k,j} \right)$$

- Dos reinas no ocupan una misma diagonal principal. Esto es en toda casilla se cumple que si esa casilla contiene una reina entonces las demás casillas de esa diagonal principal no pueden tener una reina:

```
f3 : BigFPL
f3 = bfplRead "!&[i(1:8)]{T} (!&[j(1:8)]{T} (P_{i,j} -> !&[k(-7:7)]{[k!=0]AND[i+k>0]AND[i+k<=8]AND[j+k>0]AND[j+k<=8]}(¬P_{i+k,j+k})))"
```

- Dos reinas no ocupan una misma diagonal principal. Esto es en toda casilla se cumple que si esa casilla contiene una reina entonces las demás casillas de esa diagonal principal no pueden tener una reina:

```
f3 : BigFPL
f3 = bfplRead "!&[i(1:8)]{T} (!&[j(1:8)]{T} (P_{i,j} -> !&[k(-7:7)]{[k!=0]AND[i+k>0]AND[i+k<=8]AND[j+k>0]AND[j+k<=8]}(¬P_{i+k,j+k})))"
```

$$\bigwedge_{i \in \{1..8\}} \bigwedge_{j \in \{1..8\}} \left(P_{i,j} \rightarrow \bigwedge_{\substack{k \in \{-7..7\} \\ s.t.\, \theta_1}} \neg P_{(i+k),(j+k)} \right)$$

$$\theta_1 \equiv \Big(\big[k \neq 0\big] \wedge \Big(\big[(i+k) > 0\big] \wedge \Big(\big[(i+k) \leq 8\big] \wedge \Big(\big[(j+k) > 0\big] \wedge \big[(j+k) \leq 8\big] \Big) \Big) \Big) \Big)$$

- Dos reinas no ocupan una misma diagonal principal. Esto es en toda casilla se cumple que si esa casilla contiene una reina entonces las demás casillas de esa diagonal principal no pueden tener una reina:

```
f4 : BigFPL
f4 = bfplRead "!&[i(1:8)]{T} (!&[j(1:8)]{T} (P_{i,j} -> !&[k(-7:7)]{[k!=0]AND[i+k>0]AND[i+k<=8]AND[j-k>0]AND[j-k<=8]}(¬P_{i+k,j-k})))"
```

$$\bigwedge_{i \in \{1..8\}} \bigwedge_{j \in \{1..8\}} \left(P_{i,j} \rightarrow \bigwedge_{\substack{k \in \{-7..7\} \\ s.t.\, \theta_1}} \neg P_{(i+k),(j-k)} \right)$$

$$\theta_1 \equiv \Big(\big[k \neq 0\big] \wedge \Big(\big[(i+k) > 0\big] \wedge \Big(\big[(i+k) \leq 8\big] \wedge \Big(\big[(j-k) > 0\big] \wedge \big[(j-k) \leq 8\big] \Big) \Big) \Big) \Big)$$

- Han de colocarse 8 reinas. Esto es equivalente a decir que en cada fila ha de haber una reina.

```
f5 : BigFPL
f5 = bfplRead "!&[i(1:8)]{T} (!|[j(1:8)]{T} (P_{i,j}))"
```

$$\bigwedge_{i \in \{1..8\}} \bigvee_{j \in \{1..8\}} P_{i,j}$$

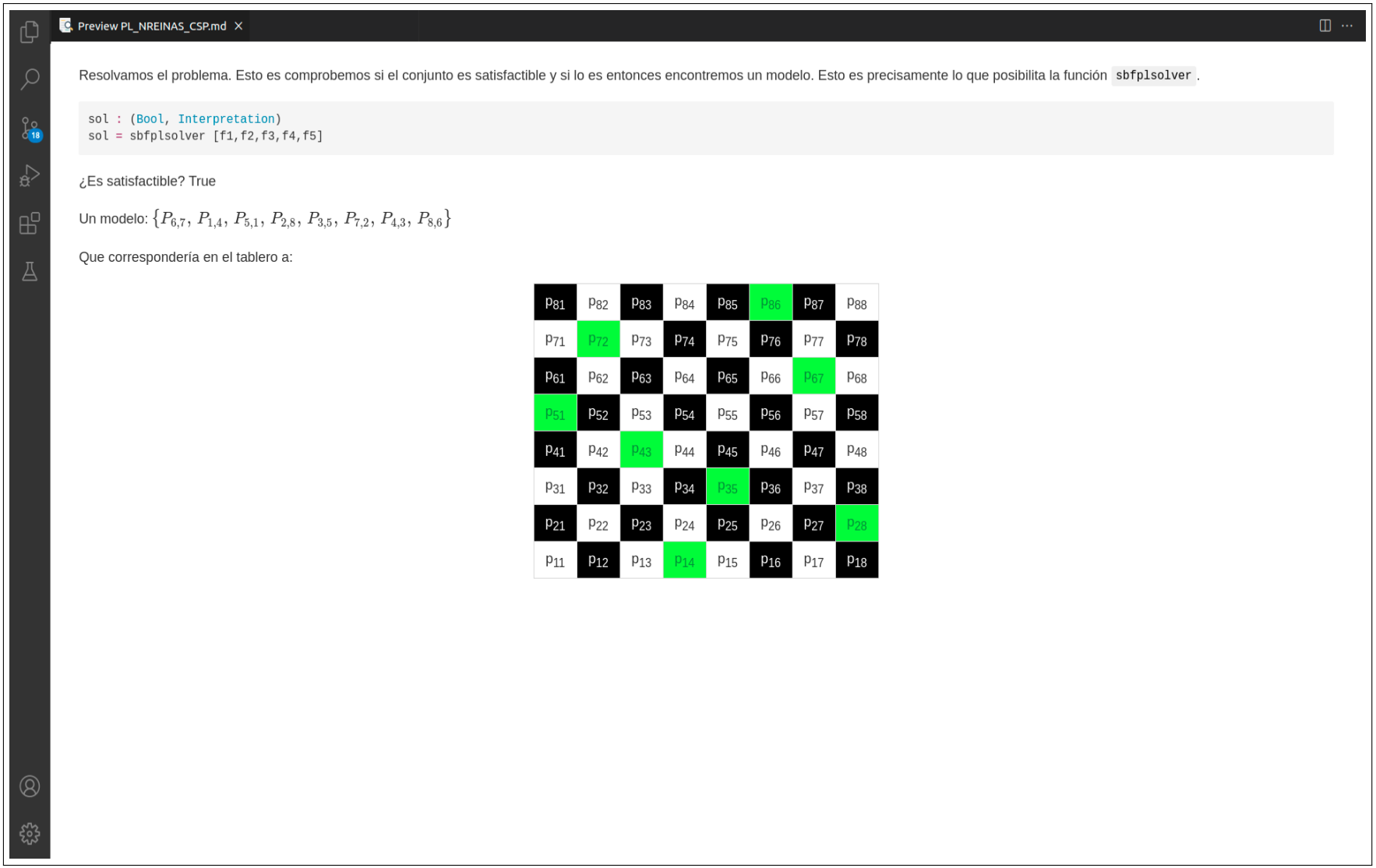


Figura 4.9: Ejemplo de uso de LogicUS.PL.CSP
Fuente propia. Creada con litvis

4.2.11. Implementación de LogicUS.FOL

Al igual que se ha presentado, de forma detallada, algunos de los aspectos más relevantes de la implementación realizada para la Lógica Proposicional, en este apartado describiremos los aspectos y funciones más relevantes destinados al trabajo con la Lógica de Primer de Orden.

Visión global de LogicUS.FOL

A modo de visión periférica, se expone una breve descripción de las principales funcionalidades y contenidos incluidos en cada uno de los módulos que constituyen el paquete.:

- **LogicUS.FOL.SintaxSemantics**: Constituye la base de la lógica de primer orden, por lo que expone la sintaxis para la definición de fórmulas y sustituciones. Además, detalla la semántica de FOL incluyendo las funciones de definición, valoración y representación de L-Structures en formato string y Latex para fórmulas, sustituciones, interpretaciones, ...
- **LogicUS.FOL.SemanticTableaux**: Desarrolla todas las herramientas necesarias para trabajar con tableros semánticos en FOL, distinguiendo los diferentes tipos de fórmulas y reglas y permitiendo además la visualización del tablero completo.
- **LogicUS.FOL.NormalForms**:Contiene las funciones necesarias para realizar la transformación de las fórmulas en formas equivalentes (Prenex) y equiconsistentes (Skolem, NNF, CNF, DNF), que son utilizadas luego en los algoritmos de decisión (refutación).
- **LogicUS.FOL.Clauses**: Proporciona algunas funciones que permiten trabajar con cláusulas proposicionales: definición, operaciones, transformación de fórmulas, conjuntos clausales, parsers, representación, etc.
- **LogicUS.FOL.Herbrand**: Define las funciones necesarias para la aplicación de las Obras Herbrand (Universo, interpretaciones, modelos, ...)
- **LogicUS.FOL.Unification**: Define las funciones para trabajar con los algoritmos de unificación: términos y átomos MGU.
- **LogicUS.FOL.Resolution**: Define las funciones para trabajar con los algoritmos de resolución, proporcionando además los mecanismos necesarios para su representación tanto en formato cadena, como en formatos visualizables (DOT).

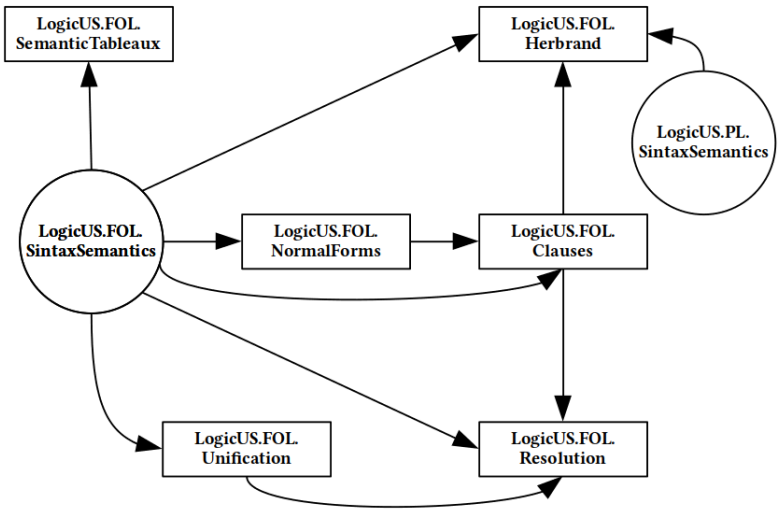


Figura 4.10: Dependencias internas de los módulos de LogicUS.FOL

Visto esto, y antes de pasar a detallar los contenidos de cada uno de los módulos, vamos a exponer la organización y dependencias internas entre los módulos comentados previamente. La *figura 4.10* presenta un diagrama de las dependencias internas de dichos módulos, donde queda patente la estructura comentada con anterioridad, en la que el módulo *LogicUS.FOL.SintaxSemantics* es la base sobre la que actúan el resto de los módulos. Además, se expone también la importancia del módulo *LogicUS.FOL.NormalForms* y *LogicUS.FOL.Clauses* encargados de transformación clausal de las fórmulas.

Por ello precisamente empezaremos detallando el contenido del módulo *LogicUS.FOL.SintaxSemantics* que nos permitirá entender las posteriores implementaciones realizadas en el resto de módulos del proyecto.

4.2.12. LogicUS.FOL.SintaxSemantics

Como hemos señalado, es el módulo que establece la base sobre la que construir la implementación del formalismo lógico en Lenguajes de Primer Orden, definiendo las estructuras que almacenan las fórmulas, sustituciones, interpretaciones y L-estructuras, y proporciona las funciones necesarias para la aplicación de las sustituciones, renombramientos, clausuras, funciones de evaluación, generación de árboles de formación, representación de las fórmulas, etc.

Sintaxis de fórmulas de Primer Orden en LogicUS

Comenzaremos presentando la sintaxis y definiendo las estructuras que almacenarán las fórmulas y que permitirán trabajar con ellas.Tengamos en cuenta que, al igual que ocurría con las proposicionales, las fórmulas de la Lógica de Primer Orden tienen una estructura recursiva, por lo que teniéndola presente podemos definir en Elm el tipo que las albergue. Nótese que el elemento básico d LP eran las variables proposicionales (átomos) que eran fórmulas en sí mismas. Sin embargo, en LPO existen elementos de menor nivel que las fórmulas, que corresponden a los términos.

Dentro de los términos distinguimos, en su momento, entre variables, constantes y funciones. Sin embargo, las constantes son, en realidad, funciones de aridad 0 (sin parámetros), y por ello en la implementación distinguiremos sólo dos casos para los términos, las variables y las funciones.

Las variables constan únicamente de un identificador (cadenas de texto, opcionalmente indexadas). Las variables pueden indexarse con una lista de subíndices y con un superíndices (que está pensado para el renombramiento de las variables, en caso necesario). Por su parte, las funciones constan de un identificador y una lista de términos que corresponden a los argumentos de la función.

Las fórmulas, por su parte, poseen como elemento fundamental (átomo) a los predicados, que constan de un identificador y una lista de términos (correspondientes a los argumentos del predicado). Nótese que las variables proposicionales corresponden, en realidad, a predicados de aridad 0, luego el predicado se podría decir que corresponde a la extensión natural de las variables proposicionales. Se reserva como predicado destacado, el predicado de igualdad. Son también predicados la fórmula insatisfactible (predicado idénticamente falso) y la fórmula válida (predicado idénticamente verdadero).

A partir de los predicados se pueden formar fórmulas mediante las conectivas de origen proposicional (conjunción, disyunción, implicación y equivalencia) y a partir de los cuantificadores existencial y universal. Nótese que estos dos últimos son la generalización de los operadores paramétricos, tratados en los CSPs, pero que definen una disyunción y conjunción, respectivamente, de infinitas fórmulas. De forma que puede observarse, con claridad, que la Lógica de Primer Orden, es una extensión natural de la Lógica Proposicional, y en ese aspecto, las estructuras implementadas para su definición corresponden también a dicha extensión.

```
type alias Ident =
  ( String, List Int )

type alias Variable =
( String, List Int, Int )

type Term
  = Var Variable
  | Func Ident (List Term)

type FormulaFOL
  = Pred Ident (List Term)
  | Equal Term Term
  | Neg FormulaFOL
  | Conj FormulaFOL FormulaFOL
  | Disj FormulaFOL FormulaFOL
  | Impl FormulaFOL FormulaFOL
  | Equi FormulaFOL FormulaFOL
  | Exists Variable FormulaFOL
  | Forall Variable FormulaFOL
  | Insat
  | Taut
```

Código 4.32: Estructura de las Fórmulas FOL en LogicUS

También de manera análoga a lo propuesto para la lógica proposicional, los conjuntos de fórmulas se definen como listas de fórmulas de Primer Orden (debido a que los elementos de un `Set` han de ser de la clase `comparable` y elm no permite definir nuevos tipos con dicha clase, ni con ninguna otra).

Como ya se ha expuesto, en otros puntos, las fórmulas pueden definirse directamente a través de los constructores, pero resulta más natural el uso de la notación infija, para lo cuál se ha definir un Parser que lea las fórmulas desde cadenas de texto. En este aspecto el parser definido establece la siguiente notación en la escritura de las fórmulas y términos.

- Las variables corresponden a cadenas de caracteres, el primero en minúscula, y opcionalmente indexados por números naturales, que han de ser escritos entre los símbolos `_{` y `}` y separados por comas. También se puede utilizar un superíndice correspondiente a un número natural (el 0 queda reservado para la ausencia de superíndice) descrito entre `^{` y `}` aunque está pensado únicamente para ser utilizado al renombrar variables. Algunos ejemplos de variables: `x`, `y_{1}`, `xA_{1,1}`, `x^{1}`.
- Las funciones son descritas de forma análogas a las variables pero precedidas del símbolo `*`.Además los argumentos, en caso de que los haya, se especifican entre paréntesis y separados por comas. Son ejemplos de constantes `*a`, `*b_{1}`, `*john`, y de funciones (no constantes): `*f(x)`, `*g_{1}(x,*a)`, `*father(*john)`,...
- Los predicados se describen de forma similar a las funciones, como cadenas de carácteres, el primero en mayúscula, y seguidos, si procede, de una lista de términos, especificados entre paréntesis y separados por comas, de igual manera a lo presentado para las funciones. Son ejemplos de predicados `P`, `Q_{1}(x)`, `Uncle(*john, *paul)`,...
- El uso de las conectivas es equivalente al propuesto para la lógica proposicional, utilizando `&` para la conjunción, `|` para la disyunción, `->` para la implicación, `<->` para la equivalencia y ¬ o `-` para la negación con la prioridad clásica (negación, conjunción, disyunción, implicación, equivalencia) y el uso de los paréntesis para indicar otra prioridad de asociación.
- Los cuantificadores se describen como `!E` para el existencial `!A` para el universal, seguidos por la variable a la que afectan indicada entre corchetes, `[` y `]`, y de la fórmula cuantificada. A modo de ejemplo, la propiedad de inductiva sobre una función f puede expresarse como: `!A[x_{1}]!A[x_{2}] (*f(x_{1})=*f(x_{2}) ->*x_{1} = *f(x_{2}) )`, o la pertenencia de un valor a al rango de una función g como `!E[x](*f(x)=*a) )`.
- Al igual que en la lógica proposicional la fórmula válida se define por `!T` y la fórmula insatisfactible por `!F`.

- No se deben poner los paréntesis externos de las fórmulas, ya que el Parser los pondrá automáticamente, por lo que su uso o no, resulta irrelevante.

En la Lógica de Primer Orden, además de las fórmulas es preciso hablar como elemento fundamental de la sintaxis de las sustituciones. Una sustitución no es más que una aplicación que asigna a una variable un término. Dicha aplicación es representada en LogicUS como un diccionario en el que las claves corresponden a variables y los valores a términos. Además de la definición de diccionario estándar que provee el propio Elm, LogicUS proporciona (a través de un parser) la definición de sustituciones desde cadenas de forma que han de escribirse con la forma {`variable1/valor1, variable2/valor2,...`} siguiendo para las variables y para los valores (términos) las notaciones descritas anteriormente para el parser de fórmulas.

A parte de la definición de esos dos tipos fundamentales se definen otras muchas funciones que permiten trabajar con las fórmulas destinadas a la comprobación de que una fórmula está bien formada, el cálculo y representación de los árboles de formación, la obtención de instancias y variables libres y ligadas en una fórmula, la clausura de fórmulas, la aplicación de sustituciones, renombramiento de variables, entre otras. Todas las funciones disponibles pueden consultarse en el módulo *LogicUS.FOL.SintaxSemantics* del paquete publicado.

Tras las cuestiones relacionadas con la implementación sintáctica pasaremos en el siguiente punto a abordar la implementación de los elementos semánticos de la Lógica de Primer Orden.

Semántica de los Lenguajes de Primer Orden en LógicUS

Mientras que en LP la semántica consistía, básicamente, en establecer los valores de verdad de las variables, y operar los mismos según las conectivas, en los lenguajes de primer orden contienen variables, constantes, símbolos de función, símbolos de predicado cuantificadores, ...

De forma que la semántica difiere ligeramente de la semántica de LP. Recuérdese que en los apartados teóricos tratamos formalmente este aspecto y se definió como elemento principal de la semántica la L-estructura. Una L-estructura consta de un conjunto no vacío, universo U, una conjunto de aplicaciones una para cada símbolo de función del lenguaje (de aridad k) y a cada k-tupla de U^k le asigna un elemento de U y un conjunto de aplicaciones una para cada símbolo de predicado del lenguaje (de aridad k) y a cada k-tupla de U^k le asigna un elemento de 0 o 1, según dicha tupla cumpla el predicado o no.

En LogicUS la implementación de esta estructura sigue la definición, lógicamente, pero las acota al trabajo con tipos comparables, universos finitos y describe las aplicaciones a través de diccionarios. Para ello se toma una interpretación como un *record* con tres propiedades:

- `const`: Un diccionario que asigna a cada símbolo de constante un elemento del universo.
- `func`: Un diccionario que asigna a cada símbolo de función, una aridad k y un diccionario tal que a toda posible k-tupla (expresada en forma de lista) de elementos del universo le asigna un elemento del universo.
- `pred`: Un diccionario que asigna a cada símbolo de predicado, una aridad k y un conjunto con las k-tuplas (expresadas en forma de listas) de elementos del dominio que verifican dicho predicado.

Nótese que no podemos condicionar, estructuralmente, algunas de las restricciones comentadas. Por ello es necesario recurrir a una función que verifique si se cumplen las condiciones expuestas, de forma que si la L-estructura no es válida entonces una fórmula no será valorable por la misma. La función `lStructureIsValid` realiza precisamente las comprobaciones previamente descritas.

En concreto están definidas como:

```
type alias L_Structure comparable =
  ( Set comparable
  , { const : Dict Ident comparable
    , func : Dict Ident ( Int, Dict (List comparable) comparable )
    , pred : Dict Ident ( Int, Set (List comparable) )
  }
  )

lStructureIsValid : L_Structure comparable -> Bool
lStructureIsValid ( u, i ) =
  (List.all (\v -> Set.member v u) <| Dict.values i.const)
    && (List.all
        (\ ( ar, f_M ) ->
          (List.all (\v -> Set.member v u) <| Dict.values f_M)
            && (List.length (Dict.keys f_M) == Set.size u ^ ar)
            && List.all (\t -> List.member t (Dict.keys f_M))
              (LE.cartesianProduct <| List.repeat ar (Set.toList u))
        )
      <|
        Dict.values i.func
      )
    && (List.all
      (\ ( ar, p_M ) ->
        List.all (\e -> List.length e == ar
          && List.all (\v -> Set.member v u) e) <| Set.toList p_M
      )
    <|
      Dict.values i.pred
  )
```

Código 4.33: Definición del tipo L-Structure en LogicUS

Con ello, podemos ya definir la evaluación de términos y fórmulas respecto a una interpretación (L-estructura). Nótese que sólo es posible llevar a cabo la evaluación de términos y fórmulas cerradas.

- La evaluación de términos corresponde a:
  - Si el término corresponde a una variable entonces, si esa variable está cuantificada, en una fórmula, entonces en la memoria de variables se dispondrá del valor del universo asociado a la misma. En otro caso no será interpretable.
  - Si el término corresponde a una constante, entonces el valor de la misma corresponderá al elemento definido en el diccionario de constantes de la L-estructura.
  - Si el término corresponde a una función (no constante), entonces primero serán evaluados los argumentos y después se evaluará (según el diccionario de funciones) la función sobre los argumentos. Si alguno de los elementos (argumentos o la función) no es interpretable, entonces no lo será el término en su conjunto.
- La evaluación de las fórmulas se realiza en base al establecimiento del valor de un predicado sobre una tupla, a la operación con las conectivas con dicho valor (igual que en la lógica proposicional) y a la evaluación de las fórmulas cuantificadas universal y existencialmente.
  - Para los predicados, el cálculo del valor de verdad se reduce a la interpretación, primero, de los argumentos del mismo y al predicado en sí después, de forma que si pertenece al conjunto asociado al símbolo de predicado correspondiente entonces se evaluará verdadero, y en otro caso se evaluará falso. Al igual que con las funciones si la aridad especificada no coincide, el símbolo no pertenece al diccionario de predicados o alguno de los argumentos no es interpretable, entonces la fórmula en su conjunto no será, tampoco, interpretable.
  - Para las conectivas basta, al igual que en la lógica proposicional, aplicar las funciones de verdad asociadas a las mismas.
  - Para la cuantificación existencial (nótese que equivaldría a $\bigvee_{i \in U}(F)$), por lo que basta considerar la disyunción de los valores de verdad de las fórmulas que resultan de sustituir la variable correspondiente al cuantificador universal por cada uno de los elementos del dominio.

  - Para la cuantificación universal (nótese que equivaldría a $\bigwedge `_{i \in U}(F)$), por lo que basta considerar la disyunción de los valores de verdad de las fórmulas que resultan de sustituir la variable correspondiente al cuantificador universal por cada uno de los elementos del dominio.

- La evaluación de un conjunto de fórmulas cerradas corresponde a la conjunción de las valoraciones de cada una de las fórmulas por separado.

En el *código 4.34* se presenta la implementación concreta de los métodos comentados, de forma que puede comprobarse que se ajustan a lo descrito previamente.

```
termInterpretation : Term -> L_Structure comparable -> Maybe comparable
termInterpretation t ls =
  if lStructureIsValid ls then
    termInterpretationAux t Dict.empty ls

  else
    Nothing

termInterpretationAux :
  Term -> Dict Variable comparable -> L_Structure comparable -> Maybe comparable
termInterpretationAux t vars_val ( u, i ) =
  case t of
    Var x ->
      Maybe.andThen
        (\v -> if Set.member v u then Just v else Nothing)
        (Dict.get x vars_val)

    Func f_ [] -> Dict.get f_ i.const

    Func f_ ts ->
      Maybe.andThen
        (\ ( ar, f_M ) ->
          if List.length ts == ar then
            Maybe.andThen
              (\args -> Dict.get args f_M)
              (termsInterpretationAux ts vars_val ( u, i ))

          else
            Nothing
        )
        (Dict.get f_ i.func)

termsInterpretation : List Term -> L_Structure comparable -> Maybe (List comparable)
termsInterpretation ts ls =
  if lStructureIsValid ls then
    ME.combine <| List.map (\t -> termInterpretationAux t Dict.empty ls) ts

  else
    Nothing

termsInterpretationAux :
  List Term -> Dict Variable comparable -> L_Structure comparable
    -> Maybe (List comparable)
termsInterpretationAux ts vars_val ls =
  ME.combine <| List.map (\t -> termInterpretationAux t vars_val ls) ts

ffolValuation : FormulaFOL -> L_Structure comparable -> Maybe Bool
ffolValuation f ls =
  if (not << ffolIsClosed) f then
    Nothing

  else if lStructureIsValid ls then
    ffolValuationAux f Dict.empty ls

  else
    Nothing
```

```
ffolValuationAux :
  FormulaFOL -> Dict Variable comparable -> L_Structure comparable -> Maybe Bool
ffolValuationAux f vars_val ( u, i ) =
  case f of
    Pred p_ ts ->
      Maybe.andThen
        (\ ( ar, p_M ) ->
          if List.length ts == ar then
            Maybe.map
              (\args -> Set.member args p_M)
              (termsInterpretationAux ts vars_val ( u, i ))

          else
            Nothing
        )
        (Dict.get p_ i.pred)

    Equal t1 t2 ->
      Maybe.map2 (==)
        (termInterpretationAux t1 vars_val ( u, i ))
        (termInterpretationAux t2 vars_val ( u, i ))

    Neg g ->
      Maybe.map not (ffolValuationAux g vars_val ( u, i ))

    Conj g h ->
      Maybe.map2 (&&)
        (ffolValuationAux g vars_val ( u, i )) (ffolValuationAux h vars_val ( u, i ))

    Disj g h ->
      Maybe.map2 (||)
        (ffolValuationAux g vars_val ( u, i )) (ffolValuationAux h vars_val ( u, i ))

    Impl g h ->
      Maybe.map2 (\x y -> not x || y)
        (ffolValuationAux g vars_val ( u, i )) (ffolValuationAux h vars_val ( u, i ))

    Equi g h ->
      Maybe.map2 (==)
        (ffolValuationAux g vars_val ( u, i )) (ffolValuationAux h vars_val ( u, i ))

    Exists x g ->
      Maybe.map (List.any identity) <| ME.combine <|
        List.map
          (\o -> ffolValuationAux g (Dict.insert x o vars_val) ( u, i ))
          (Set.toList u)

    Forall x g ->
      Maybe.map (List.all identity) <| ME.combine <|
        List.map
          (\o -> ffolValuationAux g (Dict.insert x o vars_val) ( u, i ))
          (Set.toList u)

    Insat ->
    Just False

    Taut ->
    Just True

sfolInterpretation : SetFOL -> L_Structure comparable -> Maybe Bool
sfolInterpretation fs ls =
  if List.any (not << ffolIsClosed) fs then
    Nothing

  else if lStructureIsValid ls then
    Maybe.map (List.all identity) <| ME.combine <|
      List.map (\f -> ffolValuation f ls) fs

  else
    Nothing
```

Código 4.34: Evaluación de fórmulas en LogicUS

Una vez vista la evaluación de las fórmulas hemos de pasar a abordar la resolución de la (in)satisfactibilidad, consecuencia lógica y cálculo de modelos. Recuérdese que en los fundamentos teóricos ya comentamos el carácter indecidible de la satisfactibilidad por lo que ni dicho problema, ni mucho menos el cálculo de todos los modelos son resolubles mecánicamente por ningún algoritmo. La insatisfactibilidad, la consecuencia y la validez lógica sí son semidecidibles, sin embargo, en el módulo no se provee una función que los resuelva. De hecho el resto de módulos estarán dedicados precisamente a abordar este problema.

Por último en el módulo se dedica un último apartado a la representación de fórmulas, sustituciones, y L-estructuras tanto en formato de cadena unicode como en formato Latex; así como la representación del árbol de formación tanto en formato cadena, como en formato DOT, renderizable con GraphViz.

Como cierre del módulo se expone, en la *figura 4.11* un ejemplo de aplicación de algunas de las funciones más relevantes del módulo.

4.2.13. LogicUS.FOL.SemanticTableaux

En este módulo se exponen las funciones y tipos necesarios para la realización de los tableros semánticos. En la construcción del tablero resultan de interés los tipos de fórmulas y las reglas de derivación de las mismas.

Comencemos por los tipos de fórmulas. Recuérdese que para la resolución mediante tableros en LP se consideraron cuatro clases de fórmulas (o reglas), las de tipo α (fórmulas con carácter conjuntivo), tipo β (fórmulas de carácter disyuntivo), tipo dN (doble negación) y literales L. Estas categorías se mantienen en la extensión a la Lógica de Primer Orden, añadiéndose dos nuevas reglas, γ y δ, correspondientes al tratamiento del comportamiento universal y el comportamiento existencial, respectivamente, y un tipo más para el tratamiento de la igualdad.

Se define el tipo `FormulaFOLType` con las clases comentadas (constructores), y la función `ffolType` que devuelve el tipo correspondiente a cada clase de fórmula.

Una vez definidos los tipos, o equivalentemente las clases de reglas, hemos de establecer el modo de operación de cada una de las mismas. Dado que ahora es necesario realizar sustituciones en el tratamiento de la cuantificación, se distinguen dos casos principales en el tratamiento de un paso deductivo, diferenciando las reglas γ y δ del resto, en las que no participan los cuantificadores. Aunque se provee la base para el trabajo con la igualdad no se han implementado, por el momento, las reglas deductivas asociadas a dicho predicado destacado, por lo que, actualmente, es tratado como un predicado más.

```
type FormulaPLType  =
    L | E | DN | A | B | I | T

fplType : FormulaPL -> FormulaPLType
fplType f =
    case f of
        Atom _ -> L

        Neg (Atom _) -> L

        Neg (Neg _) -> DN

        Neg (Conj _ _) -> B

        Neg (Disj _ _) -> A

        Neg (Impl _ _) -> A

        Neg (Equi _ _) -> B

        Neg Insat -> T

        Neg Taut -> I

        Conj _ _ -> A

        Disj _ _ -> B

        Impl _ _ -> B

        Equi _ _ -> A

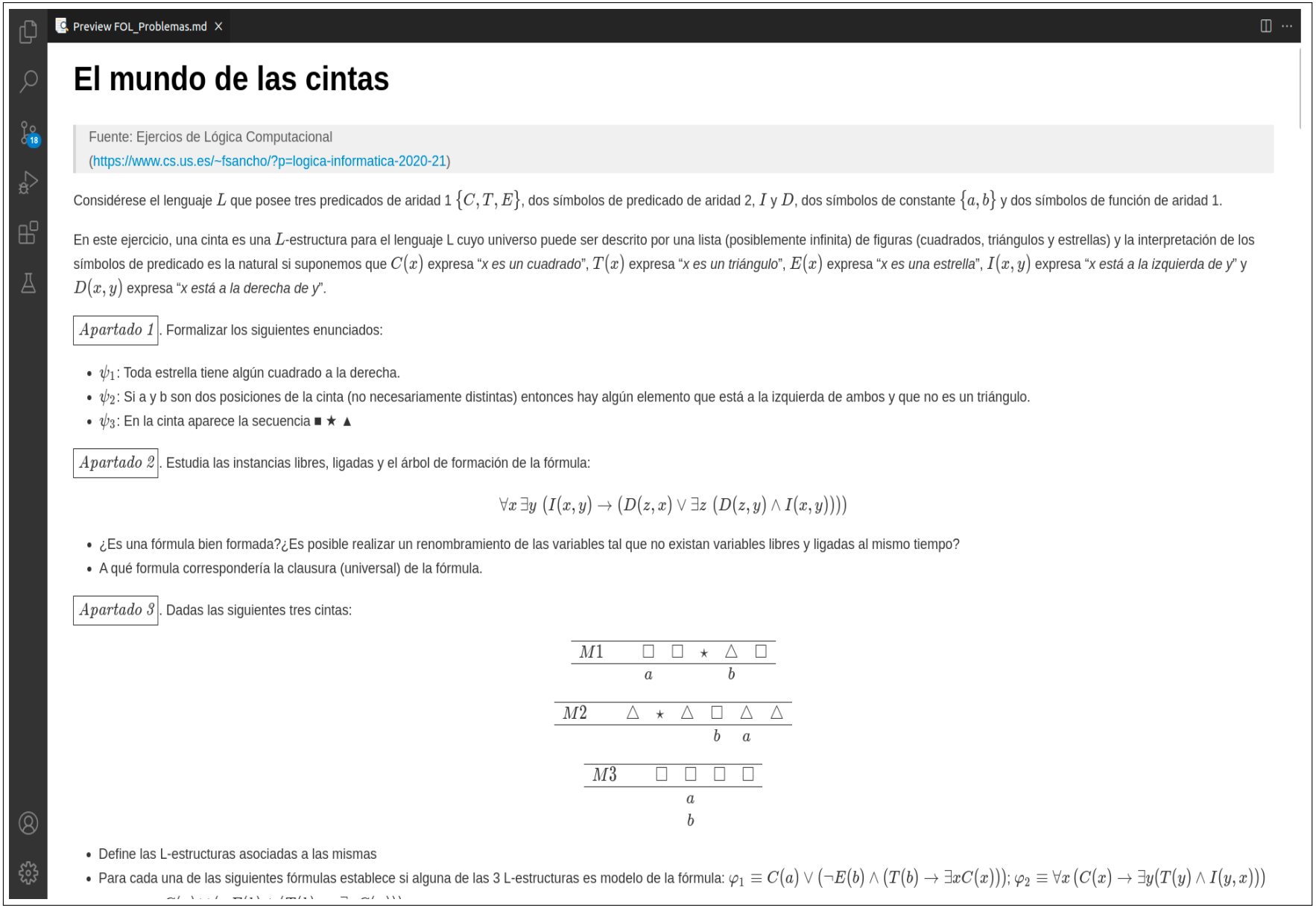
        Insat -> I

        Taut -> T
```

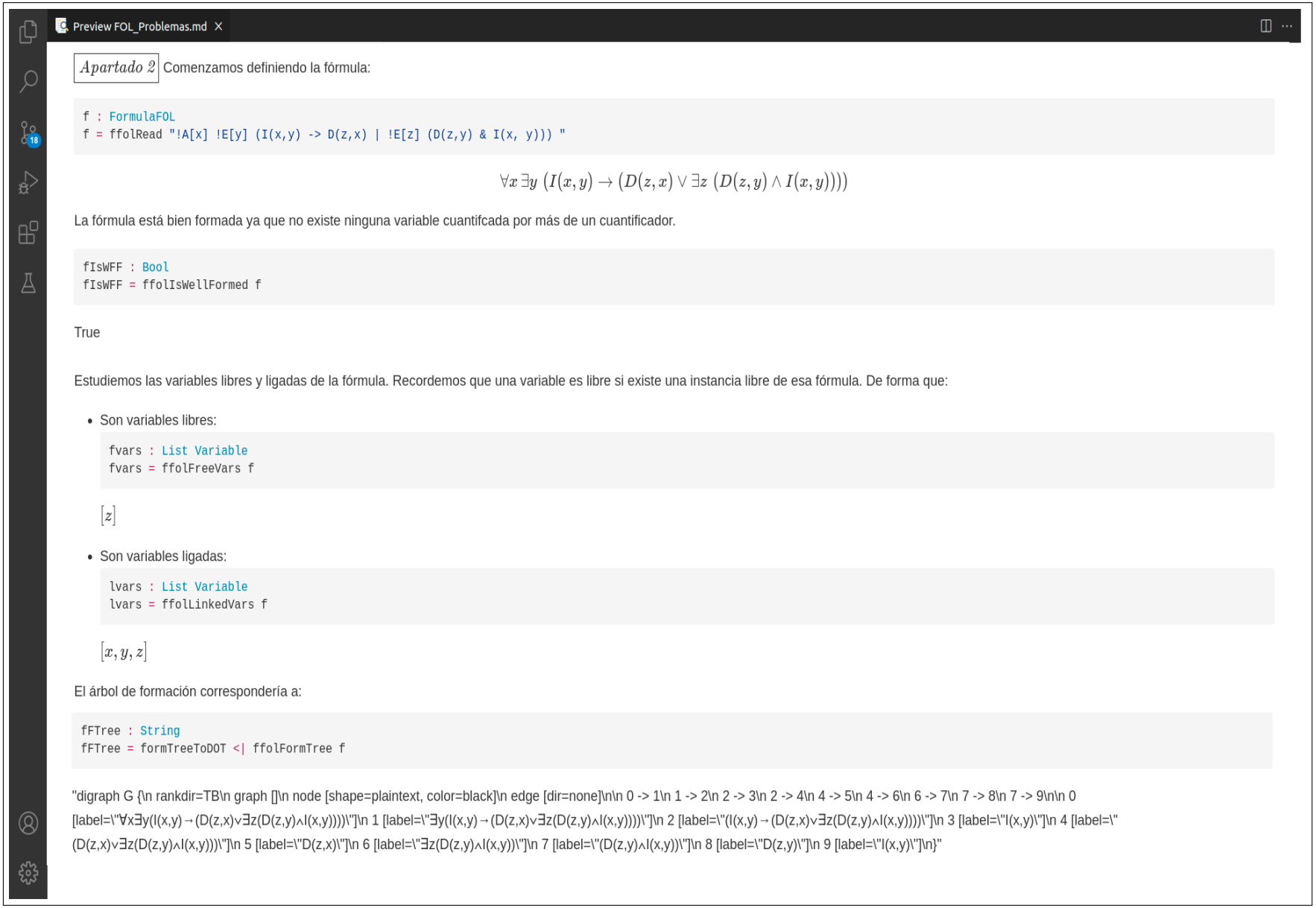
Código 4.35: Tipos de fórmulas/reglas en Tableros Semáticos en LPO en LogicUS

Así, se distinguen dos funciones para la aplicación de los pasos deductivos, correspondientes a `ffolUncuantifiedComponents` y `ffolCuantifiedComponents`. Además de dichas funciones es necesaria también otra función que se encargue de ir generando las nuevas constantes que son introducidas en la

cuantificación existencial (y excepcionalmente y una sola vez en la universal).


Preview FOL_Problemas.md
El mundo de las cintas
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Considérese el lenguaje L que posee tres predicados de aridad 1 $\{C, T, E\}$, dos símbolos de predicado de aridad 2, I y D, dos símbolos de constante $\{a, b\}$ y dos símbolos de función de aridad 1.
En este ejercicio, una cinta es una L-estructura para el lenguaje L cuyo universo puede ser descrito por una lista (posiblemente infinita) de figuras (cuadrados, triángulos y estrellas) y la interpretación de los símbolos de predicado es la natural si suponemos que $C(x)$ expresa "*x es un cuadrado*", $T(x)$ expresa "*x es un triángulo*", $E(x)$ expresa "*x es una estrella*", $I(x, y)$ expresa "*x está a la izquierda de y*" y $D(x, y)$ expresa "*x está a la derecha de y*".
$Apartado\ 1$. Formalizar los siguientes enunciados:
ψ_1: Toda estrella tiene algún cuadrado a la derecha.
ψ_2: Si a y b son dos posiciones de la cinta (no necesariamente distintas) entonces hay algún elemento que está a la izquierda de ambos y que no es un triángulo.
ψ_3: En la cinta aparece la secuencia ■ ★ ▲
$Apartado\ 2$. Estudia las instancias libres, ligadas y el árbol de formación de la fórmula:
$$\forall x\, \exists y\, (I(x, y) \to (D(z, x) \vee \exists z\, (D(z, y) \wedge I(x, y))))$$
¿Es una fórmula bien formada?¿Es posible realizar un renombramiento de las variables tal que no existan variables libres y ligadas al mismo tiempo?
A qué formula correspondería la clausura (universal) de la fórmula.
$Apartado\ 3$. Dadas las siguientes tres cintas:
M1 □ □ ⋆ △ □
a b
M2 △ ⋆ △ □ △ △
b a
M3 □ □ □ □
a
b
Define las L-estructuras asociadas a las mismas
Para cada una de las siguientes fórmulas establece si alguna de las 3 L-estructuras es modelo de la fórmula: $\varphi_1 \equiv C(a) \vee (\neg E(b) \wedge (T(b) \to \exists x C(x)))$; $\varphi_2 \equiv \forall x\, (C(x) \to \exists y (T(y) \wedge I(y, x)))$


Preview FOL_Problemas.md
Solución
Antes de nada, vamos a importar el módulo de Sintaxis y Semántica en Primer Orden y algunos otros paquetes que nos harán falta para poder trabajar con las fórmulas.
import LogicUS.FOL.SintaxSemantics exposing (..)
import Set exposing (Set)
import Dict exposing (Dict)
ffolRead : String -> FormulaFOL
ffolRead s = ffolReadExtraction <| ffolReadFromString s
Apartado 1 Formalicemos los enunciados:
ψ1: Toda estrella tiene algún cuadrado a la derecha.
psi1 : FormulaFOL
psi1 = ffolRead "!A[x](E(x) -> !E[y](C(y) & D(y,x)))"
∀x (E(x) → ∃y (C(y) ∧ D(y, x)))
ψ2: Si a y b son dos posiciones de la cinta (no necesariamente distintas) entonces hay algún elemento que está a la izquierda de ambos y que no es un triángulo.
psi2 : FormulaFOL
psi2 = ffolRead "!E[x](¬T(x) & I(x,*a) & I(x,*b))"
∃x (¬T(x) ∧ (I(x, a) ∧ I(x, b)))
ψ3: En la cinta aparece la secuencia ■ ★ ▲
psi3 : FormulaFOL
psi3 = ffolRead "!E[x] !E[y] !E[z] (C(x) & E(y) & T(z) & I(x, y) & I(y, z) & ¬!E[t] (I(t,y) & D(t, x) | I(t,z) & D(t,y)))"
∃x ∃y ∃z (C(x) ∧ (E(y) ∧ (T(z) ∧ (I(x, y) ∧ (I(y, z) ∧ ¬∃t ((I(t, y) ∧ D(t, x)) ∨ (I(t, z) ∧ D(t, y)))))))


Preview FOL_Problemas.md
Apartado 2 Comenzamos definiendo la fórmula:
f : FormulaFOL
f = ffolRead "!A[x] !E[y] (I(x,y) -> D(z,x) | !E[z] (D(z,y) & I(x, y))) "
$$\forall x \exists y \left(I(x,y) \rightarrow \left(D(z,x) \vee \exists z \left(D(z,y) \wedge I(x,y)\right)\right)\right)$$
La fórmula está bien formada ya que no existe ninguna variable cuantifcada por más de un cuantificador.
fIsWFF : Bool
fIsWFF = ffolIsWellFormed f
True
Estudiemos las variables libres y ligadas de la fórmula. Recordemos que una variable es libre si existe una instancia libre de esa fórmula. De forma que:
Son variables libres:
fvars : List Variable
fvars = ffolFreeVars f
$[z]$
Son variables ligadas:
lvars : List Variable
lvars = ffolLinkedVars f
$[x, y, z]$
El árbol de formación correspondería a:
fFTree : String
fFTree = formTreeToDOT <| ffolFormTree f
"digraph G {\n rankdir=TB\n graph []\n node [shape=plaintext, color=black]\n edge [dir=none]\n\n 0 -> 1\n 1 -> 2\n 2 -> 3\n 2 -> 4\n 4 -> 5\n 4 -> 6\n 6 -> 7\n 7 -> 8\n 7 -> 9\n\n 0 [label=\"∀x∃y(I(x,y)→(D(z,x)∨∃z(D(z,y)∧I(x,y))))\"]\n 1 [label=\"∃y(I(x,y)→(D(z,x)∨∃z(D(z,y)∧I(x,y))))\"]\n 2 [label=\"(I(x,y)→(D(z,x)∨∃z(D(z,y)∧I(x,y))))\"]\n 3 [label=\"I(x,y)\"]\n 4 [label=\"(D(z,x)∨∃z(D(z,y)∧I(x,y)))\"]\n 5 [label=\"D(z,x)\"]\n 6 [label=\"∃z(D(z,y)∧I(x,y))\"]\n 7 [label=\"(D(z,y)∧I(x,y))\"]\n 8 [label=\"D(z,y)\"]\n 9 [label=\"I(x,y)\"]\n}"

Preview FOL_Problemas.md

Representándolo:

Renombremos las variables:

```
f2 : FormulaFOL
f2 = renameVars <| f
```

$$\forall x^1 \exists y^1 \left(I(x^1, y^1) \to \left(D(z, x^1) \vee \exists z^1 \left(D(z^1, y^1) \wedge I(x^1, y^1)\right)\right)\right)$$

Preview FOL_Problemas.md

La fórmula no es cerrada

```
f2IsClosed : Bool
f2IsClosed = ffolIsClosed f2
```

False

De forma que la clausura universal de la fórmula correspondería a:

```
f3 : FormulaFOL
f3 =  ffolUniversalClausure f2
```

$$\forall z^1 \forall x^1 \exists y^1 \left(I(x^1, y^1) \rightarrow \left(D(z^1, x^1) \vee \exists z^2 \left(D(z^2, y^1) \wedge I(x^1, y^1)\right)\right)\right)$$

Apartado 3. Vamos a definir las L-Estructuras asociadas a las cintas.

$M1$	$\square$	$\square$	$\star$	$\triangle$	$\square$
	a			b	

$M2$	$\triangle$	$\star$	$\triangle$	$\square$	$\triangle$	$\triangle$
				b	a	

$M3$	$\square$	$\square$	$\square$	$\square$
		a		
		b		

Recuérdese que una L-estructura constaba de:

- Un Universo U. En este caso el universo corresponderá a las posiciones.
- Una interpretación de las constantes a y b. En este caso las posiciones asociadas a los indicadores.
- Una interpretación para los símbolos de función del lenguaje. No hay símbolos de función en L.
- Una interpretación para cada predicado. Tomaremos el significado natural dado en el enunciado.

De forma que en LogicUS debemos definir:

Preview FOL_Problemas.md

- El universo como un conjunto de elementos de la clase comparable (en nuestro caso enteros).

```
u1 : Set Int
u1 = Set.fromList <| [1,2,3,4,5]

u2 : Set Int
u2 = Set.fromList <| [1,2,3,4,5,6]

u3 : Set Int
u3 = Set.fromList <| [1,2,3,4]
```

- La interpretación de las constantes como un diccionario en el que a cada símbolo de constante le asignemos un elemento del dominio.

```
ic1 : Dict Ident Int
ic1 = Dict.fromList <| [(("a", []), 1), (("b", []), 4)]

ic2 : Dict Ident Int
ic2 = Dict.fromList <| [(("a", []), 5), (("b", []), 4)]

ic3 : Dict Ident Int
ic3 = Dict.fromList <| [(("a", []), 2), (("b", []), 2)]
```

- La interpretación de los predicados como un diccionario el que a cada símbolo de predicado se le asigna un natural correspondiente a su aridad y un conjunto de listas de elementos del universo, que son los que verifican el predicado

```
ip1 : Dict (String, List Int) (Int, Set (List Int))
ip1 =
  Dict.fromList <|
    [ (("I", []), (2, Set.fromList [ [1,2], [1,3], [1,4], [1,5], [2,3], [2,4], [2,5], [3,4], [3,5], [4,5] ]))

      , (("D", []),(2, Set.fromList [[5,4], [5,3], [5,2], [5,1], [4,3], [4,2], [4,1], [3,2], [3,1], [2,1]]))

      , (("C", []), (1, Set.fromList [ [1], [2], [5] ] ) )

      , (("E", []), (1, Set.fromList [ [3] ] ))

      , (("T", []), (1, Set.fromList [ [4] ] ))
    ]

ip2 : Dict (String, List Int) (Int, Set (List Int))
ip2 =
  Dict.fromList <|
    [ (("I", []), (2, Set.fromList <| List.concat <| List.map (\x -> List.map ( \ y -> [x, y]) (List.range (x+1) 6)) [1,2,3,4,5]))

      , (("D", []), (2, Set.fromList <| List.concat <| List.map (\x -> List.map ( \ y -> [x, y]) (List.range 1 (x-1))) [6,5,4,3,2]))

      , (("C", []), (1, Set.fromList [ [4] ] ) )
```

```
ip2 : Dict (String, List Int) (Int, Set (List Int))
ip2 =
  Dict.fromList <|
    [ (("I", []), (2, Set.fromList <| List.concat <| List.map (\x -> List.map ( \ y -> [x, y]) (List.range (x+1) 6)) [1,2,3,4,5]))

      , (("D", []), (2, Set.fromList <| List.concat <| List.map (\x -> List.map ( \ y -> [x, y]) (List.range 1 (x-1))) [6,5,4,3,2]))

      , (("C", []), (1, Set.fromList [ [4] ] ) )

      , (("E", []), (1, Set.fromList [ [2] ] ))

      , (("T", []), (1, Set.fromList [ [1], [3], [5], [6] ] ))
    ]

ip3 : Dict (String, List Int) (Int, Set (List Int))
ip3 =
  Dict.fromList <|
    [ (("I", []), (2, Set.fromList <| List.concat <| List.map (\x -> List.map ( \ y -> [x, y]) (List.range (x+1) 4)) [1,2,3]))

      , (("D", []), (2, Set.fromList <| List.concat <| List.map (\x -> List.map ( \ y -> [x, y]) (List.range 1 (x-1))) [4,3,2]))

      , (("C", []), (1, Set.fromList [ [1],[2],[3],[4] ] ) )

      , (("E", []), (1, Set.empty ))

      , (("T", []), (1, Set.empty ))
    ]
```

De forma que ahora las L-estructuras serían:

```
ls1 : L_Structure Int
ls1 = (u1, {const=ic1, func=Dict.empty, pred=ip1})

ls2 : L_Structure Int
ls2 = (u2, {const=ic2, func=Dict.empty, pred=ip2})


ls3 : L_Structure Int
ls3 = (u3, {const=ic3, func=Dict.empty, pred=ip3})
```

Ahora, para cada una de las fórmulas siguientes debemos decidir si alguna de las L-estructuras definidas la satisfacen.

- $\varphi_1 \equiv C(a) \vee (\neg E(b) \wedge (T(b) \rightarrow \exists x C(x)))$

```
f1 : FormulaFOL
f1 = ffolRead "C(*a)|(¬E(*b) & (T(*b) -> !E[x] (C(x))))"
```

$$(C(a) \vee (\neg E(b) \wedge (T(b) \rightarrow \exists x\, C(x))))$$

La evaluamos respecto a cada una de las interpretaciones.

- Respecto a M_1

```
evf1ls1 : Maybe Bool
evf1ls1 = ffolValuation f1 ls1
```

Just True

- Respecto a M_2

```
evf1ls2 : Maybe Bool
evf1ls2 = ffolValuation f1 ls2
```

Just True

- Respecto a M_3

```
evf1ls3 : Maybe Bool
evf1ls3 = ffolValuation f1 ls3
```

Just True

- $\varphi_2 \equiv \forall x\, (C(x) \rightarrow \exists y (T(y) \wedge I(y, x)))$

```
f2 : FormulaFOL
f2 = ffolRead "!A[x](C(x) -> !E[y](T(y) & I(y,x)))"
```

$$\forall x\, (C(x) \rightarrow \exists y\, (T(y) \wedge I(y, x)))$$

La evaluamos respecto a cada una de las interpretaciones.

- Respecto a M_1

```
evf2ls1 : Maybe Bool
evf2ls1 = ffolValuation f2 ls1
```

Just False

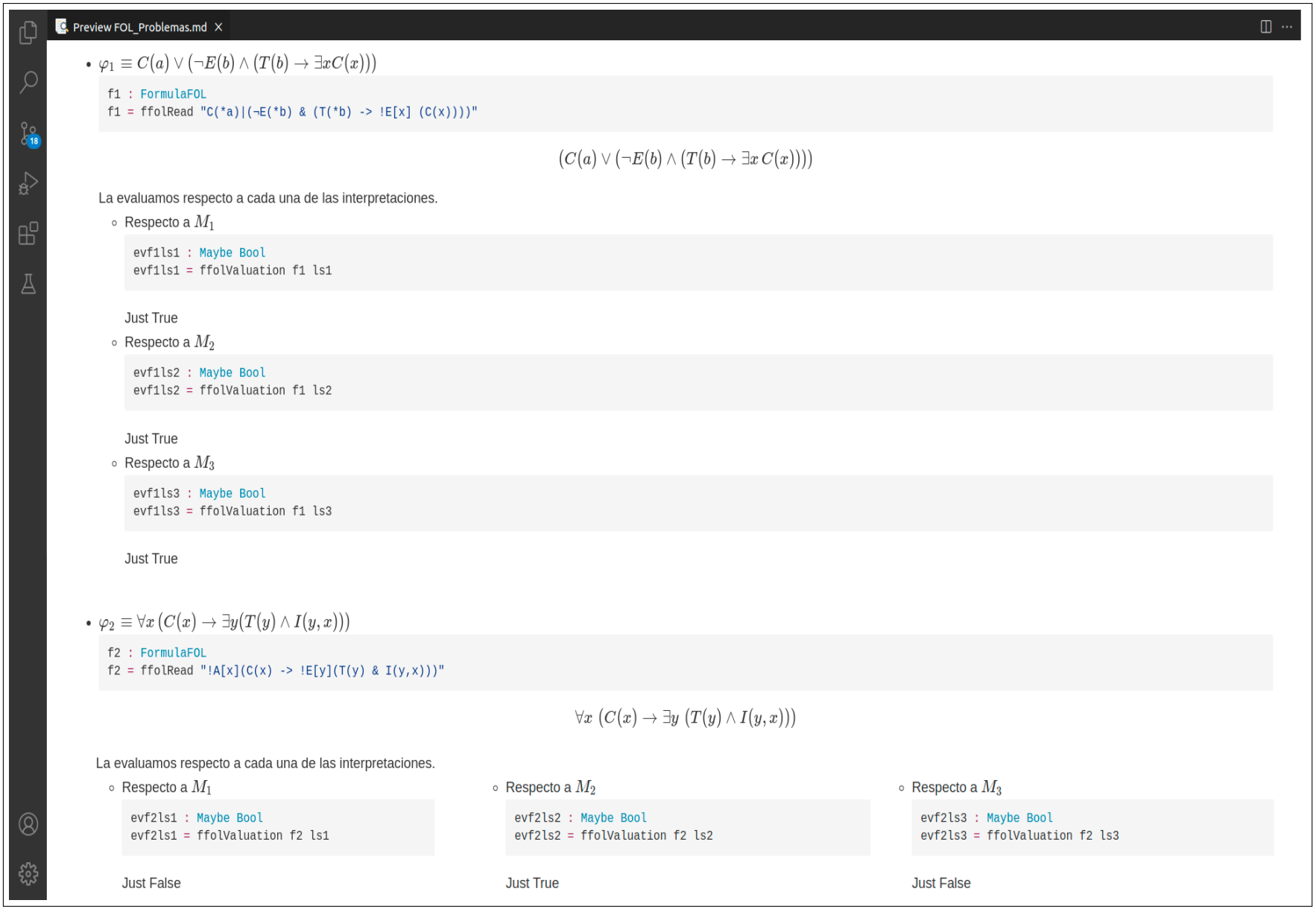


Figura 4.11: Ejemplo de uso de LogicUS.FOL.SintaxSemantics
Fuente propia. Creada con litvis

Con todo ello podemos realalizar la implementación del algoritmo de Tableros Semáticos propuesto en *algoritmo 2.6*. Recuérdese que el problema de la insatisfactibilidad es solo parcialmente decidible y, por tanto, el procedimiento mecánico puede no parar para un cierto conjunto de entrada. Dicho problema es originado por las fórmulas universales que se pueden usar infinitas veces en el proceso de deducción, una con cada término cerrado del mundo. Esto hace que tengamos que lidiar con algunas cuestiones relevantes:

- *Estructura del árbol.* Recuérdese que, a diferencia del Tablero proposicional, en el que cada nodo representaba el conjunto de fórmulas en cada paso deductivo, el Tablero Semántico para LPO, es formado tal que cada nodo contiene una fórmula y cada rama del árbol representa un conjunto de fórmulas formado por los distintos nodos que aparecen en dicha rama, y el conjunto será inconsistente si en dicha rama aparecen dos fórmulas complementarias. Para expresar la información de deducción para cada nueva fórmula generada, se indicará en la arista incidente "de entrada"la regla aplicada, las fórmulas implicadas y la sustitución asumida (sólo en las fórmulas cuantificadas). Para ello, se ha definido la estructura `TableauEdgeItem` que almacena dicha información y también la componente de la deducción (en las conectivas binarias, 1, para denotar que es la primera componente y, 2, para denotar que es la segunda).

- *Limitar la profundidad del árbol.* Dado que se está tratando con el problema de la insatisfactibilidad, que recordemos, era sólo parcialmente decidible en este caso, el procedimiento mecánico puede no parar, por tanto es imposible crear un tablero completo para todo conjunto de fórmulas inicial. Por ello, se limita la profundidad del árbol a una profundidad máxima (absoluta). De forma que en cada paso se mantiene la longitud de la rama considerada, y si supera esta profundidad máxima, entonces la rama es truncada y marcada como abierta.

- *Control de la usabilidad de las fórmulas.* Téngase en cuenta que, aunque la mayoría de los tipos de fórmulas son usables una única vez en el proceso deductivo, las fórmulas universales pueden utilizarse tantas veces como sea necesario, una para cada elemento (término cerrado) que aparezca en el conjunto de fórmulas, por lo que se mantiene en todo momento una lista con dichos términos (actualizándolo si procede). Además, hemos de controlar que no se realiza una deducción sobre la misma fórmula (universal) y el mismo elemento, por lo que los nodos (en especial, los universales) tendrán asociado el conjunto de elementos con los que ya ha sido resuelto.

  Dado el límite en la profundidad, es posible que se utilice una fórmula universal muchas veces antes de que otra sea utilizada por primera vez, pudiendo perder la capacidad deductiva por el uso inadecuado de la información que proporcionan las fórmulas. Para afrontar este hecho también se debe especificar un número máximo de utilización de una fórmula y se escogerá (teniendo en cuenta el orden de prioridad de las reglas dN, α, β, δ, γ) aquella fórmula universal que haya sido menos utilizada.

  Por todo ello, se define la estructura `TableauNodeItem` que almacena información sobre cada nodo. En concreto, la fórmula correspondiente a dicho nodo, el tipo de la misma (equivalente a la regla aplicable), el número de usos restantes para la misma (para las fórmulas no universales, un solo uso de la misma establecerá dicho parámetro a 0, dado que son usables una única vez), y la lista de términos respecto a los cuales se ha resuelto (sólo para las universales, para el resto corresponderá a una lista vacía). La estructura posee también un campo adicional por si, eventualmente, se quisiera implementar hacer uso de eliminación de los nodos irrelevantes en la deducción.

La implementación de todos los elementos relevantes, previamente comentados, se expone en el *código 4.36*. Adicionalmente a esos métodos, y al igual que en otros métodos presentados con anterioridad, se proporcionan los métodos necesarios para llevar a cabo la representación del árbol, tanto en formato cadena como en formato DOT, visualizable con GraphViz, con las funciones `semanticTableauToString` y `semanticTableauToDOT`.

Antes de pasar a describir el siguiente módulo implementado, se propone un ejemplo de uso del módulo que permita entender las funcionalidades que éste ofrece y la forma de uso de las mismas.

```
type alias TableauNodeItem =
    { f : FormulaFOL
    , t : FormulaFOLType
    , u : Int
    , ut : List Term
    , i : Int
    , ps : List Int
    }

type alias TableauEdgeItem =
    { r : FormulaFOLType
    , is : List Int
    , br : Int
    , s : Substitution
    }

semanticTableau : SetFOL -> Int -> Int -> FOLSemanticTableau
semanticTableau fs uMax dMax =
    let
        nodes =
            Dict.fromList <| List.indexedMap (\i f -> ( i, ( 0, { f = f, t = ffolType f, u = uMax, ut = [], i = i, ps = [] } ) )) fs

        edges =
            Dict.fromList <| List.indexedMap (\i v -> ( ( i, i + 1 ), v )) <| List.repeat (List.length fs - 1) Nothing

        m =
            uniqueConcatList [] <| List.concat <| List.map FOL_SS.ffolAllClosedTerms fs
    in
    let
        ( res_nodes, res_edges ) =
            semanticTableauAux m dMax uMax (List.length fs) (List.length fs) (List.length fs - 1) 0 nodes edges
    in
    Graph.fromNodesAndEdges
        (List.map (\( k, v ) -> Node k v) <| Dict.toList <| res_nodes)
        (List.map (\( ( s, t ), v ) -> Edge s t v) <| Dict.toList <| res_edges)


semanticTableauAux :
    List Term -> Int -> Int -> Int -> Int -> Int -> Int -> Dict Int ( Int, TableauNodeItem ) ->
      Dict ( Int, Int ) (Maybe TableauEdgeItem) -> ( Dict Int ( Int, TableauNodeItem ), Dict ( Int, Int ) (Maybe TableauEdgeItem) )
```

```
semanticTableauAux m dMax uMax nid nid2 nidp dcur nodes edges =
    case searchContradiction <| Dict.toList nodes of
        Just x ->
            let
                dictNodes =
                    Dict.insert nid ( -1, { f = Insat, t = I, u = uMax, ut = [], i = nid2, ps = x } ) nodes

                dictEdges =
                    Dict.insert ( nidp, nid ) (Just { r = I, is = x, br = 1, s = Dict.empty }) edges
            in
            ( dictNodes, dictEdges )


        Nothing ->
            if dcur == dMax then
                let
                    g = Maybe.withDefault Insat <| Maybe.map (\x -> (Tuple.second x).f) <| Dict.get nidp nodes
                in
                ( Dict.insert nid ( 1, { f = g, t = I, u = uMax, ut = [], i = nid2, ps = [] } ) nodes
                , Dict.insert ( nidp, nid ) Nothing edges
                )

            else
                case List.head <| List.filter (\( _, ( _, ti ) ) -> ti.t == DN && ti.u /= 0) <| Dict.toList nodes of
                    Just ( i, ( _, ti ) ) ->
                        let
                            g = Maybe.withDefault Insat <| List.head <| ffolUncuantifiedComponents ti.f
                        in
                        let
                            new_nodes =
                                Dict.insert nid ( 0, { f = g, t = ffolType g, u = uMax, ut = [], i = nid2, ps = [ i ] } )
                                  <| Dict.insert i ( 0, { ti | u = 0 } ) nodes

                            new_edges = Dict.insert ( nidp, nid ) (Just { r = DN, is = [ ti.i ], br = 1, s = Dict.empty }) edges
                        in
                        semanticTableauAux m dMax uMax (nid + 1) (nid2 + 1) nid (dcur + 1) new_nodes new_edges

                    Nothing ->
                        case List.head <| List.filter (\( _, ( _, ti ) ) -> ti.t == A && ti.u /= 0) <| Dict.toList nodes of
                            Just ( i, ( _, ti ) ) ->
                                let
                                    ( g, h ) =
                                        case ffolUncuantifiedComponents ti.f of
                                            [ g1, h1 ] -> ( g1, h1 )

                                            _ -> ( Insat, Insat )
                                in
```

```
---------------------------------------------------------------- INDENTADO ----------------------------------------------------------------
let
    new_nodes =
  Dict.insert (nid + 1) ( 0, { f = h, t = ffolType h, u = uMax, ut = [], i = nid2 + 1, ps = [ i ] } ) <|
      Dict.insert nid ( 0, { f = g, t = ffolType g, u = uMax, ut = [], i = nid2, ps = [ i ] } ) <|
          Dict.insert i ( 0, { ti | u = 0 } ) nodes

    new_edges =
  Dict.insert ( nid, nid + 1 ) (Just { r = A, is = [ ti.i ], br = 2, s = Dict.empty }) <|
      Dict.insert ( nidp, nid ) (Just { r = A, is = [ ti.i ], br = 1, s = Dict.empty }) edges
in
semanticTableauAux m dMax uMax (nid + 2) (nid2 + 2) (nid + 1) (dcur + 1) new_nodes new_edges

Nothing ->
case List.head <| List.filter (\( _, ( _, ti ) ) -> ti.t == B && ti.u /= 0) <| Dict.toList nodes of
    Just ( i, ( _, ti ) ) ->
  let
      ( g, h ) =
          case ffolUncuantifiedComponents ti.f of
              [ g1, h1 ] ->
                  ( g1, h1 )

              _ ->
                  ( Insat, Insat )
  in
  let
      new_nodes1 =
          Dict.insert nid ( 0, { f = g, t = ffolType g, u = uMax, ut = [], i = nid2, ps = [ i ] } ) <|
              Dict.insert i ( 0, { ti | u = 0 } ) nodes

      new_edges1 =
          Dict.insert ( nidp, nid ) (Just { r = B, is = [ ti.i ], br = 1, s = Dict.empty }) edges
  in
  let
      ( nodes1, edges1 ) =
          semanticTableauAux m dMax uMax (nid + 1) (nid2 + 1) nid (dcur + 1) new_nodes1 new_edges1
  in
  let
      new_nid =
          1 + (Maybe.withDefault 0 <| List.maximum <| Dict.keys nodes1)
  in
  let
      new_nodes2 =
          Dict.insert new_nid ( 0, { f = h, t = ffolType h, u = uMax, ut = [], i = nid2, ps = [ i ] } ) <|
              Dict.insert i ( 0, { ti | u = 0 } ) nodes
```

```
        new_edges2 =
            Dict.insert ( nidp, new_nid ) (Just { r = B, is = [ ti.i ], br = 2, s = Dict.empty }) edges
    in
    let
        ( nodes2, edges2 ) =
            semanticTableauAux m dMax uMax (new_nid + 1) (nid2 + 1) new_nid (dcur + 1) new_nodes2 new_edges2
    in
    ( Dict.union nodes1 nodes2, Dict.union edges1 edges2 )

      Nothing ->
    case List.head <| List.filter (\( _, ( _, ti ) ) -> ti.t == D && ti.u /= 0) <| Dict.toList nodes of
        Just ( i, ( _, ti ) ) ->
            let
                c = generateNewConst 1 m

                ( v, g ) =
                    Maybe.withDefault ( ( "", [], 0 ), Insat ) <| ffolCuantifiedComponents ti.f
            in
            let
                new_nodes =
                    Dict.insert nid ( 0, { f = FOL_SS.ffolApplySubstitution (Dict.singleton v c) g, t = ffolType g, u = uMax, ut = [],
                      i = nid2, ps = [ i ] } ) <| Dict.insert i ( 0, { ti | u = 0 } ) nodes

                new_edges =
                    Dict.insert ( nidp, nid ) (Just { r = D, is = [ i ], br = 1, s = Dict.singleton v c }) edges
            in
            semanticTableauAux (m ++ [ c ]) dMax uMax (nid + 1) (nid2 + 1) nid (dcur + 1) new_nodes new_edges

        Nothing ->
            case List.head <| List.sortBy (\( _, ( _, ti ) ) -> uMax - ti.u) <| List.filter (\(_, ( _, ti )) -> ti.t == G && ti.u /= 0)
              <| Dict.toList nodes of
                    Just ( i, ( _, ti ) ) ->
                      let
                            occurs =
                                List.map (\( v, lv ) -> ( v, 1 + List.length lv )) <| LE.gatherEquals <| List.concat
                                  <| List.map (\( _, ti2 ) -> FOL_SS.ffolAllClosedTerms ti2.f)
                                    <| List.filter (\( _, ti2 ) -> ti2.t == L || ti2.t == E) <| Dict.values nodes
                in
                let
                    tsus =
                        Maybe.withDefault (generateNewConst 1 m) <| LE.maximumBy (\x -> Maybe.withDefault 0 <| List.head
                          <| List.map Tuple.second <| List.filter (\( y, _ ) -> y == x) occurs)
                            <| List.filter (\x -> not (List.member x ti.ut)) m

                    ( v, g ) =
                        Maybe.withDefault ( ( "", [], 0 ), Insat ) <| ffolCuantifiedComponents ti.f
                in
```

```
                let
                    new_m =
                        uniqueConcatList m [ tsus ]

                    new_nodes =
                        Dict.insert nid ( 0, { f = FOL_SS.ffolApplySubstitution (Dict.singleton v tsus) g, t = ffolType g, u = uMax,
                          ut = [], i = nid2, ps = [ i ] } ) <| Dict.insert i ( 0, { ti | ut = tsus :: ti.ut, u = ti.u - 1 } ) nodes

                    new_edges =
                        Dict.insert ( nidp, nid ) (Just { r = G, is = [ ti.i ], br = 1, s = Dict.singleton v tsus }) edges
                in
                semanticTableauAux new_m dMax uMax (nid + 1) (nid2 + 1) nid (dcur + 1) new_nodes new_edges

            Nothing ->
                let
                    g =
                        Maybe.withDefault Insat <| Maybe.map (\x -> (Tuple.second x).f) <| Dict.get nidp nodes
                in
                ( Dict.insert nid ( 1, { f = g, t = I, u = uMax, ut = [], i = nid2, ps = [] } ) nodes
                , Dict.insert ( nidp, nid ) Nothing edges
                )

-------------------------------------------------------------------------------------------------------------------------------------


ffolCuantifiedComponents : FormulaFOL -> Maybe ( Variable, FormulaFOL )
ffolCuantifiedComponents f =
    case f of
        Exists v g -> Just ( v, g )

        Neg (Exists v g) -> Just ( v, FOL_SS.ffolNegation g )

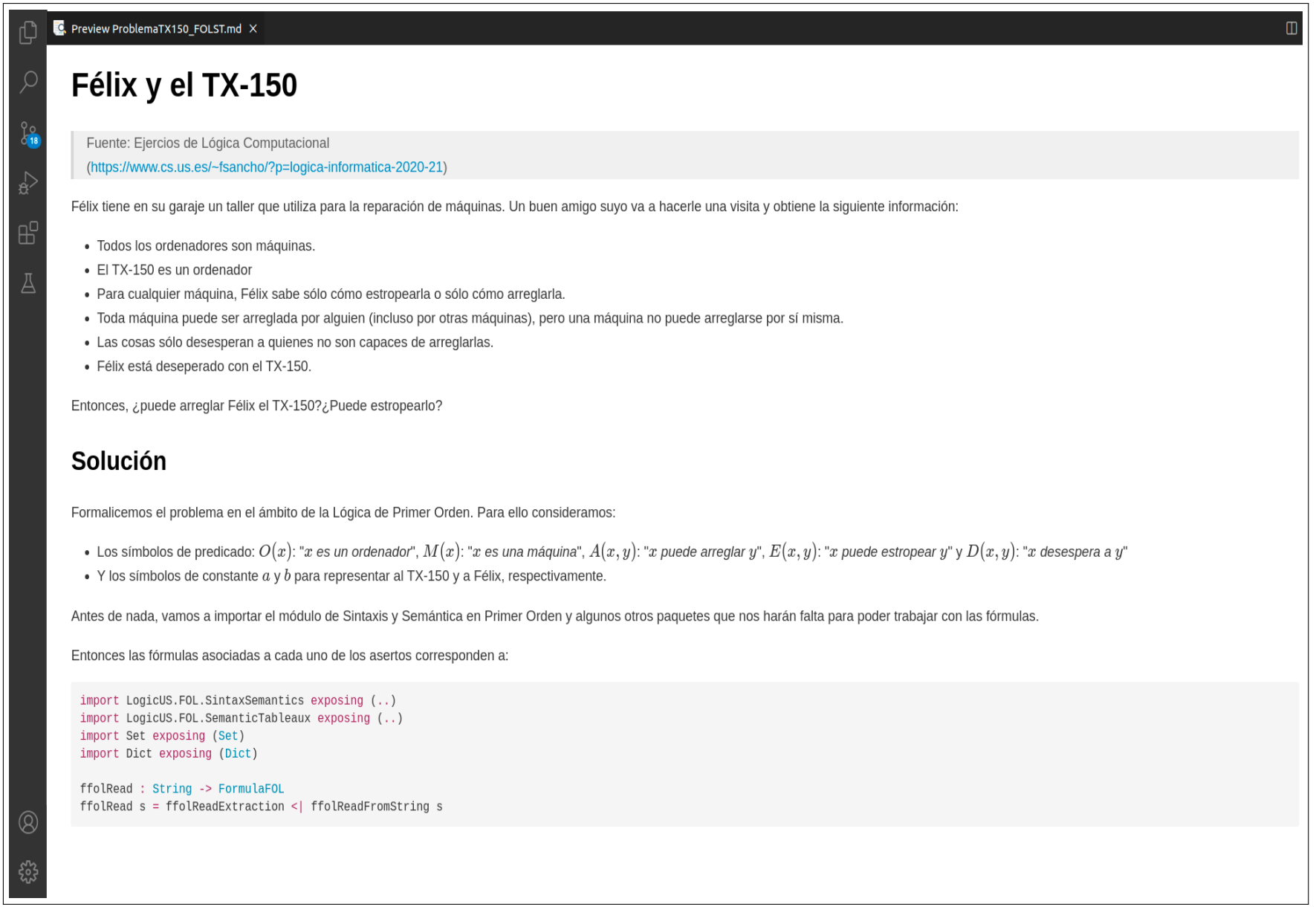
        Forall v g -> Just ( v, g )

        Neg (Forall v g) -> Just ( v, FOL_SS.ffolNegation g )

        _ -> Nothing


generateNewConst : Int -> List Term -> Term
generateNewConst i m =
    let
      c = Func ( "c", [ i ] ) []
    in
    if List.member c m then generateNewConst (i + 1) m else c
```

Código 4.36: Construcción del Tablero Semático FOL en LogicUS


Preview ProblemaTX150_FOLST.md
Félix y el TX-150
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Félix tiene en su garaje un taller que utiliza para la reparación de máquinas. Un buen amigo suyo va a hacerle una visita y obtiene la siguiente información:
Todos los ordenadores son máquinas.
El TX-150 es un ordenador
Para cualquier máquina, Félix sabe sólo cómo estropearla o sólo cómo arreglarla.
Toda máquina puede ser arreglada por alguien (incluso por otras máquinas), pero una máquina no puede arreglarse por sí misma.
Las cosas sólo desesperan a quienes no son capaces de arreglarlas.
Félix está deseperado con el TX-150.
Entonces, ¿puede arreglar Félix el TX-150?¿Puede estropearlo?
Solución
Formalicemos el problema en el ámbito de la Lógica de Primer Orden. Para ello consideramos:
Los símbolos de predicado: $O(x)$: "x es un ordenador", $M(x)$: "x es una máquina", $A(x,y)$: "x puede arreglar y", $E(x,y)$: "x puede estropear y" y $D(x,y)$: "x desespera a y"
Y los símbolos de constante a y b para representar al TX-150 y a Félix, respectivamente.
Antes de nada, vamos a importar el módulo de Sintaxis y Semántica en Primer Orden y algunos otros paquetes que nos harán falta para poder trabajar con las fórmulas.
Entonces las fórmulas asociadas a cada uno de los asertos corresponden a:
import LogicUS.FOL.SintaxSemantics exposing (..)
import LogicUS.FOL.SemanticTableaux exposing (..)
import Set exposing (Set)
import Dict exposing (Dict)
ffolRead : String -> FormulaFOL
ffolRead s = ffolReadExtraction <| ffolReadFromString s

Preview ProblemaTX150_FOLST.md

Apartado 1 Formalicemos los enunciados:

- Todos los ordenadores son máquinas.

```
psi1 : FormulaFOL
psi1 = ffolRead "!A[x](O(x) -> M(x))"
```

$$\forall x\ (O(x) \rightarrow M(x))$$

- El TX-150 es un ordenador

```
psi2 : FormulaFOL
psi2 = ffolRead "O(*a)"
```

$$O(a)$$

- Para cualquier máquina, Félix sabe sólo cómo estropearla o sólo cómo arreglarla.

```
psi3 : FormulaFOL
psi3 = ffolRead "!A[x](M(x) -> (A(*b, x) <-> ¬E(*b,x)))"
```

$$\forall x\ (M(x) \rightarrow (A(b,x) \leftrightarrow \neg E(b,x)))$$

- Toda máquina puede ser arreglada por alguien (incluso por otras máquinas), pero una máquina no puede arreglarse por sí misma.

```
psi4 : FormulaFOL
psi4 = ffolRead "!A[x](M(x) -> (!E[y] A(y, x) & ¬A(x,x)))"
```

$$\forall x\ (M(x) \rightarrow (\exists y\, A(y,x) \wedge \neg A(x,x)))$$

- Las cosas sólo desesperan a quienes no son capaces de arreglarlas.

```
psi5 : FormulaFOL
psi5 = ffolRead "!A[x]!A[y](D(x,y) <-> ¬A(y,x))"
```

$$\forall x \forall y\ (D(x,y) \leftrightarrow \neg A(y,x))$$

- Félix está deseperado con el TX-150.

```
psi6 : FormulaFOL
psi6 = ffolRead "D(*a,*b)"
```

Preview ProblemaTX150_FOLST.md

$$D(a,b)$$

Ahora veámos si es posible que Félix pueda arreglar el TX-150. Esto es si el conjunto de las fórmulas anteriores junto con $A(b,a)$ es consistente. Probémos que no mediante tableros semánticos.

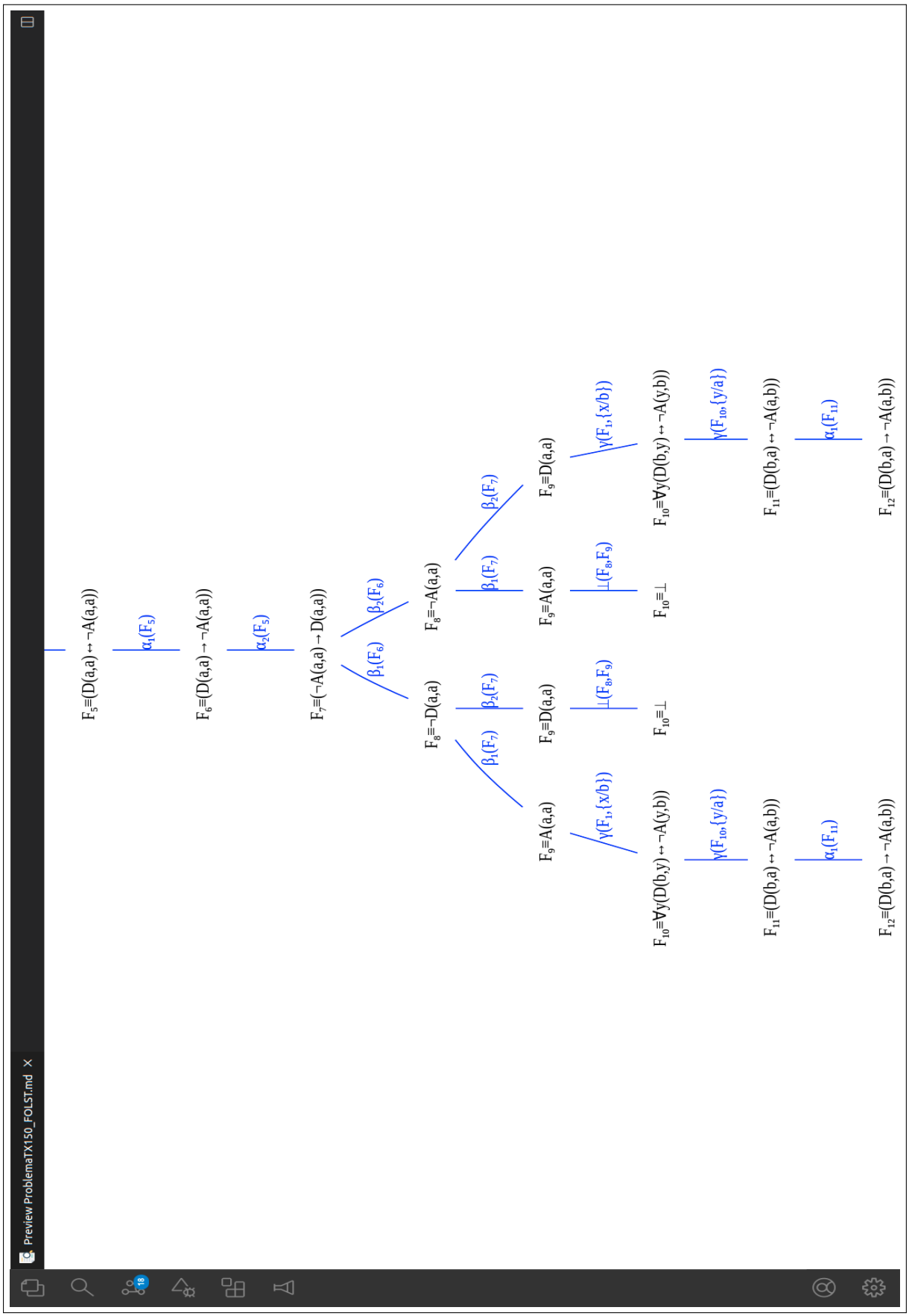
```
psi7 : FormulaFOL
psi7 = ffolRead "A(*b, *a)"

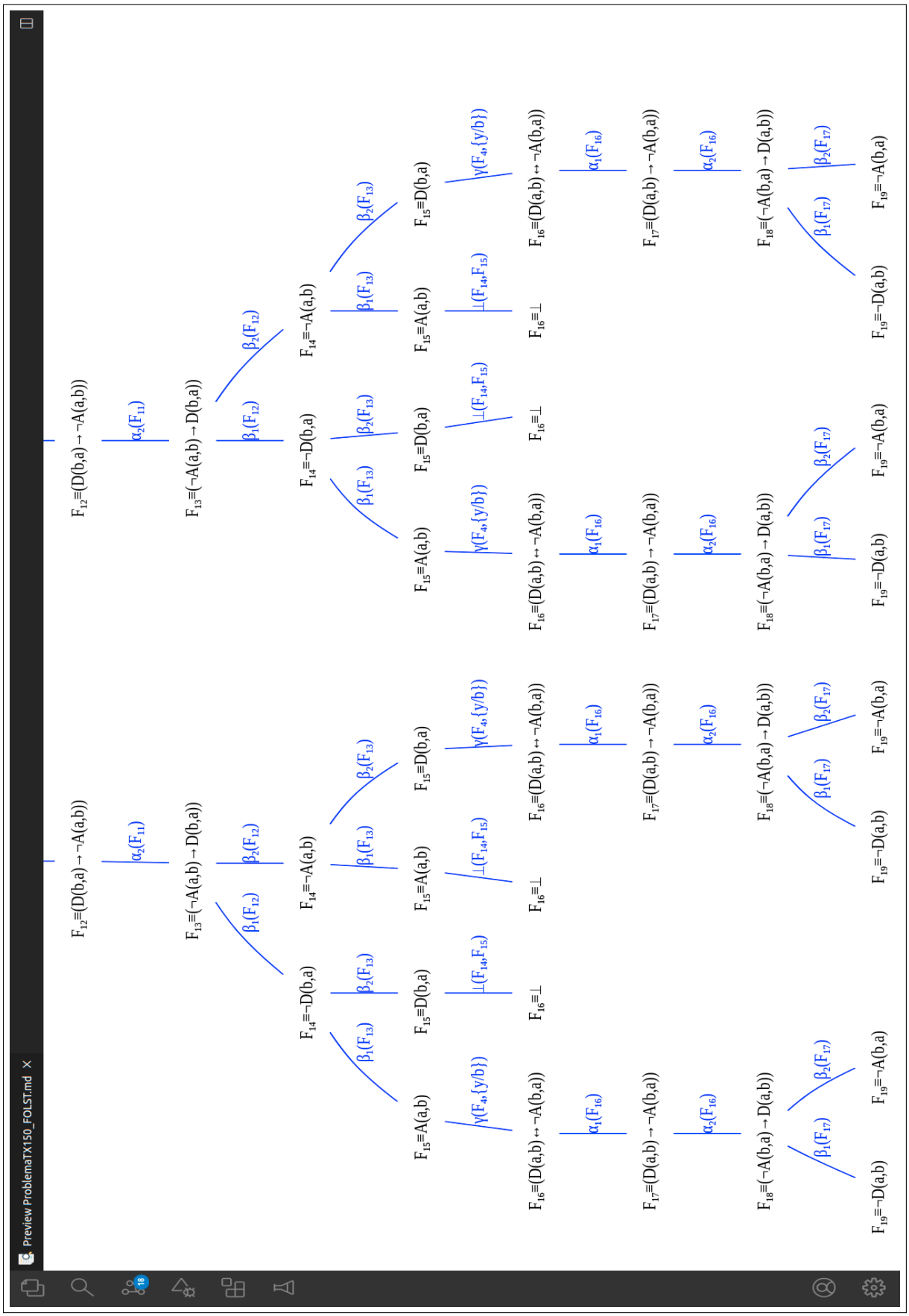
sigma1 : SetFOL
sigma1 = [ psi5, psi6, psi7]
```

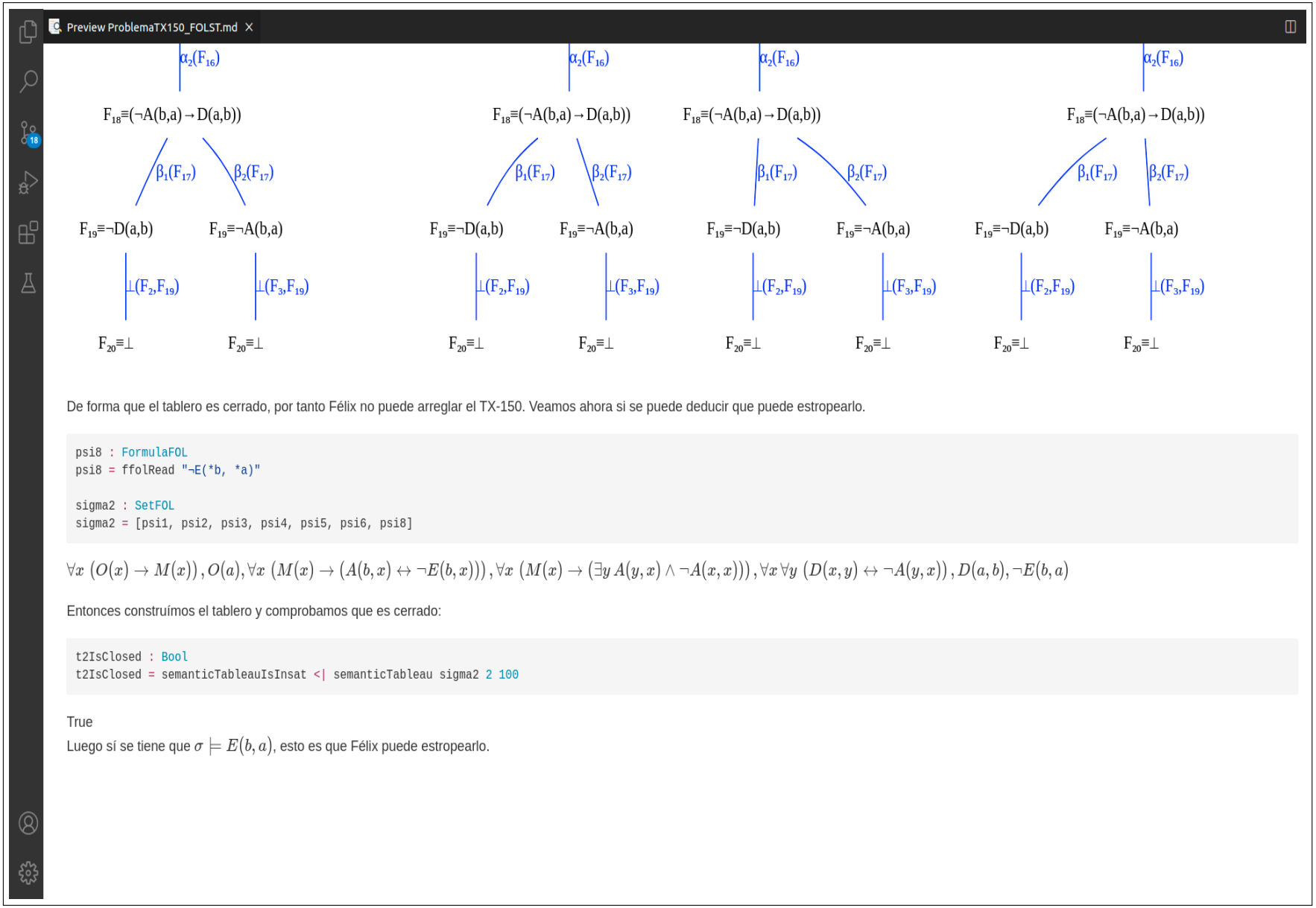
$$\forall x \forall y \left(D(x,y) \leftrightarrow \neg A(y,x)\right), D(a,b), A(b,a)$$

Entonces construímos el tablero. Nótese que en realidad, las únicas fórmulas que son interesantes son las últimas tres.:

```
t1 : FOLSemanticTableau
t1 = semanticTableau sigma1 2 30
```


Preview ProblemaTX150_FOLST.md
F5≡(D(a,a)↔¬A(a,a))
α1(F5)
F6≡(D(a,a)→¬A(a,a))
α2(F5)
F7≡(¬A(a,a)→D(a,a))
β1(F6)
β2(F6)
F8≡¬D(a,a)
F8≡¬A(a,a)
β1(F7)
β2(F7)
β1(F7)
β2(F7)
F9≡A(a,a)
F9≡D(a,a)
F9≡A(a,a)
F9≡D(a,a)
γ(F1,{x/b})
⊥(F8,F9)
⊥(F8,F9)
γ(F1,{x/b})
F10≡∀y(D(b,y)↔¬A(y,b))
F10≡⊥
F10≡⊥
F10≡∀y(D(b,y)↔¬A(y,b))
γ(F10,{y/a})
γ(F10,{y/a})
F11≡(D(b,a)↔¬A(a,b))
F11≡(D(b,a)↔¬A(a,b))
α1(F11)
α1(F11)
F12≡(D(b,a)→¬A(a,b))
F12≡(D(b,a)→¬A(a,b))


Preview ProblemaTX150_FOLST.md
F12≡(D(b,a)→¬A(a,b))
α2(F11)
F13≡(¬A(a,b)→D(b,a))
β2(F12)
β1(F12)
F14≡¬D(b,a)
F14≡¬A(a,b)
β1(F13)
β2(F13)
F15≡A(a,b)
F15≡D(b,a)
γ(F4,{y/b})
⊥(F14,F15)
F16≡(D(a,b)↔¬A(b,a))
F16≡⊥
α1(F16)
α2(F16)
F17≡(D(a,b)→¬A(b,a))
F18≡(¬A(b,a)→D(a,b))
β1(F17)
β2(F17)
F19≡¬D(a,b)
F19≡¬A(b,a)



De forma que el tablero es cerrado, por tanto Félix no puede arreglar el TX-150. Veamos ahora si se puede deducir que puede estropearlo.

```
psi8 : FormulaFOL
psi8 = ffolRead "¬E(*b, *a)"

sigma2 : SetFOL
sigma2 = [psi1, psi2, psi3, psi4, psi5, psi6, psi8]
```

$$\forall x\,(O(x) \to M(x)), O(a), \forall x\,(M(x) \to (A(b,x) \leftrightarrow \neg E(b,x))), \forall x\,(M(x) \to (\exists y\,A(y,x) \wedge \neg A(x,x))), \forall x\,\forall y\,(D(x,y) \leftrightarrow \neg A(y,x)), D(a,b), \neg E(b,a)$$

Entonces construímos el tablero y comprobamos que es cerrado:

```
t2IsClosed : Bool
t2IsClosed = semanticTableauIsInsat <| semanticTableau sigma2 2 100
```

True

Luego sí se tiene que $\sigma \models E(b,a)$, esto es que Félix puede estropearlo.

Figura 4.12: Ejemplo de uso de LogicUS.FOL.SemanticTableaux
Fuente propia. Creada con litvis

4.2.14. LogicUS.FOL.NormalForms

En este punto presentaremos la implementación de la transformación de fórmulas genéricas en fórmulas equivalentes y/o equiconsistentes en las formas normales: *Prenex*, *Skolem*, *NNF*, *CNF* y DNF. Trataremos en detalle la implementación de las dos primeras, ya que la implementación del resto es análoga a lo planteado para LP a partir de la forma de Skolem.

Recuérdese que el paso a forma Prenex requería de la extracción de los cuantificadores del cuerpo de la fórmula a la cabeza, tratando de priorizar la extracción de los existenciales antes de la de los universales (debido a lo ya comentado en el apartado teórico, acerca de la aridad de las funciones de Skolem dependiendo del orden de extracción). Dicho proceso se realiza a través del árbol de formación de la fórmula (en realidad la propia definición recursiva de la fórmula constituye el propio árbol). En consecuencia, los pasos que se siguen para la extracción según el apartado teórico son:

- Eliminación de equivalencias, sustituyéndolas por la conjunción de las implicaciones.
- Interiorización de las negaciones, aplicando las Leyes de De Morgan y las leyes de negación de cuantificadores $(\neg(\forall/\exists)xF(x) \equiv (\exists/\forall)x\neg F(x))$
- Extracción recursiva de los operadores. De forma que el esquema básico se reduce a recorrer el árbol desde las hojas hasta la raíz exteriorizando en cada paso los cuantificadores. Además, el proceso es almacenado en forma de grafo que puede ser representado gráficamente para la comprensión del método.

El *código 4.37* se muestra la implementación de dicho método.

```
ffolToPrenex : FormulaFOL -> FormulaFOL
ffolToPrenex f =
    let
        f1 =
            ffolRemoveAllEquiv f
    in
    let
        f2 =
            FOL_SS.renameVars f1
    in
    let
        f3 =
            ffolInteriorizeNeg f2
    in
    let
        ret1 =
            ffolToPrenexAux 3 f3
    in
    applyCuantifiers ret1.cuants ret1.f



ffolToPrenex2 :
  FormulaFOL -> (List Cuantifier, FormulaFOL, Graph FormulaFOL (Bool, List Cuantifier ))
ffolToPrenex2 f =
    let
        f1 =
            ffolRemoveAllEquiv f
    in
    let
        f2 =
            FOL_SS.renameVars f1
    in
    let
        f3 =
            ffolInteriorizeNeg f2
    in
    let
        ret1 =
            ffolToPrenexAux 2 f3
    in
```

```
    let
        nodes_res =
            Node 1 f :: ret1.nodes

        edges_res =
            Edge 1 2 ( False, [] ) :: ret1.edges
    in
    ( ret1.cuants, ret1.f, Graph.fromNodesAndEdges nodes_res edges_res )


ffolToPrenexAux : Int -> FormulaFOL -> PrenexCalcResult
ffolToPrenexAux nid f =
    case f of
        Conj g h ->
            let
                ret1 =
                    ffolToPrenexAux (nid + 1) g
            in
            let
                max_nid1 =
                    nid + List.length ret1.nodes
            in
            let
                ret2 =
                    ffolToPrenexAux (max_nid1 + 1) h
            in
            let
                max_nid2 =
                    max_nid1 + List.length ret2.nodes
            in
            let
                cuants =
                    sortWithFirstE ret1.cuants ret2.cuants

                openFormula =
                    Conj ret1.f ret2.f
            in
            let
                nodes =
                    [ Node nid f,
            Node (max_nid2 + 1) (applyCuantifiers cuants <| openFormula)
          ] ++ ret1.nodes ++ ret2.nodes

                edges =
                    [ Edge nid (nid + 1) ( False, [] )
                    , Edge nid (max_nid1 + 1) ( False, [] )
                    , Edge max_nid1 (max_nid2 + 1) ( True, ret1.cuants )
                    , Edge max_nid2 (max_nid2 + 1) ( False, ret2.cuants )
                    ]
                        ++ ret1.edges
                        ++ ret2.edges
            in
            { nodes = nodes, edges = edges, cuants = cuants, f = openFormula }

        Disj g h ->
            let
                ret1 =
                    ffolToPrenexAux (nid + 1) g
            in
            let
                max_nid1 =
                    nid + List.length ret1.nodes
            in
            let
                ret2 =
                    ffolToPrenexAux (max_nid1 + 1) h
            in
            let
                max_nid2 =
                    max_nid1 + List.length ret2.nodes
            in
            let
```

```
                cuants =
                    sortWithFirstE ret1.cuants ret2.cuants

                openFormula =
                    Disj ret1.f ret2.f
            in
            let
                nodes =
                    [ Node nid f,
            Node (max_nid2 + 1) (applyCuantifiers cuants <| openFormula)
          ] ++ ret1.nodes ++ ret2.nodes

                edges =
                    [ Edge nid (nid + 1) ( False, [] )
                    , Edge nid (max_nid1 + 1) ( False, [] )
                    , Edge max_nid1 (max_nid2 + 1) ( True, ret1.cuants )
                    , Edge max_nid2 (max_nid2 + 1) ( False, ret2.cuants )
                    ]
                        ++ ret1.edges
                        ++ ret2.edges
            in
            { nodes = nodes, edges = edges, cuants = cuants, f = openFormula }

        Impl g h ->
            let
                ret1 =
                    ffolToPrenexAux (nid + 1) g
            in
            let
                max_nid1 =
                    nid + List.length ret1.nodes
            in
            let
                ret2 =
                    ffolToPrenexAux (max_nid1 + 1) h
            in
            let
                max_nid2 =
                    max_nid1 + List.length ret2.nodes
            in
            let
                cuants =
                    sortWithFirstE (List.map contraryCuantifier ret1.cuants) ret2.cuants

                openFormula =
                    Impl ret1.f ret2.f
            in
            let
                nodes =
                    [ Node nid f,
            Node (max_nid2 + 1) (applyCuantifiers cuants <| openFormula)
          ] ++ ret1.nodes ++ ret2.nodes

                edges =
                    [ Edge nid (nid + 1) ( False, [] )
                    , Edge nid (max_nid1 + 1) ( False, [] )
                    , Edge max_nid1 (max_nid2 + 1)
            ( True, List.map contraryCuantifier ret1.cuants )
                    , Edge max_nid2 (max_nid2 + 1) ( False, ret2.cuants )
                    ]
                        ++ ret1.edges
                        ++ ret2.edges
            in
            { nodes = nodes, edges = edges, cuants = cuants, f = openFormula }

        g ->
            let
                ( cuants1, h ) =
                    extractHeaderCuantifiers g
            in
            if FOL_SS.ffolIsOpen h then
                { nodes = [ Node nid f ], edges = [], cuants = cuants1, f = h }
```

```
            else
                let
                    ret1 =
                        ffolToPrenexAux (nid + 1) h
                in
                let
                    max_nid1 =
                        nid + List.length ret1.nodes

                    cuants =
                        cuants1 ++ ret1.cuants
                in
                let
                    nodes =
                        [ Node nid f,
              Node (max_nid1 + 1) (applyCuantifiers cuants <| ret1.f)
            ] ++ ret1.nodes

                    edges =
                        [ Edge nid (nid + 1) ( False, [] ),
              Edge max_nid1 (max_nid1 + 1) ( False, [] )
            ] ++ ret1.edges
                in
                { nodes = nodes, edges = edges, cuants = cuants, f = ret1.f }
```

Código 4.37: Extracción de la forma Prenex en LogicUS

A partir de la implementación realizada del cálculo de la forma Prenex podemos implementar la Skolemización de las fórmulas. Para ello, nos serviremos de dos funciones: la función de cálculo de sustituciones de Skolem, y la función principal. La primera se trata de una función definida por plegado a partir de la lista de cuantificadores (que son extraídos de la cabeza de la fórmula en forma prenex) de forma que para cada existencial se toma la sustitución por la función de Skolem notada por ş (para evitar colisiones con otras definidas por el usuario) de aridad el número de universales procesados previamente. La función principal, por su parte, aplica la sustitución obtenida del procedimiento anterior.

```
getSkolemSubs : List Cuantifier -> Substitution
getSkolemSubs cS =
    let
        ( subs, _, _ ) =
            List.foldl
                (\c ( res, lF, nE ) ->
                    case c of
                        A x -> ( res, lF ++ [ Var x ], nE )

                        E x ->
                            let
                                nres = Dict.insert x (Func ( "ş", [ nE + 1 ] ) lF) res
                            in
                            ( nres, lF, nE + 1 )
                )
                ( Dict.empty, [], 0 )
                cS
    in
    subs

ffolToSkolem : FormulaFOL -> FormulaFOL
ffolToSkolem f =
    if ffolIsPrenex f then
        let ( lc, g ) =  extractHeaderCuantifiers f in
        FOL_SS.ffolApplySubstitution (getSkolemSubs lc) g

    else
        let ( cs, fo, _ ) = ffolToPrenex2 f in
        FOL_SS.ffolApplySubstitution (getSkolemSubs cs) fo
```

Código 4.38: Extracción de la forma Skolem en LogicUS

Téngase en cuenta que la forma de Skolem proporciona una fórmula abierta, de forma que para el cálculo de las formas normales (*NNF*,*CNF* y *DNF*) basta aplicar la eliminación de implicaciones y la interiorización de negaciones, conjunciones y disyunciones, implementadas de forma análoga a lo mostrado para LP, para obtenerlas.

En este caso mostraremos el caso de ejemplo de uso junto con el siguiente módulo que abordaremos, en relación a la forma clausal.

4.2.15. LogicUS.FOL.Clauses

Si bien en el punto anterior introdujimos las formas normales y presentamos el cálculo de la FNC, base del cálculo de cláusulas, no incluimos en el mismo la presentación de las mismas ya que no forman parte de dicho módulo, ya que la representación adoptada para ellas es independiente de los literales que corresponden a fórmulas. Vamos a presentar, análogamente a como lo hicimos para LP, los detalles de su implementación.

Al contrario que en LP, en el que existía un único elemento (las variables proposicionales) que pueden corresponder a átomos, y aunque en la Lógica de Primer Orden podríamos tomar un solo tipo de átomos (correspondiente a los predicados), se ha hecho una diferenciación en dos tipos de átomos, uno correspondiente a los predicados y otro para el predicado de igualdad.

Así, los literales de las cláusulas corresponderán a tuplas de un átomo y un valor booleano que indica el signo del literal (`True` para el positivo y `False` para el negativo). Así mismo, las cláusulas estarán formadas como listas de literales, aunque en realidad corresponderían a conjuntos. Aunque Elm no permite definir un tipo como comparable, sí definiremos una función de comparación para la ordenación de los literales en las cláusulas, priorizando los positivos, la igualdad posee máximo valor alfabético.

Al igual que en LP, se han definido una serie de operaciones básicas con cláusulas (ordenación, unión, subsunción, sustitución, definida válida, definida positiva, definida negativa, eliminación de cláusulas subsumidas y tautológicas en conjuntos/listas de cláusulas) y también otras, como función de lectura desde cadena (con formato análogo al de LP), funciones de transformación de fórmulas a conjuntos clausales, funciones de representación, y funciones de transformación de átomos y literales a funciones proposicionales tomando los predicados junto con los términos como variables proposicionales, especialmente interesantes cuando tratemos, a continuación, con los trabajos de Herbrand.

4.2.16. LogicUS.FOL.Herbrand

En este módulo se han implementado algunos de los trabajos de Herbrand en el ámbito de la Lógica de Primer Orden. Este módulo establece una conexión directa entre la Lógica Proposicional y la Lógica de Primer.

Dentro de este módulo, denotamos por *Signatura* de una fórmula cerrada (o un conjunto de fórmulas cerradas) a una terna que contiene un conjunto de los símbolos de constantes, una asignación de cada símbolo de función con su correspondiente aridad y una asignación de cada símbolo de predicado con su correspondiente aridad, que toman parte en la fórmula (o conjunto). Nótese que un mismo símbolo de función y/o predicado no puede aplicarse para aridades distintas.

Extendemos la misma definición para términos (de forma que la última asignación correponderá a la aplicación vacía, diccionario vacío). El *código 4.39* se muestra la definición de la estructura (tipo) y de las funciones de cálculo de las mismas.

```
type alias Signature =
    ( Set (String, List Int), Dict (String, List Int) Int, Dict (String, List Int) Int )


union2Signatures : Signature -> Signature -> Signature
union2Signatures ( cs1, fs1, ps1 ) ( cs2, fs2, ps2 ) =
    let
        cs =
            Set.union cs1 cs2

        fs =
            Dict.union fs1 fs2

        ps =
            Dict.union ps1 ps2
    in
    ( cs, fs, ps )


signatureTerm : Term -> Signature
signatureTerm t =
    case t of
        Var _ ->
            ( Set.empty, Dict.empty, Dict.empty )

        Func f [] ->
            ( Set.singleton f, Dict.empty, Dict.empty )

        Func f terms ->
            List.foldl
        (\x ac -> union2Signatures ac (signatureTerm x))
        ( Set.empty, Dict.singleton f (List.length terms), Dict.empty )
        terms


ffolSignature : FormulaFOL -> Maybe Signature
ffolSignature f =
    if FOL_SS.ffolIsOpen f then
        Just <| ffolSignatureAux f

    else
        Nothing


ffolSignatureAux : FormulaFOL -> Signature
ffolSignatureAux f =
    case f of
        FOL_SS.Pred p terms ->
            List.foldl
        (\x ac -> union2Signatures ac (signatureTerm x))
        ( Set.empty, Dict.empty, Dict.singleton p (List.length terms) )
        terms

        FOL_SS.Equal t1 t2 ->
            List.foldl
        (\x ac -> union2Signatures ac (signatureTerm x))
        ( Set.empty, Dict.empty, Dict.singleton ( "=", [] ) 2 )
        [ t1, t2 ]

        FOL_SS.Neg x ->
            ffolSignatureAux x

        FOL_SS.Conj x y ->
            union2Signatures (ffolSignatureAux x) (ffolSignatureAux y)

        FOL_SS.Disj x y ->
            union2Signatures (ffolSignatureAux x) (ffolSignatureAux y)

        FOL_SS.Impl x y ->
            union2Signatures (ffolSignatureAux x) (ffolSignatureAux y)
```

```
        FOL_SS.Equi x y ->
            union2Signatures (ffolSignatureAux x) (ffolSignatureAux y)

        FOL_SS.Forall _ x ->
            ffolSignatureAux x

        FOL_SS.Exists _ x ->
            ffolSignatureAux x

        FOL_SS.Insat ->
            ( Set.empty, Dict.empty, Dict.empty )

        FOL_SS.Taut ->
            ( Set.empty, Dict.empty, Dict.empty )


sfolSignature : SetFOL -> Maybe Signature
sfolSignature ls =
    if List.all FOL_SS.ffolIsOpen ls then
        Just <|
      List.foldl
        (\f ac -> union2Signatures ac (ffolSignatureAux f))
        ( Set.empty, Dict.empty, Dict.empty )
        ls

    else
        Nothing
```

Código 4.39: Definición y cálculo de Signaturas en LogicUS

A partir de las signaturas se definen varios elementos de los trabajos de Herbrand:

- *Universo de Herbrand*: Corresponde a todos los términos cerrados que se pueden formar a partir de las constantes (introduciendo una única constante si no existiese ninguna en la signatura) y los símbolos de función. Nótese que un signo de función es suficiente para que el universo sea infinito $\{a, f(a), f(f(a)), \dots\}$. Por ello computacionalmente, definimos el universo de Herbrand de orden n como el universo de Herbrand acotado a funciones anidadas, a lo sumo n veces.
- *Base de Herbrand*: Corresponde al conjunto de todos los átomos del lenguaje asociado a un conjunto de fórmulas que resultan de considerar cada uno de los símbolos de predicado, de aridad k, sobre todas las posibles k-tuplas formadas por elementos del universo de Herbrand. Al igual que en el caso anterior, se define la base de Herbrand de orden n como el conjunto de los posibles átomos formados a partir de las k-tuplas de elementos del universo de Herbrand de orden n.
- *Interpretaciones de Herbrand*: Corresponde al conjunto de todas las posibles interpretaciones para el conjunto de las fórmulas de la base de Herbrand. Se define el conjunto de interpretaciones de Herbrand de orden n como el conjunto de todas las posibles interpretaciones para la base de Herbrand de orden n. Nótese que dada la definición, proposicional, de las interpretaciones, estas corresponden a todos los posibles subconjuntos del universo de Herbrand (de orden n).
- *Modelos de Herbrand*: Corresponde al conjunto de todas las interpretaciones de Herbrand que hacen verdadero el conjunto. Para realizar la valoración se realiza de forma análoga a como se hacía en la Lógica Proposicional, extendiendo la definición para los cuantificadores, mediante la conjunción/disyunción de todos los elementos del universo.
- *Extensión de Herbrand*: Corresponde al conjunto de todas las posibles fórmulas que resultan de realizar todas las posibles sustituciones sobre las fórmulas de un conjunto (de fórmulas abiertas), considerando para cada variable todos los posibles elementos del universo de Herbrand. Por las razones expuestas, se acota a orden n. Sobre ella se pueden aplicar algoritmos de reducción de la satisfactibilidad proposicional.

```
ffolHerbrandUniverse : FormulaFOL -> Int -> Maybe (List Term)
ffolHerbrandUniverse f n =
    if FOL_SS.ffolIsOpen f then
        Just <| ffolHerbrandUniverseAux f n

    else
        Nothing


ffolHerbrandUniverseAux : FormulaFOL -> Int -> List Term
ffolHerbrandUniverseAux f n =
    signatureHerbrandUniverse (ffolSignatureAux f) n


sfolHerbrandUniverse : List FormulaFOL -> Int -> Maybe (List Term)
sfolHerbrandUniverse fs n =
    if List.all FOL_SS.ffolIsOpen fs then
        Maybe.map (\xs -> signatureHerbrandUniverse xs n) (sfolSignature fs)

    else
        Nothing




signatureHerbrandBase : Signature -> Int -> List FormulaFOL
signatureHerbrandBase ( cs, fs, ps ) n =
    let
        uH =
            signatureHerbrandUniverse ( cs, fs, ps ) n
    in
    List.foldl
    (\x ac -> uniqueConcatList ac x)
    []
    <| List.map
      (\( p, a ) ->
        List.map (\ts -> Pred p ts) (List.Extra.cartesianProduct (List.repeat a uH)))
      (Dict.toList ps)



ffolHerbrandBase : FormulaFOL -> Int -> Maybe (List FormulaFOL)
ffolHerbrandBase f n =
    if FOL_SS.ffolIsOpen f then
        Just <| ffolHerbrandBaseAux f n

    else
        Nothing


ffolHerbrandBaseAux : FormulaFOL -> Int -> List FormulaFOL
ffolHerbrandBaseAux f n =
    let
        s =
            ffolSignatureAux f
    in
    signatureHerbrandBase s n



sfolHerbrandBase : SetFOL -> Int -> Maybe (List FormulaFOL)
sfolHerbrandBase fs n =
    if List.all FOL_SS.ffolIsOpen fs then
        Nothing

    else
        sfolHerbrandBaseAux fs n


sfolHerbrandBaseAux : SetFOL -> Int -> Maybe (List FormulaFOL)
sfolHerbrandBaseAux fs n =
```

```
    case fs of
        [] ->
            Nothing

        x :: xs ->
            ffolHerbrandBase
        (FOL_NF.ffolToSkolem <|
          List.foldl
            (\f ac -> FOL_SS.Conj ac (FOL_SS.ffolUniversalClausure f))
            (FOL_SS.ffolUniversalClausure x)
            xs
        )
        n



signatureHerbrandInterpretations : Signature -> Int -> List (List FormulaFOL)
signatureHerbrandInterpretations s n =
    powerset <| signatureHerbrandBase s n



ffolHerbrandInterpretations : FormulaFOL -> Int -> Maybe (List (List FormulaFOL))
ffolHerbrandInterpretations f n =
    if FOL_SS.ffolIsOpen f then
        Just <| ffolHerbrandInterpretationsAux f n

    else
        Nothing


ffolHerbrandInterpretationsAux : FormulaFOL -> Int -> List (List FormulaFOL)
ffolHerbrandInterpretationsAux f n =
    let
        s =
            ffolSignatureAux f
    in
    signatureHerbrandInterpretations s n



sfolHerbrandInterpretations : SetFOL -> Int -> Maybe (List (List FormulaFOL))
sfolHerbrandInterpretations fs n =
    if List.all FOL_SS.ffolIsOpen fs then
        Just <| sfolHerbrandInterpretationsAux fs n

    else
        Nothing


sfolHerbrandInterpretationsAux : SetFOL -> Int -> List (List FormulaFOL)
sfolHerbrandInterpretationsAux fs n =
    let
        s =
            Maybe.withDefault ( Set.empty, Dict.empty, Dict.empty ) <| sfolSignature fs
    in
    signatureHerbrandInterpretations s n



ffolInterpretsHerbrand : FormulaFOL -> List FormulaFOL -> List Term -> Maybe Bool
ffolInterpretsHerbrand f iH uH =
    if FOL_SS.ffolIsOpen f then
        Just <|
      ffolInterpretsHerbrandAux
        (List.foldr (\x ac -> FOL_SS.Forall x ac) f (FOL_SS.ffolVarSymbols f))
        iH
        uH

    else
        Nothing
```

```
ffolInterpretsHerbrandAux : FormulaFOL -> List FormulaFOL -> List Term -> Bool
ffolInterpretsHerbrandAux f iH uH =
    case f of
        FOL_SS.Pred _ _ ->
            List.member f iH

        FOL_SS.Equal _ _ ->
            List.member f iH

        FOL_SS.Neg f1 ->
            not (ffolInterpretsHerbrandAux f1 iH uH)

        FOL_SS.Conj f1 f2 ->
            let
                if1 =
                    ffolInterpretsHerbrandAux f1 iH uH

                if2 =
                    ffolInterpretsHerbrandAux f2 iH uH
            in
            if1 && if2

        FOL_SS.Disj f1 f2 ->
            let
                if1 =
                    ffolInterpretsHerbrandAux f1 iH uH

                if2 =
                    ffolInterpretsHerbrandAux f2 iH uH
            in
            if1 || if2

        FOL_SS.Impl f1 f2 ->
            let
                if1 =
                    ffolInterpretsHerbrandAux f1 iH uH

                if2 =
                    ffolInterpretsHerbrandAux f2 iH uH
            in
            not if1 || if2

        FOL_SS.Equi f1 f2 ->
            let
                if1 =
                    ffolInterpretsHerbrandAux f1 iH uH

                if2 =
                    ffolInterpretsHerbrandAux f2 iH uH
            in
            if1 == if2

        FOL_SS.Forall v f1 ->
            List.all (\x -> x) <|
        List.map
          (\o ->
            ffolInterpretsHerbrandAux
              (FOL_SS.ffolApplySubstitution (Dict.singleton v o) f1)
              iH
              uH
          )
          uH

        FOL_SS.Taut ->
            True

        _ ->
            False
```

```
sfolInterpretsHerbrand : SetFOL -> List FormulaFOL -> List Term -> Maybe Bool
sfolInterpretsHerbrand fs iH uH =
    case fs of
        [] ->
            Nothing

        x :: xs ->
            if List.all FOL_SS.ffolIsOpen fs then
                ffolInterpretsHerbrand (List.foldl FOL_SS.Conj x xs) iH uH

            else
                Nothing



ffolHerbrandModels : FormulaFOL -> Int -> Maybe ( List Term, List (List FormulaFOL) )
ffolHerbrandModels f n =
    if FOL_SS.ffolIsOpen f then
        Just <| ffolHerbrandModelsAux f n

    else
        Nothing


ffolHerbrandModelsAux : FormulaFOL -> Int -> ( List Term, List (List FormulaFOL) )
ffolHerbrandModelsAux f n =
    let
        uH =
            ffolHerbrandUniverseAux f n
    in
    let
        iHs =
            List.filter
        (\iH -> Maybe.withDefault False <| ffolInterpretsHerbrand f iH uH)
        (ffolHerbrandInterpretationsAux f n)
    in
    ( uH, iHs )



sfolHerbrandModels : SetFOL -> Int -> Maybe ( List Term, List (List FormulaFOL) )
sfolHerbrandModels fs n =
    case fs of
        [] ->
            Nothing

        x :: xs ->
            if List.all FOL_SS.ffolIsOpen fs then
                Just <| ffolHerbrandModelsAux (List.foldl FOL_SS.Conj x xs) n

            else
                Nothing



ffolHerbrandExtension : FormulaFOL -> Int -> Maybe (List FormulaPL)
ffolHerbrandExtension f n =
    if FOL_SS.ffolIsOpen f then
        Just <| ffolHerbrandExtensionAux f n

    else
        Nothing


ffolHerbrandExtensionAux : FormulaFOL -> Int -> List FormulaPL
ffolHerbrandExtensionAux f n =
    let
        uH =
            ffolHerbrandUniverseAux f n

        vs =
            FOL_SS.ffolVarSymbols f
```

```
    in
    let
        substitutions =
            List.map (
        \x -> List.map2 (\y z -> ( y, z )) vs x
      )
      (cartesianProduct <| List.repeat (List.length vs) uH)
    in
    List.map
    (\xs ->
      Maybe.withDefault PL_SS.Insat
        <| ffolTofpl
        <| FOL_SS.ffolApplySubstitution (Dict.fromList xs) f
    )
    substitutions



sfolHerbrandExtension : SetFOL -> Int -> Maybe (List FormulaPL)
sfolHerbrandExtension fs n =
    if List.all FOL_SS.ffolIsOpen fs then
        Just <|
      List.foldl (\x ac -> uniqueConcatList ac (ffolHerbrandExtensionAux x n)) [] fs

    else
        Nothing


ffolTofpl : FormulaFOL -> Maybe FormulaPL
ffolTofpl f =
    case f of
        FOL_SS.Pred _ _ ->
            Just <| PL_SS.Atom ( FOL_SS.ffolToString f, [] )

        FOL_SS.Equal _ _ ->
            Just <| PL_SS.Atom ( FOL_SS.ffolToString f, [] )

        FOL_SS.Neg x ->
            Maybe.map PL_SS.Neg <| ffolTofpl x

        FOL_SS.Conj x y ->
            Maybe.map2 PL_SS.Conj (ffolTofpl x) (ffolTofpl y)

        FOL_SS.Disj x y ->
            Maybe.map2 PL_SS.Disj (ffolTofpl x) (ffolTofpl y)

        FOL_SS.Impl x y ->
            Maybe.map2 PL_SS.Impl (ffolTofpl x) (ffolTofpl y)

        FOL_SS.Equi x y ->
            Maybe.map2 PL_SS.Equi (ffolTofpl x) (ffolTofpl y)

        FOL_SS.Forall _ _ ->
            Nothing

        FOL_SS.Exists _ _ ->
            Nothing

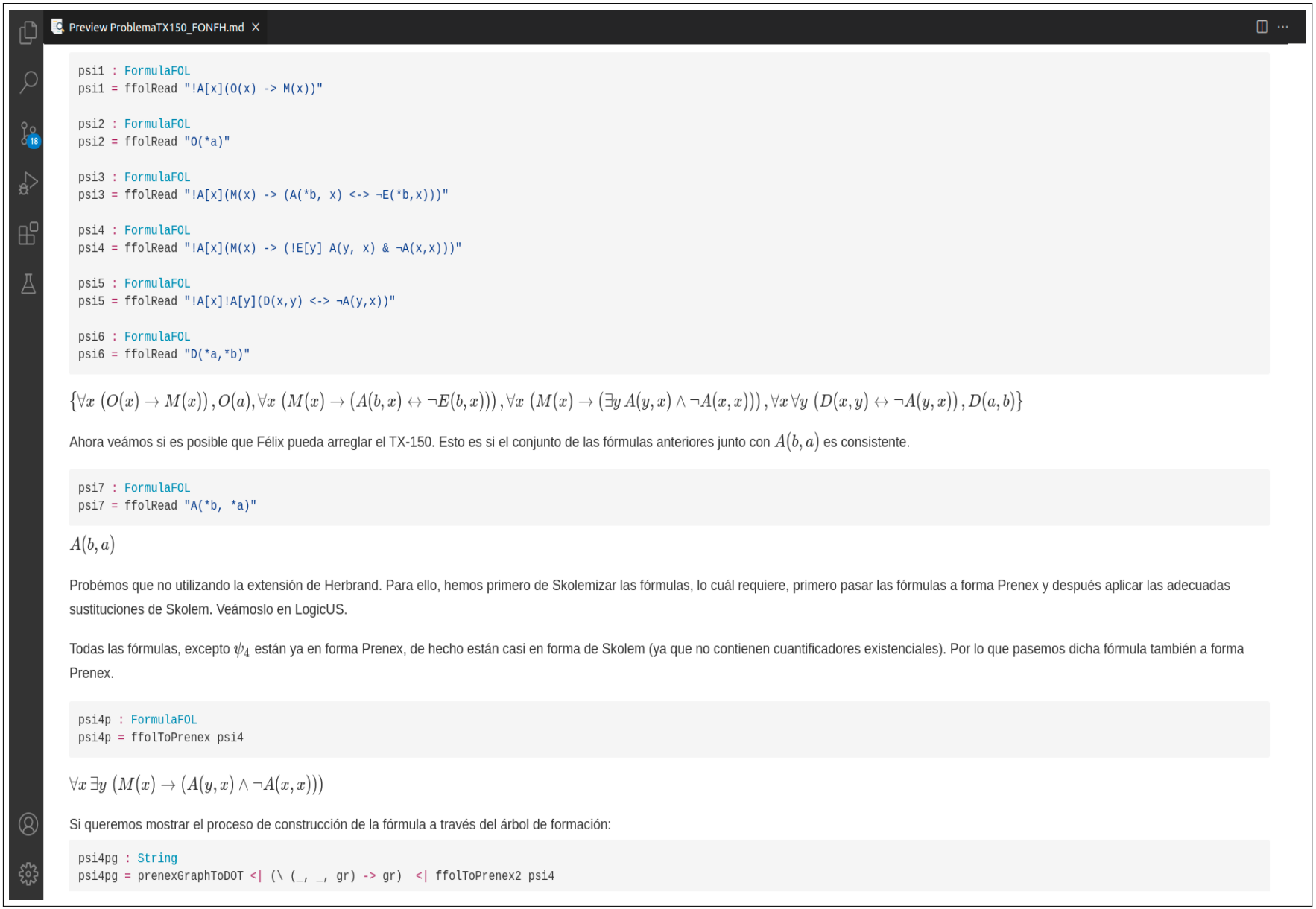
        FOL_SS.Insat ->
            Just PL_SS.Insat

        FOL_SS.Taut ->
            Just PL_SS.Taut
```

Código 4.40: Implementación de los trabajos de Herbrand en LogicUS

Para finalizar, expondremos un ejemplo de uso de éste módulo, en el que mostraremos, desde una perspectiva aplicada, los distintos conceptos e implementaciones expuestas durante el desarrollo de este módulo y el módulo de *LogicUS.FOL.NormalForms*. (Los ejemplos relativos al módulo de LogicUS.FOL.Clauses se tratarán junto con los del módulo de LogicUS.FOL.Resolution).


Preview ProblemaTX150_FONFH.md
Félix y el TX-150
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Félix tiene en su garaje un taller que utiliza para la reparación de máquinas. Un buen amigo suyo va a hacerle una visita y obtiene la siguiente información:
Todos los ordenadores son máquinas.
El TX-150 es un ordenador
Para cualquier máquina, Félix sabe sólo cómo estropearla o sólo cómo arreglarla.
Toda máquina puede ser arreglada por alguien (incluso por otras máquinas), pero una máquina no puede arreglarse por sí misma.
Las cosas sólo desesperan a quienes no son capaces de arreglarlas.
Félix está deseperado con el TX-150.
Entonces, ¿puede arreglar Félix el TX-150?
Solución
Formalicemos el problema en el ámbito de la Lógica de Primer Orden. Para ello consideramos:
Los símbolos de predicado: $O(x)$: "x es un ordenador", $M(x)$: "x es una máquina", $A(x,y)$: "x puede arreglar y", $E(x,y)$: "x puede estropear y" y $D(x,y)$: "x desespera a y"
Y los símbolos de constante a y b para representar al TX-150 y a Félix, respectivamente.
Antes de nada, vamos a importar el módulo de Sintaxis y Semántica en Primer Orden y algunos otros paquetes que nos harán falta para poder trabajar con las fórmulas.
Entonces las fórmulas asociadas a cada uno de los asertos corresponden a:
import LogicUS.FOL.SintaxSemantics exposing (..)
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.FOL.NormalForms exposing (..)
import LogicUS.FOL.Herbrand exposing (..)
import LogicUS.PL.Clauses exposing (..)
import LogicUS.PL.Resolution exposing (..)
import Set exposing (Set)
import Dict exposing (Dict)
ffolRead : String -> FormulaFOL
ffolRead s = ffolReadExtraction <| ffolReadFromString s



$$\{\forall x\,(O(x)\to M(x)),O(a),\forall x\,(M(x)\to(A(b,x)\leftrightarrow\neg E(b,x))),\forall x\,(M(x)\to(\exists y\,A(y,x)\wedge\neg A(x,x))),\forall x\forall y\,(D(x,y)\leftrightarrow\neg A(y,x)),D(a,b)\}$$

Ahora veámos si es posible que Félix pueda arreglar el TX-150. Esto es si el conjunto de las fórmulas anteriores junto con $A(b,a)$ es consistente.

```
psi7 : FormulaFOL
psi7 = ffolRead "A(*b, *a)"
```

$$A(b,a)$$

Probémos que no utilizando la extensión de Herbrand. Para ello, hemos primero de Skolemizar las fórmulas, lo cuál requiere, primero pasar las fórmulas a forma Prenex y después aplicar las adecuadas sustituciones de Skolem. Veámoslo en LogicUS.

Todas las fórmulas, excepto ψ_4 están ya en forma Prenex, de hecho están casi en forma de Skolem (ya que no contienen cuantificadores existenciales). Por lo que pasemos dicha fórmula también a forma Prenex.

```
psi4p : FormulaFOL
psi4p = ffolToPrenex psi4
```

$$\forall x\,\exists y\,(M(x)\to(A(y,x)\wedge\neg A(x,x)))$$

Si queremos mostrar el proceso de construcción de la fórmula a través del árbol de formación:

```
psi4pg : String
psi4pg = prenexGraphToDOT <| (\ (_, _, gr) -> gr)  <| ffolToPrenex2 psi4
```

Una vez tenemos las fórmula en la forma Prenex podemos pasar a forma de Skolem sin más que aplicar las sustituciones de Skolem pertinentes.

```
psi4sk : FormulaFOL
psi4sk =  ffolToSkolem psi4p
```

$$(M(x) \to (A(\S_1(x), x) \land \neg A(x, x)))$$


Preview ProblemaTX150_FONFH.md
Podemos obtener también la sustitución de skolem realizada:
psi4sk_sub : Substitution
psi4sk_sub = getSkolemSubs <| (Tuple.first<<extractHeaderCuantifiers) psi4p
$\{y/\S_1(x)\}$
Mostrado con detalle cómo realizar el paso a forma de Skolem de una fórmula vamos a pasar todo el conjunto a forma de Skolem.
fs : SetFOL
fs = List.map (ffolToSkolem) [psi1, psi2, psi3, psi4, psi5, psi6, psi7]
$\{(O(x) \to M(x)), O(a), (M(x) \to (A(b,x) \leftrightarrow \neg E(b,x))), (M(x) \to (A(\S_1(x),x) \wedge \neg A(x,x))), (D(x,y) \leftrightarrow \neg A(y,x)), D(a,b), A(b,a)\}$
Ahora hemos de verificar la insatisfactibilidad del conjunto a través de la extensión de Herbrand. Nótese que la extensión asociada al problema es ifinita ya que se posee un símbolo de función (no constante).
De forma que para establecer la insatisfactibilidad podemos utilizar el siguiente algoritmo:
Algoritmo
Input: Σ: Un conjunto de fórmulas de un Lenguaje de Primer Orden.
Output: La EH de orden mínimo que es inconsistente si el conjunto es inconsistente. En otro caso $\uparrow$
Procedimiento:
(1) Hacer $i \leftarrow 1$
(2) Si EH(\Sigma) de orden k es inconsiste entonces PARAR y DEVOLVER i
(3) Si no, hacer $i \leftarrow i+1$ y volver a (2)
Programemos dicho algoritmo en formato recursivo y estableciendo un orden máximo.
insatConEH : SetFOL -> Int -> (Int, ClausePLSet)
insatConEH s maxO =
insatConEHAux s maxO 0
insatConEHAux : SetFOL -> Int -> Int -> (Int, ClausePLSet)
insatConEHAux s maxO i =
let
eh_i = splToClauses <| Maybe.withDefault [] <| sfolHerbrandExtension s i
in
if i > maxO then
(i, [])
else if (Tuple.first << csplSCFResolution) eh_i then
(i, eh_i)
else
insatConEHAux s maxO (i+1)

Figura 4.13: Ejemplo de uso de LogicUS.FOL.NormalForms y LogicUS.FOL.Herbrand
Fuente propia. Creada con litvis

4.2.17. LogicUS.FOL.Unification y LogicUS.FOL.Resolution

En este apartado vamos a presentar la implementación del algoritmo de Resolución de Primer Orden, que hará uso de la unificación de términos y átomos, por lo que las presentaremos conjuntamente.

LogicUS.FOL.Unification

Este módulo proporciona las herramientas para el cálculo del Unificador de Máxima Generalidad (MGU) de dos átomos o términos. Recuérdese que la unificación de dos átomos consiste en una doble sustitución (una para cada átomo) de forma que $A\{s_1\} \equiv B\{s_2\}$. Dicho unificador, u, se dice de máxima generalidad si cualquier otro unificador, v, para los átomos puede describirse como la composición de u con otra sustitución v'.

Pasemos ahora a tratar la implementación del cálculo de unificación de términos y átomos. Recuérdese que:

- Dos términos t y t' son unificables si :
  - Ambos corresponden a variables. En tal caso la unificación es la sustitución de una de las variables por la otra.
  - Uno de ellos es una variable y el otro es una función (o constante), que no depende de dicha variable. De manera que la unificación correspondería a sustituir la variable por la función.
- Dos listas de términos ts y ts' son unificables si:
  - Ambas son listas vacías. De forma que la unificación corresponde a la sustitución vacía.
  - Ambas tienen el mismo número de elementos, los términos son unificables dos a dos y no se producen incompatibilidades, esto es, que una misma variable tenga que ser sustituida por dos funciones distintas (realizando las sucesivas operaciones de reducción por transitividad).
- Dos átomos son unificables si los respectivos símbolos de predicado son idénticos y los argumentos de ambos predicados son unificables.

En el formalismo expusimos el algoritmo (ver *algoritmo 2.8*) de unificación para cada uno de los casos anteriores. Dicho algoritmo ha sido implementado en Elm, casi mediante una descripción (en el propio lenguaje) de las funciones allí descritas. Dicha implementación se encuentra detallada en el *código 4.41*.

```
termMGU : Term -> Term -> Maybe Substitution
termMGU t1 t2 =
    case t1 of
        Var x ->
            case t2 of
                Var y ->
                    if x == y then Just Dict.empty
                    else Just <| Dict.fromList [ ( x, t2 ) ]

                Func _ _ ->
                    if List.member x (FOL_SS.termVarSymbols t2) then Nothing
                    else Just <| Dict.fromList [ ( x, t2 ) ]

        Func f fts ->
            case t2 of
                Var y ->
                    if List.member y (FOL_SS.termVarSymbols t1) then Nothing
                    else Just <| Dict.fromList [ ( y, t1 ) ]

                Func g gts ->
                    if f == g then termsMGU fts gts
                    else Nothing
```

```
termsMGU : List Term -> List Term -> Maybe Substitution
termsMGU lt1 lt2 =
    case ( lt1, lt2 ) of
        ( [], [] ) ->
            Just Dict.empty

        ( _ :: _, [] ) ->
            Nothing

        ( [], _ :: _ ) ->
            Nothing

        ( t :: ts, r :: rs ) ->
            let
                s1_ =
                    termMGU t r
            in
            case s1_ of
                Just s1 ->
                    let
                        s2_ =
                            termsMGU
                                (List.map (\x -> FOL_SS.termApplySubstitution s1 x) ts)
                                (List.map (\x -> FOL_SS.termApplySubstitution s1 x) rs)
                    in
                    Maybe.map finalizelistTermMGU
                        <| Maybe.map (\s -> FOL_SS.substitutionComposition s s1) s2_

                _ ->
                    Nothing




finalizelistTermMGU : Substitution -> Substitution
finalizelistTermMGU s =
    let
        sf =
            Dict.filter (\x tr -> Var x /= tr) s
    in
    case
    List.filter
            (\( x, _ ) ->
                List.member x (FOL_SS.termsVarSymbols <| Dict.values <| Dict.remove x sf)
            )
            (Dict.toList sf)
    of
        [] ->
            sf

        ( k, v ) :: _ ->
            FOL_SS.substitutionComposition
                (Dict.remove k sf)
                (Dict.singleton k v)


atomsMGU : FormulaFOL -> FormulaFOL -> Maybe Substitution
atomsMGU a1 a2 =
    case ( a1, a2 ) of
        ( Pred p1 ts1, Pred p2 ts2 ) ->
            if p1 == p2 then
                termsMGU ts1 ts2

            else
                Nothing

        _ ->
            Nothing
```

Código 4.41: Implementación del cálculo de MGU en LogicUS

LogicUS.FOL.Resolution

En este módulo se exponen los mecanismos para llevar a cabo la resolución no restringida sobre cláusulas de Primer Orden. Para ello, se han definido una serie de estructuras y funciones que permitan llevar a cabo el procedimiento de resolución.

Recuérdese que un concepto fundamental en la resolución en LPO es la separación de cláusulas, de forma que dos cláusulas serán resolubles, entre otras restricciones, sólo si no contienen las mismas variables. En caso de que no sean cláusulas separadas se realiza un renombramiento de las variables para hacerlas separadas. Para ello, se toman las variables de ambas cláusulas y, para aquellas que son comunes a ambas, se sustituyen, en la segunda cláusula, por una variable del mismo símbolo pero con el primer superíndice disponible (que no aparezca en ninguna de las dos cláusulas).

```
cfol2SeparationSubst : ClauseFOL -> ClauseFOL -> Substitution
cfol2SeparationSubst c1 c2 =
    let
        vs_r =
            Set.toList <|
                Set.filter
                    (\x -> Set.member x <| FOL_CL.cfolVarSymbols c1)
                    (FOL_CL.cfolVarSymbols c2)

        vs_c =
            Set.union
                (FOL_CL.cfolVarSymbols c1)
                (FOL_CL.cfolVarSymbols c2)
    in
    let
        getIndex s is i =
            if Set.member ( s, is, i ) vs_c then
                getIndex s is (i + 1)

            else
                i
    in
    let
        subs =
            Dict.fromList
                <| List.map
                    (\ ( s, is, i ) ->
                        ( ( s, is, i ), Var ( s, is, getIndex s is 0 ) ))
                    vs_r
    in
    subs

cfol2Separation : ClauseFOL -> ClauseFOL -> ( ClauseFOL, ClauseFOL )
cfol2Separation c1 c2 =
    let
        s =
            cfol2SeparationSubst c1 c2
    in
    ( c1, FOL_CL.cfolApplySubstitution s c2 )
```

Código 4.42: Implementación de separación de cláusulas en LogicUS

Una vez se dispone del procedimiento de cálculo de unificación y de separación de cláusulas podemos implementar la regla de resolución de dos cláusulas. Recuérdese que para que dos cláusulas sean resolubles han de poseer dos literales complementarios cuyos átomos sean unificables, correspondiendo la resolvente a:

$$\frac{A \cup \{L\}\ ,\ B \cup \{L'\}}{(A \cup B)\{UMG(\{l, l'\})\}}$$

Donde L, L' corresponde al literal y el complementario y l y l' a sus respectivos átomos.

La función `cfol2ContraryLiterals` se encarga de la búsqueda de esos dos literales complementarios, mientras que la función `cfol2AllResolvents` calcula todas las posibles resolventes, comprobando si los átomos de cada pareja de literales complementarios son unificables.

Con estas dos funciones es posible implementar el algoritmo de resolución no restringida en Primer Orden sin más que aplicar el algoritmo de resolución proposicional, algoritmo de búsqueda, (presentado en el módulo LogicUS.PL.Resolution), cambiando la regla de resolución proposicional por la regla de resolución en Primer Orden, por lo que la implementación es totalmente equivalente. Para muestra de ello, en el *código 4.43* se muestra la implementación de las funciones de obtención de resolventes y del algoritmo de resolución.

Como en casos anteriores, haciendo que los nodos correspondan a las cláusulas y las aristas a las sustituciones (separación + MGU) aplicadas en cada una de las resolventes, obtenemos un grafo que puede ser representado tanto en formato cadena como en formato DOT visualizable con GraphViz.

Para finalizar la descripción del módulo exponemos un ejemplo de uso del mismo (que servirá también de ejemplo del módulo LogicUS.FOL.Clauses) en el que se muestran los distintos mecanismos descritos con anterioridad.

```
cfol2ContraryLiterals : ClauseFOL -> ClauseFOL -> List ( ClauseFOLLiteral, ClauseFOLLiteral )
cfol2ContraryLiterals c1 c2 =
    let
        searchContraryLiteral ( a, sign ) c =
            List.filter (\( a_, sign_ ) -> FOL_CL.cfolAtomSymbol a == FOL_CL.cfolAtomSymbol a_ && sign /= sign_) c
    in
    List.foldl (\( a, sign ) ac -> ac ++ (List.map (\l -> ( ( a, sign ), l )) <| searchContraryLiteral ( a, sign ) c2)) [] c1


cfol2AllResolvents : ClauseFOL -> ClauseFOL -> List Resolvent
cfol2AllResolvents c1 c2 =
    let
        r =
            cfol2SeparationSubst c1 c2

        c2_ =
            FOL_CL.cfolApplySubstitution r c2
    in
    let
        cls =
            cfol2ContraryLiterals c1 c2_
    in
    List.foldl
        (\( ( a1, sg1 ), ( a2, sg2 ) ) ac ->
            case FOL_UN.atomsMGU (FOL_CL.clauseFOLAtomToAtom a1) (FOL_CL.clauseFOLAtomToAtom a2) of
                Just mgu ->
                    let
                        c1__ =
                            List.filter
                                (\x -> x /= ( FOL_CL.cfolAtomApplySubstitution mgu a1, sg1 ))
                                (FOL_CL.cfolApplySubstitution mgu c1)

                        c2__ =
                            List.filter
                                (\x -> x /= ( FOL_CL.cfolAtomApplySubstitution mgu a2, sg2 ))
                                (FOL_CL.cfolApplySubstitution mgu c2_)
                    in
                    ac ++ [ { c1 = c1, c2 = c2, r = r, mgu = mgu, res = FOL_CL.cfolUnion c1__ c2__ } ]

                Nothing ->
                    ac
        )
        []
        cls
```

```
recoverResolutionPath : Int -> Dict Int ResolutionItem -> ( List (Node ClauseFOL), List (Edge ( Substitution, Substitution )) )
recoverResolutionPath i refDict =
    case Dict.get i refDict of
        Just ri ->
            let
                ( nodes1, edges1 ) =
                    recoverResolutionPath ri.p1 refDict

                ( nodes2, edges2 ) =
                    recoverResolutionPath ri.p2 refDict
            in
            ( uniqueConcatList nodes1 nodes2 ++ [ Node i ri.c ]
            , uniqueConcatList edges1 edges2
                ++ [ Edge ri.p1 i ( Dict.empty, Dict.filter (\k _ -> Set.member k (FOL_CL.cfolVarSymbols <| Maybe.withDefault []
                        <| Maybe.map .c <| Dict.get ri.p1 refDict)) ri.mgu )
                   , Edge ri.p2 i ( ri.r, Dict.filter (\k _ -> Set.member k (FOL_CL.cfolVarSymbols <| Maybe.withDefault []
                        <| Maybe.map .c <| Dict.get ri.p2 refDict)) ri.mgu )
                   ]
            )

        _ ->
            ( [], [] )


resolventsWithClosedSCFResolutionAux : List ( Int, ResolutionItem ) -> Int -> ClauseFOL -> List ResolutionItem
resolventsWithClosedSCFResolutionAux closed id c =
    List.foldl
        (\( i, ri ) ac ->
            List.foldl
                (\r ac2 ->
                    if not
                        <| FOL_CL.cfolIsTautology r.res || List.any (\x -> FOL_CL.cfolSubsumes x.c r.res) ac
                            || List.any (\( _, x ) -> FOL_CL.cfolSubsumes x.c r.res) closed
                    then
                        { c = r.res, p1 = id, p2 = i, r = r.r, mgu = r.mgu } ::
                            List.filter (\x -> not <| FOL_CL.cfolSubsumes r.res x.c) ac2

                    else
                        ac2
                )
                ac
                (cfol2AllResolvents ri.c c)
        )
        []
        closed
```

```
openedUpdationSCFResolutionAux : List ( Int, ResolutionItem ) -> List ( Int, ResolutionItem ) -> List ( Int, ResolutionItem )
openedUpdationSCFResolutionAux xs new_opens =
    let
        res =
            List.foldl
                (\( li, ri ) ( ac, rest ) ->
                    let
                        add_ac =
                            LE.takeWhile (\( lx, _ ) -> lx <= li) rest
                    in
                    if List.any (\( _, x ) -> FOL_CL.cfolSubsumes x.c ri.c) (ac ++ add_ac) then
                        ( ac ++ add_ac, List.drop (List.length add_ac) rest )

                    else
                        ( ac ++ add_ac ++ [ ( li, ri ) ]
                        , List.filter (\( _, x ) -> not (FOL_CL.cfolSubsumes ri.c x.c)) <| List.drop (List.length add_ac) rest
                        )
                )
                ( [], xs )
                new_opens
    in
    Tuple.first res ++ Tuple.second res


csfolSCFResolutionAux : List ( Int, ResolutionItem ) -> List ( Int, ResolutionItem ) -> Int ->
    ( List (Node ClauseFOL), List (Edge ( Substitution, Substitution )) )
csfolSCFResolutionAux closed opened nid =
    case opened of
        [] -> ( [], [] )

        ( _, ri ) :: xs ->
            if List.isEmpty ri.c then
                let refDict = Dict.fromList <| closed ++ [ ( nid + 1, ri ) ] in
                    recoverResolutionPath (nid + 1) refDict

            else
                let
                    r_closed = resolventsWithClosedSCFResolutionAux closed (nid + 1) ri.c
                in
                let
                    new_closed = closed ++ [ ( nid + 1, ri ) ]

                    new_opened = openedUpdationSCFResolutionAux xs (List.sortBy (\x -> Tuple.first x)
                        <| List.map (\x -> ( List.length x.c, x )) r_closed)
                in
                csfolSCFResolutionAux new_closed new_opened (nid + 1)
```

```
csfolSCFResolution : List ClauseFOL -> ( Bool, ResolutionTableau )
csfolSCFResolution clauses =
    let
        cs =
            FOL_CL.csfolRemoveEqClauses clauses
    in
    let
        new_cs =
            List.sortBy (\x -> Tuple.first x) <|
                List.map
                    (\x -> ( List.length x, { c = x, p1 = 0, p2 = 0, r = Dict.empty, mgu = Dict.empty } ))
                    (FOL_CL.csfolRemoveSubsumedClauses <| FOL_CL.csfolRemoveTautClauses <| cs)
    in
    let
        ( nodes, edges ) =
            csfolSCFResolutionAux [] new_cs 0
    in
    let
        nid_max =
            Maybe.withDefault 0 <| List.maximum <| List.map (\x -> x.id) <| nodes

        nodes_clauses =
            List.map (\x -> x.label) <| nodes
    in
    let
        final_nodes =
            List.map (\x -> Node x.id ( List.member x.label cs, x.label )) nodes
                ++ (List.indexedMap (\i x -> Node (nid_max + i + 1) ( True, x ))
                    <| List.filter (\x -> not (List.member x nodes_clauses)) cs)
    in
    ( edges /= [], Graph.fromNodesAndEdges final_nodes edges )
```

Código 4.43: Implementación de resolución de cláusulas en LogicUS


Preview ProblemaTX150_FOR.md
Félix y el TX-150
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Félix tiene en su garaje un taller que utiliza para la reparación de máquinas. Un buen amigo suyo va a hacerle una visita y obtiene la siguiente información:
Todos los ordenadores son máquinas.
El TX-150 es un ordenador
Para cualquier máquina, Félix sabe sólo cómo estropearla o sólo cómo arreglarla.
Toda máquina puede ser arreglada por alguien (incluso por otras máquinas), pero una máquina no puede arreglarse por sí misma.
Las cosas sólo desesperan a quienes no son capaces de arreglarlas.
Félix está deseperado con el TX-150.
Entonces, ¿puede estropear Félix el TX-150?
Solución
Formalicemos el problema en el ámbito de la Lógica de Primer Orden. Para ello consideramos:
Los símbolos de predicado: $O(x)$: "x es un ordenador", $M(x)$: "x es una máquina", $A(x,y)$: "x puede arreglar y", $E(x,y)$: "x puede estropear y" y $D(x,y)$: "x desespera a y"
Y los símbolos de constante a y b para representar al TX-150 y a Félix, respectivamente.
Antes de nada, vamos a importar el módulo de Sintaxis y Semántica en Primer Orden y algunos otros paquetes que nos harán falta para poder trabajar con las fórmulas.
Entonces las fórmulas asociadas a cada uno de los asertos corresponden a:
import LogicUS.FOL.SintaxSemantics exposing (..)
import LogicUS.PL.SintaxSemantics exposing (..)
import LogicUS.FOL.NormalForms exposing (..)
import LogicUS.FOL.Herbrand exposing (..)
import LogicUS.PL.Clauses exposing (..)
import LogicUS.PL.Resolution exposing (..)
import Set exposing (Set)
import Dict exposing (Dict)
ffolRead : String -> FormulaFOL
ffolRead s = ffolReadExtraction <| ffolReadFromString s

Preview ProblemaTX150_FOR.md

```
psi1 : FormulaFOL
psi1 = ffolRead "!A[x](O(x) -> M(x))"

psi2 : FormulaFOL
psi2 = ffolRead "O(*a)"

psi3 : FormulaFOL
psi3 = ffolRead "!A[x](M(x) -> (A(*b, x) <-> ¬E(*b,x)))"

psi4 : FormulaFOL
psi4 = ffolRead "!A[x](M(x) -> (!E[y] A(y, x) & ¬A(x,x)))"

psi5 : FormulaFOL
psi5 = ffolRead "!A[x]!A[y](D(x,y) <-> ¬A(y,x))"

psi6 : FormulaFOL
psi6 = ffolRead "D(*a,*b)"
```

$$\{\forall x\,(O(x)\to M(x)),O(a),\forall x\,(M(x)\to(A(b,x)\leftrightarrow\neg E(b,x))),\forall x\,(M(x)\to(\exists y\,A(y,x)\wedge\neg A(x,x))),\forall x\forall y\,(D(x,y)\leftrightarrow\neg A(y,x)),D(a,b)\}$$

Ahora veámos si de la información anterior podemos deducir que Félix ha de poder estropear el TX-150. Esto es si $\Psi\models E(b,a)$.

```
psi7 : FormulaFOL
psi7 = ffolRead "E(*b, *a)"
```

$$E(b,a)$$

Para poder utilizar el método de resolución no restringida, primero hemos de pasar las fórmulas a forma Clausal. De forma que el conjunto de cláusulas asociadas a $\Psi\cup\{\neg E(*b,*a)\}$ corresponde a

```
cs : ClauseFOLSet
cs = sfolToClauses [psi1, psi2, psi3, psi4, psi5, psi6, Neg psi7]
```

$$\{\{M(x),\neg O(x)\},\{O(a)\},\{\neg A(b,x),\neg E(b,x),\neg M(x)\},\{A(b,x),E(b,x),\neg M(x)\},\{A(\$_1(x),x),\neg M(x)\},\{\neg A(x,x),\neg M(x)\},\{\neg A(y,x),\neg D(x,y)\},\{A(y,x),D(x,y)\},\{D(a,b)\},\{\neg E(b,a)\}\}$$

Una vez obtenida el conjunto de cláusulas podemos aplicar la Resolución no restringida en Primer Orden como sigue. Comprobamos que es insatisfactible:

```
res : ( Bool, ResolutionTableau )
res = csfolSCFResolution cs
```

True

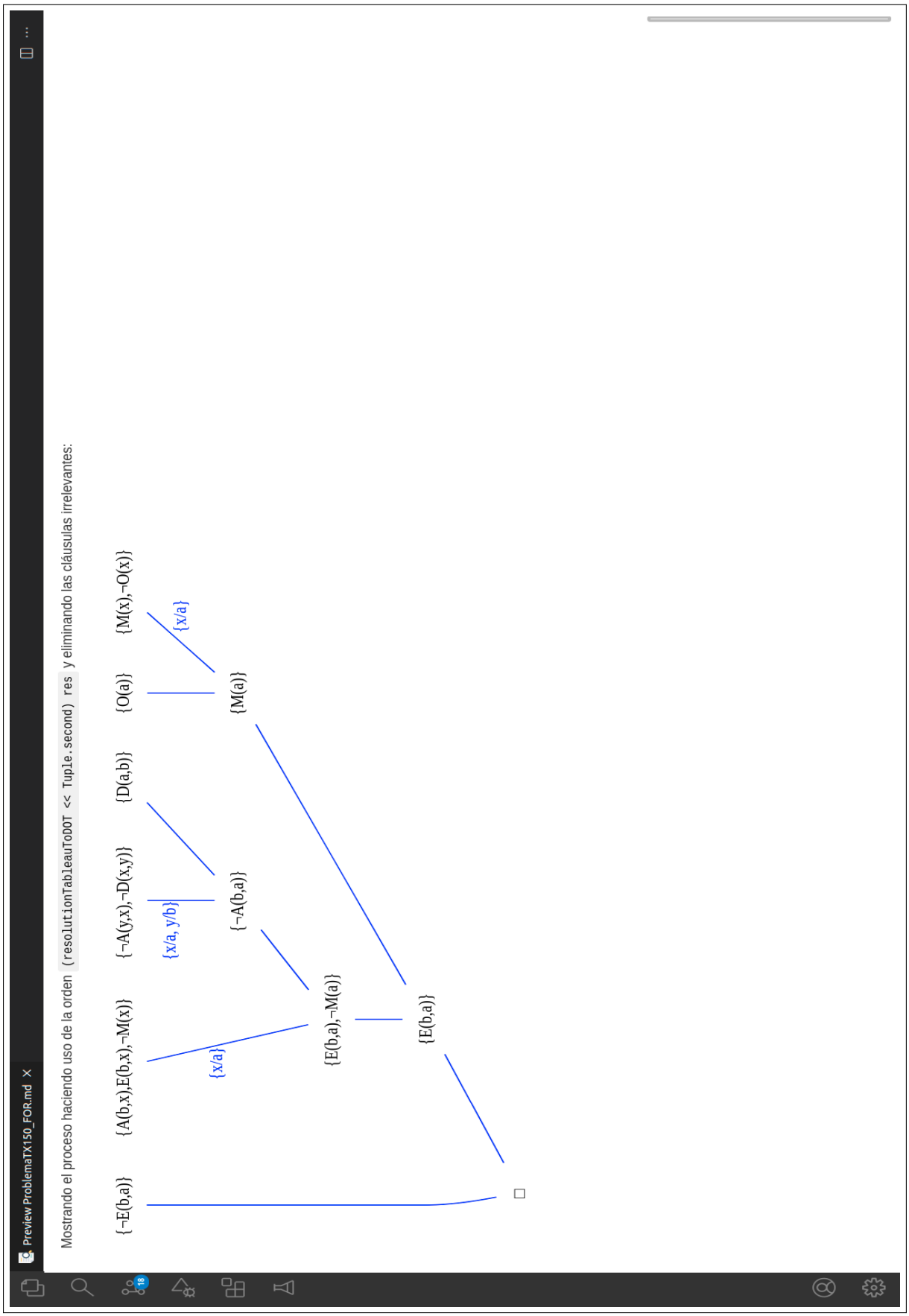


Figura 4.14: Ejemplo de uso de LogicUS.FOL.Clauses y LogicUS.FOL.Resolution
Fuente propia. Creada con litvis

5 | Instalación y Uso de LogicUS

En este capítulo mostraremos los requisitos para la instalación del entorno así como las distintas vías para el manejo de las herramientas desarrolladas.

5.1. Instalación del entorno

El entorno está accesible tanto para Windows como para sistemas basados en Linux (Ubuntu, MacOS, ...). Describiremos las instrucciones generales y comentaremos algún detalle que pueda cambiar según el SO anfitrión:

1. Instalación de NodeJS y *npm*: Basta ir a la a la página de descargas de NodeJS (https://nodejs.org/es/download/) y descargar y elegir la versión que se quiere descargar (que ya tiene incluído *npm*) según el sistema operativo anfitrión. En Linux puede instalrse a través del gestor de paquetes ATP con las órdenes `sudo apt install nodejs` y `sudo apt install npm`.

2. Instalación de Elm: Basta acudir a la página guía de inicio de Elm, al apartado de instalación concretamente, (https://guide.elm-lang.org/install/elm.html) y descargar el instalador de Elm (disponible para Windows y Mac) o seguir las instrucciones de instalación para Linux (que consiste en descargar la fuente, hacerla ejecutable y añadirla al directorio de ejecución, normalmente `/usr/local/bin`).

3. Instalación de algunas extensiones para *npm*. Concretamente `elm-test` y `elm-format` con la instrucción `sudo npm install --global --unsafe-perm elm-test elm-format`.

4. Instalación de litvis (con VSCode o Atom). Es necesario disponer de una herramienta que permita ejecutar el software de *litvis*, el cual está disponible a través de la instalación de una extensión tanto en VSCode como en Atom. A continuación describiremos la instalación de cada una de ellas.

   Visual Studio Code + litvis

   4.A1 Instalación de *VSCode*: Basta ir a la a la página de descargas de Visual Studio Code (https://code.visualstudio.com/download) y elegir la versión que se quiere descargar según el sistema operativo anfitrión.

   4.A2 Instalación y configuración de *litvis*. Para poder hacer uso de *litvis* es suficiente con instalar algunas extensiones en VSCode, concretamente la extensión *elm* (de Elm tooling), la extensión *markdown-preview-enhanced-with-litvis* (de giCentre) y *prettier - Code formatter* (de Prettier) y realizar las configuraciones (acudiendo a `File >Preferences >Settings`):

   - En `Text Editor >Formatting` habilitar la opción `Format On Save`
   - En `Extenssions >Markdown Preview Enhanced with litvis` desabilitar `Live Update`
   - En `Extenssions >Prettier` habilitar `Resolve Global Modules`.

Atom + litvis

4.B1 Instalación de *Atom*: Basta ir a la a la página de descargas de Atom (https://atom.io/) y elegir la versión que se quiere descargar según el sistema operativo anfitrión.

4.B2 Instalación y configuración de *litvis*. Para poder hacer uso de *litvis* es suficiente con instalar algunas extensiones en Atom, concretamente las extensiones *language-elm*, *language-markdown markdown-preview-enhanced-with-litvis* y *prettier-atom*. (NOTA: Al crear el primer documento *litvis* pueden ser necesarioas otras dependecias como *linter*, *linter-ui-default* y *busy-signal*. Instalas dichas referencias también, si son solicitadas).

y realizar las configuraciones, acudiendo a `Settings >Packages` (Windows), *Atom ¿Preferences ¿Packages* (MacOS) o (Edit ¿Preferences ¿Packages):

- En `Prettier-atom` acceder a los ajustes (`Settings`) y habilitar la opción `Format files On Save`
- En `Markdown Preview Enhanced with litvis` acceder a los ajustes (`Settings`) y desabilitar `live Update`

Una vez realizada la instalación, podemos hacer uso de dos herramientas para el uso de las librerías de LogicUS. En esta sección se muestran distintas alternativas para el uso de las librerías y funciones diseñadas. En el repositorio LogicUS de Elm (vicramgon/logicus) se encuentra publicada toda la información de cada uno de los módulos que contiene el paquete y de las funciones que estos contienen. En concreto, se abordará el uso de dos herramientas : *Elm REPL* y *litvis*. Aunque, seguidamente, abordaremos las mismas con más detalle, el uso de ambas herramientas es equivalente en cuanto a la sintaxis del código, aunque el uso de la consola está enfocado más a un uso puntual y reactivo que *litvis*, cuyo propósito es la generación de documentos (documentación teorico-práctica, ejercicios resueltos, prácticas, ...).

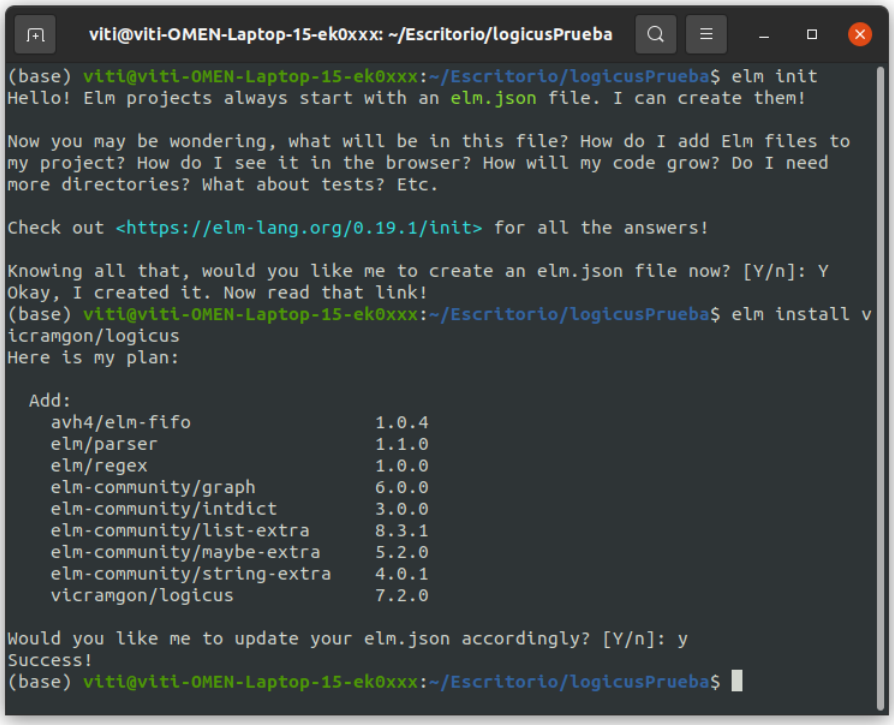


Figura 5.1: Iniciación del proyecto e instalación de librerías
Fuente propia

5.2. Elm REPL

Al realizar la instalación de elm, Elm REPL es instalado automáticamente, pero para poder utilizarlo debemos de crear primero un proyecto (aunque sea de forma temporal) incluyendo las librerías necesarias (LogicUS) en las dependencias del mismo. Para ello se ejecutará la orden `elm init` y tras ello se descargará y referenciará la librería LogicUS utilizando la instrucción `elm install vicramgon/logicus`.

Antes de utilizar alguna función de un módulo debemos importarlo para lo cual utilizamos la orden `import LogicUS.Section.Module exposing (..)` . En los argumentos de exposing se pueden establecer qué funciones son las que se importan, y aunque es común utilizar el comodín (..) (establece que se importen todas las funciones) hay que tener un poco de cuidado porque si dos módulos distintos contienen funciones con un mismo nombre, existirá ambigüedad en su llamada y Elm lanzará un error.

Aún así, puede ser necesario hacer uso de dos funciones con el mismo nombre de distintos módulos, M1 y M2. Para ello, podemos importar los módulos asignándoles un alias ejecutando `import LogicUS.(PL/FOL).M1 as M1 exposing (..)` y llamando a la función después ejecutando `M1.function arg1 arg2`.

Ahora ya podríamos hacer uso de las funciones de los distintos módulos. Veamos un ejemplo, utilizando resolución proposicional, si $F \equiv (p \to q) \leftrightarrow (q \leftrightarrow (p \vee q))$ es una tautología:

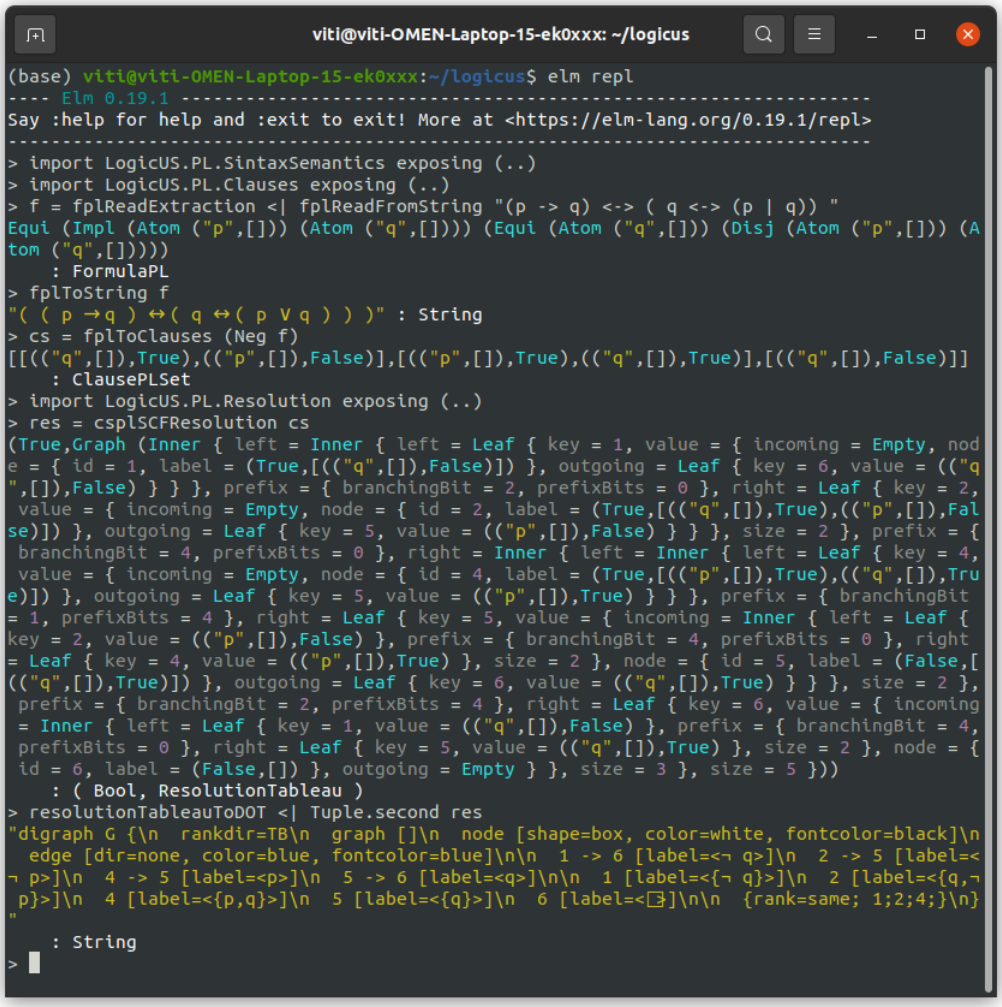


Figura 5.2: Uso de Elm REPL. Ejemplo
Fuente propia

Ahora podríamos copiar el texto en formato DOT y renderizarlo utilizando un visualizador online para ver el tablero resultante del proceso de resolución.

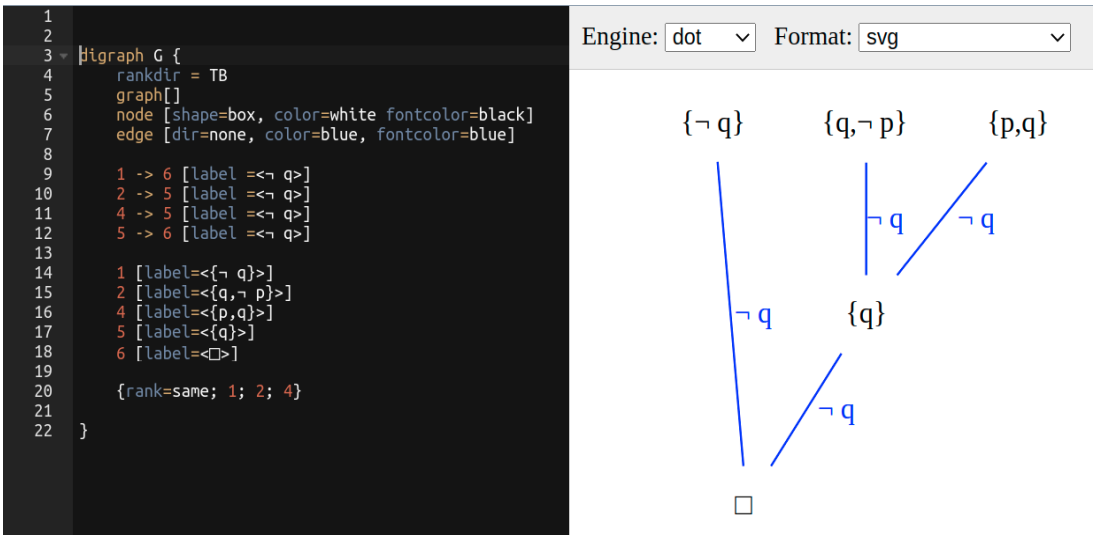


Figura 5.3: Uso de Elm REPL. Ejemplo (Tablero)
Fuente propia

5.3. LogicUS + *litvis*

Sin embargo, si queremos realizar y guardar algunos ejercicios, prácticas o alguna documentación con ejemplos, tendríamos que hacerlo a través de capturas y tendríamos que volver a escribir las órdenes para volver a ejecutarlo. Para facilitar esta tarea, la plataforma *litvis* permite crear documentos combinando el lenguaje Markdown junto a bloques ejecutables.

La extensión *markdown enhanced preview with litvis* permite generar una previsualización en tiempo real de la ejecución y renderización del documento, además de incorporar tecnologías como GraphViz que nos permiten visualizar, dentro del mismo documento, las distintas representaciones gráficas. Complementariamente a todo ello, permite además guardar el documento (con extensión *.md*) de forma que pueda compartirse y visualizarse en otros equipos haciendo uso de la extensión, y también generar otros documentos en formatos como HTML o PDF, lo que permitiría guardar ejercicios ya resueltos, generar documentación, etc.

Para poder utilizar la librería LogicUS, al igual que todos los paquetes que no son nativos de Elm, hemos de incluirlos como referencias. Para ello basta con especificarlas en la cabecera del documento, de forma similar a como se hace en el archivo de configuración de un proyecto *elm.json* pero utilizando el formato YAML. Un ejemplo de ello con la librería LogicUS correspondería a:

```
---
    elm:
        dependencies:
            vicramgon/logicus: latest
---
```

A continuación mostramos el código MD completo del ejercicio presentado anteriormente *Félix y el TX-150* abordándolo mediante resolución no restringida en Primer Orden. En él se pueden observar fragmentos de texto (incluyendo fórmulas en formato latex), bloques de ejecución de Elm y bloques *dot* que permiten la visualización de la salida generada por Elm sin más que pegar el código DOT generado en el bloque de Elm dentro del bloque. Mostramos finalmente, también, el método para la exportación a PDF.


ProblemaTX150_FOR.md - logicus - Visual Studio Code
File Edit Selection View Go Run Terminal Help
ProblemaTX150_FOR.md U
NOTEBOOKS > ProblemaTX150_FOR.md > # Félix y el TX-150 > ## Solución
Félix y el TX-150
> Fuente: Ejercios de Lógica Computacional
> (https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Félix tiene en su garaje un taller que utiliza para la reparación de máquinas. Un buen amigo suyo va a hacerle una visita y obtiene la siguiente información:
- Todos los ordenadores son máquinas.
- El TX-150 es un ordenador
- Para cualquier máquina, Félix sabe sólo cómo estropearla o sólo cómo arreglarla.
- Toda máquina puede ser arreglada por alguien (incluso por otras máquinas), pero una máquina no puede arreglarse por sí misma.
- Las cosas sólo desesperan a quienes no son capaces de arreglarlas.
- Félix está deseperado con el TX-150.
Entonces, ¿puede estropear Félix el TX-150?
Solución
Formalicemos el problema en el ámbito de la Lógica de Primer Orden. Para ello consideramos:
- Los símbolos de predicado: $O(x)$: "_x es un ordenador_", $M(x)$: "_x es una máquina_", $A(x,y)$: "_x puede arreglar y_", $E(x,y)$: "_x puede estropear y_" y $D(x, y)$: "_x desespera a y_"
- Y los símbolos de constante a y b para representar al TX-150 y a Félix, respectivamente.
Antes de nada, vamos a importar el módulo de Sintaxis y Semántica en
Preview ProblemaTX150_FOR.md
Félix y el TX-150
Fuente: Ejercios de Lógica Computacional
(https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21)
Félix tiene en su garaje un taller que utiliza para la reparación de máquinas. Un buen amigo suyo va a hacerle una visita y obtiene la siguiente información:
Todos los ordenadores son máquinas.
El TX-150 es un ordenador
Para cualquier máquina, Félix sabe sólo cómo estropearla o sólo cómo arreglarla.
Toda máquina puede ser arreglada por alguien (incluso por otras máquinas), pero una máquina no puede arreglarse por sí misma.
Las cosas sólo desesperan a quienes no son capaces de arreglarlas.
Félix está deseperado con el TX-150.
Entonces, ¿puede estropear Félix el TX-150?
Solución
Formalicemos el problema en el ámbito de la Lógica de Primer Orden. Para ello consideramos:
Los símbolos de predicado: $O(x)$: "x es un ordenador", $M(x)$: "x es una máquina", $A(x,y)$: "x puede arreglar y", $E(x,y)$: "x puede estropear y" y $D(x,y)$: "x desespera a y"
Y los símbolos de constante a y b para representar al TX-150 y a Félix, respectivamente.
Antes de nada, vamos a importar el módulo de Sintaxis y Semántica en Primer Orden y algunos otros paquetes que nos harán falta para poder trabajar con las fórmulas.
Entonces las fórmulas asociadas a cada uno de los asertos corresponden a:
master*
Ln 106, Col 228
Spaces: 2
UTF-8
LF
Markdown
Go Live
Prettier

```
54
55  Entonces las fórmulas asociadas a cada uno de los asertos
    corresponden a:
56
57  ```elm {l id="imports"}
58
59  import LogicUS.FOL.SintaxSemantics exposing (..)
60  import LogicUS.FOL.Clauses exposing (..)
61  import LogicUS.FOL.Resolution exposing (..)
62  import LogicUS.FOL.NormalForms exposing (..)
63  import Set exposing (Set)
64  import Dict exposing (Dict)
65
66  ffolRead : String -> FormulaFOL
67  ffolRead s = ffolReadExtraction <| ffolReadFromString s
68  ```
69
70  ```elm {l m=("$ " ++ (sfolToMathString2 [psi1, psi2, psi3, psi4,
    psi5, psi6]) ++ " $") context="1" follows="imports"}
71  psi1 : FormulaFOL
72  psi1 = ffolRead "!A[x](O(x) -> M(x))"
73
74  psi2 : FormulaFOL
75  psi2 = ffolRead "O(*a)"
76
77  psi3 : FormulaFOL
78  psi3 = ffolRead "!A[x](M(x) -> (A(*b, x) <-> ¬E(*b,x)))"
79
80  psi4 : FormulaFOL
81  psi4 = ffolRead "!A[x](M(x) -> (!E[y] A(y, x) & ¬A(x,x)))"
82
83  psi5 : FormulaFOL
84  psi5 = ffolRead "!A[x]!A[y](D(x,y) <-> ¬A(y,x))"
85
86  psi6 : FormulaFOL
87  psi6 = ffolRead "D(*a,*b)"
88  ```
89
90  Ahora veámos si de la información anterior podemos deducir que Félix
```

Entonces las fórmulas asociadas a cada uno de los asertos corresponden a:

```
import LogicUS.FOL.SintaxSemantics exposing (..)
import LogicUS.FOL.Clauses exposing (..)
import LogicUS.FOL.Resolution exposing (..)
import LogicUS.FOL.NormalForms exposing (..)
import Set exposing (Set)
import Dict exposing (Dict)

ffolRead : String -> FormulaFOL
ffolRead s = ffolReadExtraction <| ffolReadFromString s
```

```
psi1 : FormulaFOL
psi1 = ffolRead "!A[x](O(x) -> M(x))"

psi2 : FormulaFOL
psi2 = ffolRead "O(*a)"

psi3 : FormulaFOL
psi3 = ffolRead "!A[x](M(x) -> (A(*b, x) <-> ¬E(*b,x)))"

psi4 : FormulaFOL
psi4 = ffolRead "!A[x](M(x) -> (!E[y] A(y, x) & ¬A(x,x)))"

psi5 : FormulaFOL
psi5 = ffolRead "!A[x]!A[y](D(x,y) <-> ¬A(y,x))"

psi6 : FormulaFOL
psi6 = ffolRead "D(*a,*b)"
```

$$\forall x\ (O(x) \to M(x))$$
$$O(a)$$
$$\forall x\ (M(x) \to (A(b,x) \leftrightarrow \neg E(b,x)))$$
$$\forall x\ (M(x) \to (\exists y\, A(y,x) \wedge \neg A(x,x)))$$
$$\forall x \forall y\ (D(x,y) \leftrightarrow \neg A(y,x))$$
$$D(a,b)$$

ProblemaTX150_FOR.md - logicus - Visual Studio Code
File Edit Selection View Go Run Terminal Help
ProblemaTX150_FOR.md U
NOTEBOOKS > ProblemaTX150_FOR.md > # Félix y el TX-150 > ## Solución
88 ```
89
90 Ahora veámos si de la información anterior podemos deducir que Félix ha de poder estropear el TX-150. Esto es si $\Psi \models E(b,a)$.
91
92 ```elm {l m=("$ " ++ (ffolToMathString psi7) ++ " $"), context="1"}
93 psi7 : FormulaFOL
94 psi7 = ffolRead "E(*b, *a)"
95 ```
96
97 Para poder utilizar el método de resolución no restringida, primero hemos de pasar las fórmulas a forma Clausal. De forma que el conjunto de cláusulas asociadas a $\Psi \cup \{\neg E(*b, *a)\}$ corresponde a
98
99 <!-- r=("$ " ++ (csfolToString cs) ++ " $") -->
100
101 ```elm {l context="1"}
102 cs : ClauseFOLSet
103 cs = sfolToClauses [psi1, psi2, psi3, psi4, psi5, psi6, Neg psi7]
104 ```
105
106 $ \{ \{M(x),\neg O(x)\}, \, \{O(a)\}, \, \{\neg A(b,x),\neg E(b,x), \neg M(x)\}, \, \{A(b,x),E(b,x),\neg M(x)\}, \\ \{A(ş₁(x),x),\neg M(x)\}, \, \{\neg A(x,x),\neg M(x)\}, \, \{\neg A(y,x),\neg D(x,y)\}, \, \{A(y,x),D(x,y)\}, \\ \{D(a,b)\}, \, \{\neg E(b,a)\}\} $
107
108 Una vez obtenida el conjunto de cláusulas podemos aplicar la Resolución no restringida en Primer Orden como sigue. Comprobamos que es insatisfactible:
109
110 ```elm {l r=(Tuple.first res) context="1"}
111 res : (Bool, ResolutionTableau)
112 res = csfolSCFResolution cs
113 ```
114
115 Mostrando el proceso haciendo uso de la orden `(resolutionTableauToDOT << Tuple.second) res` y eliminando las cláusulas irrelevantes:
116
Preview ProblemaTX150_FOR.md
Ahora veámos si de la información anterior podemos deducir que Félix ha de poder estropear el TX-150. Esto es si $\Psi \models E(b,a)$.
psi7 : FormulaFOL
psi7 = ffolRead "E(*b, *a)"
$E(b,a)$
Para poder utilizar el método de resolución no restringida, primero hemos de pasar las fórmulas a forma Clausal. De forma que el conjunto de cláusulas asociadas a $\Psi \cup \{\neg E(*b,*a)\}$ corresponde a
cs : ClauseFOLSet
cs = sfolToClauses [psi1, psi2, psi3, psi4, psi5, psi6, Neg psi7]
$\{\{M(x),\neg O(x)\}, \{O(a)\}, \{\neg A(b,x),\neg E(b,x),\neg M(x)\}, \{A(b,x),E(b,x),\neg M(x)\}, \{A(ş_1(x),x),\neg M(x)\}, \{\neg A(x,x),\neg M(x)\}, \{\neg A(y,x),\neg D(x,y)\}, \{A(y,x),D(x,y)\}, \{D(a,b)\}, \{\neg E(b,a)\}\}$
Una vez obtenida el conjunto de cláusulas podemos aplicar la Resolución no restringida en Primer Orden como sigue. Comprobamos que es insatisfactible:
res : (Bool, ResolutionTableau)
res = csfolSCFResolution cs
True
Mostrando el proceso haciendo uso de la orden (resolutionTableauToDOT << Tuple.second) res y eliminando las cláusulas irrelevantes:
{¬E(b,a)}
{A(b,x),E(b,x),¬M(x)}
{¬A(y,x),¬D(x,y)}
{D(a,b)}
{O(a)}
{x/a, y/b}
master*
Ln 106, Col 228
Spaces: 2
UTF-8
LF
Markdown
Go Live
Prettier

● ProblemaTX150_FOR.md - logicus - Visual Studio Code
File Edit Selection View Go Run Terminal Help
ProblemaTX150_FOR.md U
NOTEBOOKS > ProblemaTX150_FOR.md > # Félix y el TX-150 > ## Solución
(resolutionTableauToDOT << Tuple.second) res` y eliminando las cláusulas irrelevantes:
^^^elm{r=((resolutionTableauToDOT << Tuple.second) res) context="1"}^^^^
```dot
digraph G {
rankdir=TB
graph []
node [shape=box, color=white, fontcolor=black]
edge [dir=none, color=blue, fontcolor=blue]
1 -> 5
2 -> 12
3 -> 27
4 -> 5 [label=<{x/a}>]
5 -> 26
10 -> 12 [xlabel=<{x/a, y/b}>]
12 -> 25
24 -> 25 [xlabel=<{x/a} >]
25 -> 26
26 -> 27
1 [label=<{O(a)}>]
2 [label=<{D(a,b)}>]
3 [label=<{¬E(b,a)}>]
4 [label=<{M(x),¬O(x)}>]
5 [label=<{M(a)}>]
10 [label=<{¬A(y,x),¬D(x,y)}>]
12 [label=<{¬A(b,a)}>]
24 [label=<{A(b,x),E(b,x),¬M(x)}>]
25 [label=<{E(b,a),¬M(a)}>]
26 [label=<{E(b,a)}>]
27 [label=<□>]
{rank=same; 1;2;3;4;10;24}
}
```
Preview ProblemaTX150_FOR.md
cláusulas irrelevantes:
"digraph G {\n rankdir=TB\n graph []\n node [shape=box, color=white, fontcolor=black]\n edge [dir=none, color=blue, fontcolor=blue]\n\n 1 -> 5\n 2 -> 12\n 3 -> 27\n 4 -> 5 [label=<{x/a}>]\n 5 -> 26\n 10 -> 12 [label=<{x/a, y/b}>]\n 12 -> 25\n 24 -> 25 [label=<{x/a}>]\n 25 -> 26\n 26 -> 27\n\n 1 [label=<{O(a)}>]\n 2 [label=<{D(a,b)}>]\n 3 [label=<{¬E(b,a)}>]\n 4 [label=<{M(x),¬O(x)}>]\n 5 [label=<{M(a)}>]\n 10 [label=<{¬A(y,x),¬D(x,y)}>]\n 12 [label=<{¬A(b,a)}>]\n 24 [label=<{A(b,x),E(b,x),¬M(x)}>]\n 25 [label=<{E(b,a),¬M(a)}>]\n 26 [label=<{E(b,a)}>]\n 27 [label=<□>]\n 28 [label=<{¬A(b,x),¬E(b,x),¬M(x)}>]\n 29 [label=<{A(ş₁(x),x),¬M(x)}>]\n 30 [label=<{¬A(x,x),¬M(x)}>]\n 31 [label=<{A(y,x),D(x,y)}>]\n\n {rank=same; 1;2;3;4;10;24;28;29;30;31;}\n}"^
{¬E(b,a)}
{A(b,x),E(b,x),¬M(x)}
{¬A(y,x),¬D(x,y)}
{D(a,b)}
{O(a)}
{M(x),¬O(x)}
{x/a, y/b}
{x/a}
{x/a}
{¬A(b,a)}
{M(a)}
{E(b,a),¬M(a)}
{E(b,a)}
□
master*
Ln 150, Col 2
Spaces: 2
UTF-8
LF
Markdown
Go Live
Prettier

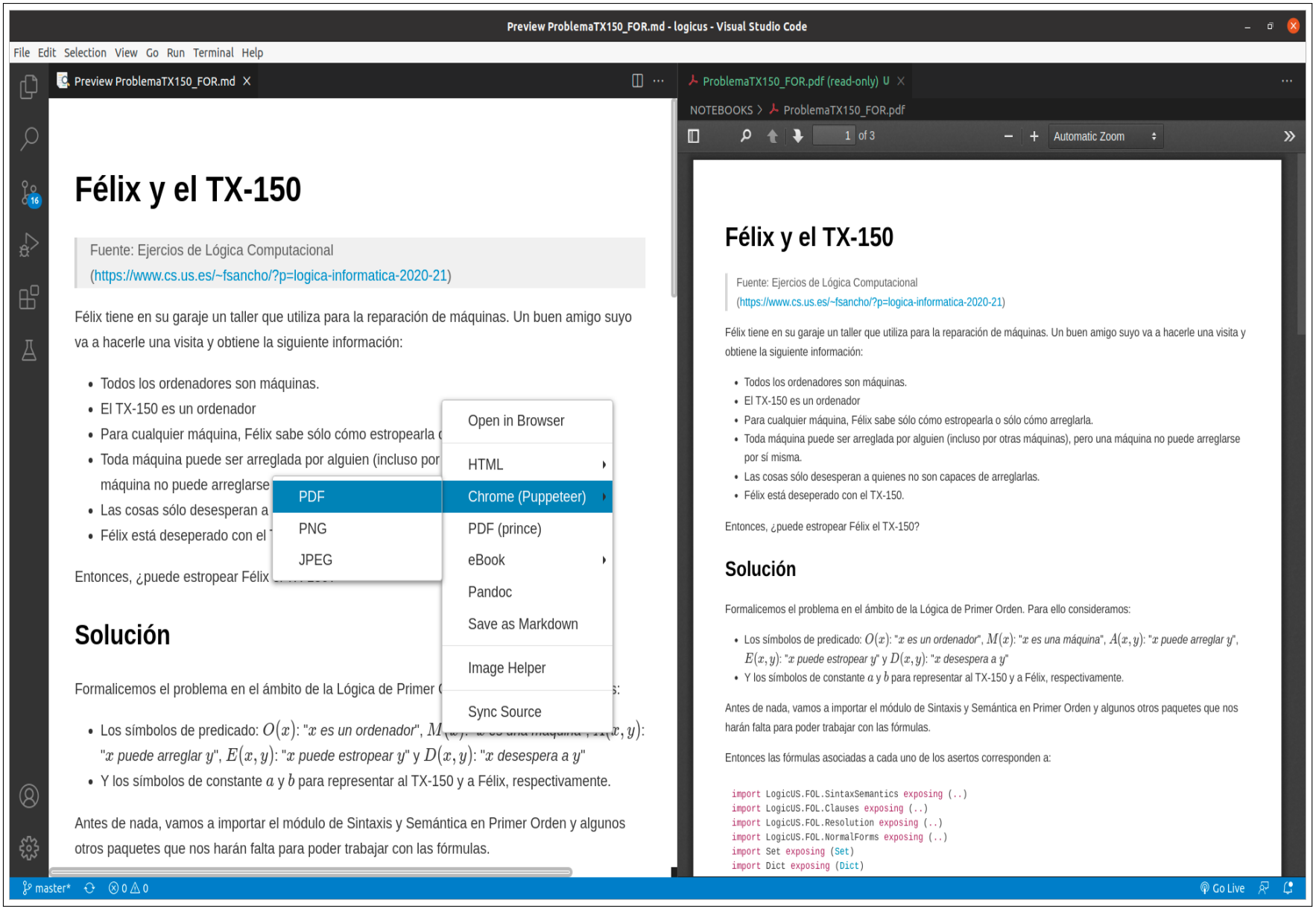


Figura 5.4: Ejemplo de uso de LogicUS + *litvis*
Fuente propia. Creada con litvis

6 | Conclusiones y Trabajo Futuro

Una vez presentado todo el trabajo realizado se va a realizar una valoración final del proyecto para poder, tomando perspectiva, conocer hasta qué punto se han conseguido los objetivos y requerimientos planteados inicialmente y poder plantear, así mismo, posibles líneas futuras de desarrollo que quedan abiertas y la proyección de la herramienta.

En efecto, el uso de la programación declarativa orientada a la Web con el lenguaje Elm, nos ha permitido diseñar e implementar un sólida librería que provee la mayoría de las funcionalidades que se consideraron en el primer momento. Si realizamos una visión global de los módulos implementados:

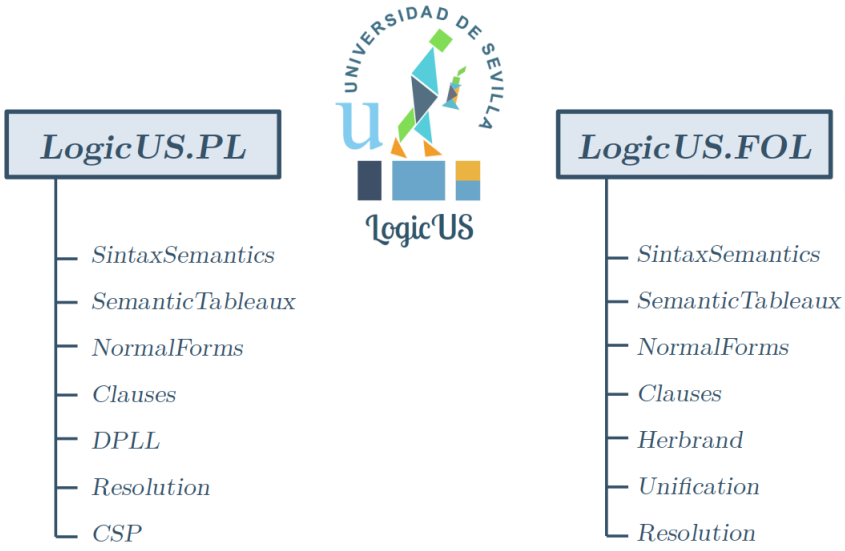


Figura 6.1: Paquete LogicUS
Fuente propia.
Disponible en https://package.elm-lang.org/packages/vicramgon/logicus/latest/

la librería cubre prácticamente los contenidos principales que se pretendían abordar al inicio del proyecto, permitiendo el trabajo con la Lógica Proposicional y la Lógica de Primer Orden, tanto en cuestiones comunes a ambas: representación de fórmulas (además de forma natural gracias a los parsers), la evaluación de dichas fórmulas según interpretaciones y L-estructuras, según el caso, el cálculo y representación de tableros semánticos, la transformación de fórmulas en formas normales y cláusulas; como de algoritmos específicos de cada una de ellas como son Tableros Semánticos, DPLL, Resolución Proposicional (junto con las distintas estrategias) y el tratamiento de fórmulas parametrizadas orientadas al modelado de Problemas de Satisfacción de Restricciones en LP, o la aplicación de los trabajos de Herbrand, la unificación o la Resolución no restringida en Primer Orden.

Si bien es cierto que quedan algunas cuestiones que abordar, como son el trabajo con Sistemas Deductivos en LP, o el trabajo con igualdad en LPO, la estructura modular de la herramienta y las estructuras diseñadas permiten que dichas cuestiones puedan ser completadas a posteriori, sin necesidad

de introducir cambios en las cuestiones ya presentes en el proyecto.

Además de todas las funciones orientadas al cálculo de los distintos algoritmos, se han diseñado en todos los aspectos funciones dedicadas a la representación para el trabajo en distintos ámbitos:

- Representación como cadena en formato UNICODE. Orientadas, principalmente, a un uso de representación en la consola de Elm (Elm REPL), de forma que puedan visualizarse de forma cómoda tanto las fórmulas y cláusulas como los desarrollos de los algoritmos o el contenido de cada una de las variables (tratadas por Elm como funciones).
- Representación como cadena en formato Latex. Orientada al uso en conjunción con *litvis* u otras librerías como *MathJax* o *Katex*, de cara a la creación de una interfaz Web.
- Representación como cadena en formato DOT que permite un procesamiento posterior para la representación como grafo.

Precisamente apoyándonos en *litvis*, conseguimos una herramienta totalmente funcional, con la que podemos generar documentos que incluyan contenido típico en *Markdown* (texto, fórmulas látex, tablas, imágenes, etc.) junto con bloques y resultados de la ejecución de Elm, cuyo formato podemos manejar con el uso de los parámetros de configuración de los chunks (mostrar el chunk, interpretar la salida en formato raw/markdown/json, ...) y permitir incluso la visualización de los diagramas básicos utilizando el formato DOT, al que hemos adaptado, precisamente, las representaciones de distintos algoritmos, a fin de permitir una total integración entre los distintos componentes del sistema.

Por ello, con las herramientas y sistemas provistos se dispone ya de una herramienta completa que cubre, prácticamente, la totalidad de los contenidos de la asignatura Lógica Informática, favoreciendo el desarrollo de contenidos prácticos, y sirviendo de apoyo a los alumnos que pueden buscar en la misma un sistema de ayuda a la hora de comprender el funcionamiento de los algoritmos y, en general, el proceso de resolución de los problemas.

En su conjunto encontramos el trabajo satisfactorio en su conjunto, considerando Elm un lenguaje ideal para su desarrollo, por su cercanía al lenguaje formal (dado su carácter funcional), por su potente y seguro sistema de detección de errores que permite detectar en tiempo real los errores y evitar que se trasladen, y su orientación Web permitiendo el desarrollo Web bajo el mismo entorno en el que se desarrollan los módulos de cálculo (back-end) y su capacidad de integración con JS, que lo hace potente y versátil. Además de todo ello, el poder de un repositorio con una gran cantidad de paquetes dedicados a muy distintas áreas facilita el desarrollo y su documentación establece una gran capacidad de uso y facilidad de integración de los mismos. En general la experiencia con este lenguaje ha sido excelente con solo algunas cuestiones puntuales como la incapacidad de definición de operadores infijos (que puede suplirse con comodidad a través de los Parsers, y de forma más personalizada) o un problema más importante y que sí creemos deberían tratar de resolver, correspondiente a la imposibilidad de definir tipos heredados de clases, principalmente *comparable* que imposibilita el trabajo más eficiente con conjuntos en vez de con listas. En cualquier caso, ninguno de ellos es drástico y pueden ser suplidos, de forma más o menos cómoda, por la correcta gestión del desarrollador.

En cuanto a *litvis* también lo vemos como una herramienta excelente aunque sí hemos observado (y así se le ha transmitido a los propios desarrolladores) la necesidad de integrar los bloques de visualización con los propios bloques Elm, de forma que, al igual que se puede elegir mostrar la salida en formato *raw* o *markdown*, se pudiese interpretar la salida en formato DOT, Tikz, ... Otro problema (aunque es usual en otros renderizadores markdown) es la necesidad de la caché que guarda resultados que pueden no ser mostrados posteriormentes de forma válida hasta que se realiza una limpieza de la misma... El resto del uso del sistema nos parece cómodo, intuitivo y versátil; convirtiéndose en una herramienta indispensable para nuestro proyecto.

En cuanto al propio desarrollo de LogicUS, creemos que las estructuras elegidas así como las decisiones de diseño llevadas a cabo han resultado satisfactorias y el sistema final cumple los objetivos propuestos. Sin embargo existen ciertos puntos que no han sido del todo satisfactorios y que intentarán mejorarse en futuras versiones, dado que nos gustaría que el proyecto no quedase anclado en este punto y pudiese

crecer en un plano futuro (cercano). De hecho, algunas de las cuestiones que han quedado pendientes y en las que se trabaja actualmente son:

- El desarrollo de módulos dedicados al trabajo con sistemas deductivos, cláusulas de Horn, encadenamiento, ...
- El desarrollo de módulos y funciones dedicadas al trabajo con la Lógica de Primer Orden con Igualdad y a la búsqueda de modelos (finitos) en Primer Orden.
- La investigación e implementación de heurísticas que permitan establecer un mejor comportamiento de los algoritmos, sobre todo en el trabajo con CSP y con la Lógica de Primer Orden.
- El desarrollo de una GUI basada en el uso de HTML + CSS + JS, que permita la resolución automática o guiada de ejercicios, basada en los principios de usabilidad y accesibilidad.

En este sentido, el proyecto puede tener otras muchas trayectorias de ampliación:

- Inclusión de nuevos módulos dentro de las Lógicas Proposicional y Primer Orden dedicados a otros aspectos fuera de la asignatura, pero también presentes en otras asignaturas de objetivos similares, como el trabajo con inconsistencia, la Deducción Natural, el trabajo con Lógicas de Primer Orden con Aritmética, ...
- El desarrollo de módulos y funciones dedicadas a otro tipo de lógicas, como puede ser la Lógica Difusa, Lógicas Modales, ...
- El desarrollo de módulos dedicados a la representación de soluciones de problemas CSP en modelos de planta,...
- Integración de los sistemas con sistemas potentes de resolución, aprovechando la posibilidad de comunicación aplicación-servidor que provee Elm.
- Y más...

Referencias

[1] R. C. Moore, *Logic and Representation*. USA: Center for the Study of Language and Information, 1995.

[2] "gicentre/litvis: Literate Visualization: Theory, software and examples." [Online]. Available: https://github.com/gicentre/litvis

[3] "Elm - A delightful language for reliable webapps." [Online]. Available: https://elm-lang.org/

[4] E. Czaplicki, "Elm: Concurrent FRP for Functional GUIs," Tech. Rep., 2012.

[5] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, feb 1984. [Online]. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/27.2.97

[6] J. Wood, A. Kachkaev, and J. Dykes, "Design Exposition with Literate Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 759–768, jan 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8440080/

[7] "litvis/documents at main · gicentre/litvis." [Online]. Available: https://github.com/gicentre/litvis/tree/main/documents

[8] "markdown-preview-enhanced-with-litvis/diagrams.md at main · gicentre/markdown-preview-enhanced-with-litvis." [Online]. Available: https://github.com/gicentre/markdown-preview-enhanced-with-litvis/blob/main/docs/diagrams.md

[9] "Home - TouIST." [Online]. Available: https://www.irit.fr/TouIST/

[10] J. Fairbank, *Programming Elm : build safe and maintainable front-end applications*. Raleigh, North Carolina: Pragmatic Bookshelf, 2019.

[11] "litvis/documents at main · gicentre/litvis." [Online]. Available: https://github.com/gicentre/litvis/tree/main/documents

[12] U. Nilsson and J. Maluszynski, *Logic, programming and prolog*. John Wiley & Sons, 1997.

[13] M. Leuschel, "Logic program specialisation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1706. Springer Verlag, 1999, pp. 155–188.

[14] J. ALonso Jiménez, J.A. Borrego Díaz, *Deducción automática*. Kronos, 2002.

[15] S. J. Russell, P. Norvig, M. C. R. Juan, and L. J. Aguilar, *Inteligencia artificial: un enfoque moderno*. Pearson Educacion, 2011.

[16] D. L. Poole, A. K. Mackworth, and R. G. Goebel, *Computational intelligence: a logical approach*. Oxford Univ. Press, 1998.

[17] K. Doets, *From logic to logic programming*. MIT Press, 1994.

[18] M. D. Davis, R. Sigal, and E. J. Weyuker, *Computability, Complexity, and Languages*. Academic Press, 1994.

[19] M. Ben-Ari, *Mathematical logic for computer science*. Springer, 2012.

[20] M. Huth and M. D. Ryan, *Logic in computer science: modelling and reasoning about systems*. Cambridge University Press, 2018.

[21] "Lógica Informática 2020-21 - Fernando Sancho Caparrini." [Online]. Available: https://www.cs.us.es/~fsancho/?p=logica-informatica-2020-21

[22] J. Fairbank, *Programming Elm: build safe and maintainable front-end applications*. Pragmatic Bookshelf, 2019.

[23] J. Fernandez, O. Gasquet, A. Herzig, D. Longin, E. Lorini, F. Maris, and P. Régnier, "TouIST: a Friendly Language for Propositional Logic and More TouIST: a Friendly Language for Propositional Logic and More TouIST: a Friendly Language for Propositional Logic and More," pp. 5240–5242, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02925894

[24] P. Blackburn, H. van Ditmarsch, M. Manzano, and F. Soler-Toscano, Eds., *Tools for Teaching Logic*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 6680. [Online]. Available: http://link.springer.com/10.1007/978-3-642-21350-2

[25] E. Paluzo Hidalgo, "Lógica de primer orden en Haskell," Universidad de Sevilla, Tech. Rep., 2017. [Online]. Available: http://hdl.handle.net/11441/63139