

UNIVERSIDAD DE LAS AMÉRICAS – PUEBLA

ESCUELA DE CIENCIAS

DEPARTAMENTO DE FÍSICA Y MATEMÁTICAS



“LÓGICA APLICADA A ANSWER SETS”

TESIS PROFESIONAL PRESENTADA POR,

Juan Antonio Navarro Pérez

COMO REQUISITO PARCIAL PARA
OBTENER EL TÍTULO DE
LICENCIADO EN MATEMÁTICAS

Sta Catarina Mártir, Cholula, Puebla.

Primavera de 2003

UNIVERSIDAD DE LAS AMÉRICAS – PUEBLA

ESCUELA DE CIENCIAS



“LÓGICA APLICADA A ANSWER SETS”

Tesis profesional presentada por,

Juan Antonio Navarro Pérez

Como requisito parcial para obtener el Título de
Licenciado en Matemáticas

JURADO CALIFICADOR

Andrés Ramos

PRESIDENTE

Maxim Todorov

SECRETARIO Y DIRECTOR

Mauricio Osorio Galindo

VOCAL Y CO-DIRECTOR

Sta Catarina Mártir, Cholula, Puebla.

Mayo de 2003

Este trabajo esta dedicado con especial cariño a mis padres, quienes siempre me apoyaron y han sido siempre una pieza fundamental en todos mis logros.

Muchas gracias también a mis hermanos y familiares por haberme motivado en todo momento. A mis amigos y compañeros con quienes compartí tantos momentos gratos.

Gracias a mis sinodales y todos mis profesores por su compromiso y extensa dedicación con la tarea de la enseñanza.

Gracias porque aprendí que la vida es mucho más que argumentos y demostraciones formales. Entendí la importancia de nuestras creencias y la fe que tenemos en ellas.

Índice general

1. Introducción	1
2. Marco Teórico	4
2.1. Conceptos Generales	4
2.1.1. El Enfoque de la Programación Lógica	4
2.1.2. Answer Sets y Modelos Estables	6
2.1.3. Notas y Consideraciones Preliminares	7
2.2. Lógica Proposicional	9
2.2.1. Lenguaje Formal	9
2.2.2. Lógicas Multivaluadas	10
2.2.3. Sistemas Axiomáticos	11
2.2.4. Lógicas Intermedias	13
2.2.5. Notación y Resultados Básicos	13
2.3. Programación Lógica	15
2.3.1. Clases de Programas Lógicos	15
2.3.2. Semánticas para Programas Lógicos	16
2.3.3. La Semántica de Answer Sets	17
3. Caracterización de Answer Sets	20
3.1. Resultados Preliminares	21
3.1.1. Lógicas Intermedias	21
3.1.2. Reducciones	23
3.2. Caracterizaciones	25
3.2.1. Modelos Mínimos	25
3.2.2. Answer Sets	26
3.2.3. Min-Sets	29
4. Traducciones	31
4.1. Nociones de Equivalencia	31
4.1.1. Equivalencia Fuerte	31
4.1.2. Extensión Conservativa	35
4.2. Traducciones de Programas	39

<i>ÍNDICE GENERAL</i>	II
4.2.1. Reducción de Implicaciones Anidadas	40
4.2.2. Teorías Proposicionales a Programas Generales	41
4.2.3. Eliminación de Restricciones	43
4.3. Propiedades de las Traducciones	44
5. Aplicaciones y Trabajo Futuro	47
5.1. Técnicas de Depuración	47
5.2. Algoritmo para calcular Answer Sets	48
5.3. Inferencia Modal No Monótona	49
6. Conclusiones	51

Capítulo 1

Introducción

La programación lógica (Logic Programming, LP) ofrece una manera natural de representar conocimiento declarativo utilizando el lenguaje de la lógica matemática. Hechos y relaciones entre diversos objetos pueden ser capturados en un programa lógico para que, con la ayuda de herramientas computacionales, seamos capaces de contestar preguntas sobre los objetos descritos y encontrar soluciones a problemas particulares.

La semántica de answer sets, introducida por Gelfond y Lifschitz [13] con el nombre de modelos estables y generalizada después por Lifschitz, Tang y Turner [18], representa la acumulación de una buena cantidad de avances y conocimientos en razonamiento no-monótono, sistemas autónomos de agentes y aplicaciones de inteligencia artificial.

La lista de aplicaciones prácticas y casos exitosos para resolver problemas de la vida real ha crecido de manera impresionante en los últimos años. Destacan, principalmente, aplicaciones para resolver problemas combinatorios, proveer esquemas de planeación y el modelado de agentes lógicos por mencionar algunas de las aplicaciones típicas.

El objetivo principal de este trabajo se centra en el estudio de propiedades y conceptos de la semántica de los answer sets, y algunas otras semánticas cercanas, mediante el uso de lógicas proposicionales. Ofrecemos un nuevo enfoque hacia la teoría de programación con answer sets a través de diversas herramientas y relaciones con la lógica intuicionista, así como la amplia gama de lógicas intermedias.

Este trabajo consiste, de hecho, en la recopilación, revisión y reorganización de diversos resultados y avances que, siguiendo esta línea de investigación, hemos presentado ya en diversos artículos de talleres, congresos y revistas en los últimos dos años. La contribución principal de este trabajo es entonces, el ofrecer un panorama amplio y detallado de todos los resultados y las nociones propuestas en estos trabajos previos.

Uno de los primeros resultados importantes, presentado en [29], establece una caracterización de la semántica de los answer sets en términos de la lógica intuicionista. Esta caracterización, en prosa, establece que

Una fórmula es consecuencia lógica de un programa lógico en la semántica de answer sets si y solo si puede ser demostrada en cada extensión consistente e

intuicionísticamente completa formada agregando átomos negados y doblemente negados al programa.

Este resultado sigue la línea de investigación iniciada por Pearce en [33], donde él propone una caracterización similar para el caso restringido de programas disyuntivos. Otro resultado similar fue presentado en [17] en términos de las llamadas “lógicas de equilibrio”. Utilizando este tipo de enfoques podemos no sólo obtener resultados más generales, sino además estudiar y analizar otras posibilidades al considerar, por ejemplo, otras variedades de lógicas en el enunciado de la caracterización.

Los resultados obtenidos nos permiten, además, definir la noción de inferencia no-monótona para cualquier teoría proposicional (con los conectivos usuales \wedge , \vee , \neg , \rightarrow) en términos de una lógica monotónica. Obtenemos también como consecuencia directa un esquema que nos permite distinguir nociones como *conocimiento* y *creencia*. Otro de los resultados, presentado en [28], muestra que la lógica intuicionista utilizada en esta caracterización puede ser reemplazada por cualquier otra lógica intermedia sin alterar la validez del resultado.

También en [29] estudiamos los modelos mínimos y su relación con la semántica de answer sets. A partir de los resultados obtenidos también se definió y caracterizó la semántica de los *min-sets* que considera a los answer sets que son, además, modelos mínimos. Diversas aplicaciones teóricas y prácticas, ver por ejemplo [5, 14, 16, 19, 22], justifican el interés en el estudio de los modelos mínimos.

Otra de las nociones importantes que hemos estudiado es la equivalencia entre programas lógicos. Consideramos, por ejemplo, la definición de equivalencia fuerte presentada en [17]. La equivalencia fuerte se relaciona con un importante concepto de la ingeniería de software y es que, si dos programas lógicos son fuertemente equivalentes, podemos reemplazar localmente uno por el otro dentro de cualquier programa más grande.

Los mismos autores de [17] ofrecieron una caracterización de equivalencia fuerte en términos de la lógica de 3-valores G_3 . En [23] se demostró que esta misma relación se sigue satisfaciendo para teorías proposicionales arbitrarias y no solo para clases restringidas de programas lógicos. Se probó también en [29] que la equivalencia fuerte en el caso de los min-sets queda caracterizada por la lógica G_3 .

La extensión conservativa, presentada por Baral en [3], es una versión un poco más relajada de equivalencia. Un programa se considera extensión conservativa de otro si sus modelos, restringidos al lenguaje del segundo programa, son exactamente los mismos. Lo que permite la extensión conservativa es agregar nuevos elementos al lenguaje y, posiblemente, lograr simplificar la estructura de los programas. Una extensión de este tipo es válida siempre que, ignorando los nuevos elementos agregados al lenguaje, no se alteran los modelos capturados por la semántica.

En [29] presentamos una primera definición de lo que llamamos una extensión conservativa fuerte que, además de permitir extensiones en el lenguaje de los programas, preserva todas las propiedades importantes de la equivalencia fuerte como el hecho de poder hacer reemplazos locales. Estudiamos de nuevo este concepto en [24] donde se analizaron,

además, diversas propiedades importantes para traducciones de programas lógicos basadas en el concepto de la extensión conservativa.

En otros trabajos se estudiaron también traducciones que se pueden hacer para simplificar la estructura y la complejidad de los programas lógicos. En diversas aplicaciones de planeación, diagnóstico y razonamiento, como por ejemplo [4, 9, 12], se proponen diversos algoritmos donde, para modelar situaciones dinámicas, se deben calcular sucesivamente los answer sets de diversos programas lógicos.

Una característica recurrente en este tipo de aplicaciones es que la serie de programas lógicos, para los cuales se deben encontrar sus answer sets, tienen siempre una gran parte común. De un programa a otro sólo se modifican unas cuantas reglas que sirven para modelar el comportamiento dinámico del sistema. En [26] presentamos una serie de traducciones que podrían aplicarse a este gran bloque común de los programas para simplificar su estructura y hacer más eficiente el proceso evitando recálculo.

Tomando en cuenta las propiedades de equivalencia, extensiones conservativas y sus caracterizaciones en términos de la lógica G_3 podemos estar seguros de que las transformaciones propuestas pueden, efectivamente, aplicarse de manera local a sectores particulares del programa sin modificar su semántica declarativa. Usando las traducciones propuestas en [26], complementadas con una traducción adicional de [24], es posible traducir teorías proposicionales arbitrarias a la clase de programas disyuntivos simples.

Este tipo de resultados nos permiten, por ejemplo, calcular los answer sets de cualquier programa proposicional traduciéndolo a uno disyuntivo y utilizando alguna de las herramientas existentes para calcular answer sets en esta clase de programas.

La estructura de la presentación de este trabajo es la siguiente: en el Capítulo 2 se encuentra el marco teórico de la investigación, presentamos primero de manera informal algunos de los conceptos e ideas principales para introducir después las definiciones y notaciones formales utilizadas en el trabajo; en el Capítulo 3 se recopilan y reestructuran varios de los resultados presentados en [28, 29] relativos a la caracterización de answer sets utilizando lógicas intermedias, se caracterizan también las semánticas de modelos mínimos y de min-sets; en el Capítulo 4 presentamos primero un estudio completo de las nociones de equivalencia y las extensiones conservativas rescatando resultados de [23, 24, 26, 29], presentamos y revisamos después diversas traducciones, principalmente de [24, 26], que nos permiten simplificar la estructura de los programas lógicos; finalmente, en el Capítulo 6, presentamos las conclusiones del trabajo y posibles líneas de investigación, como las discutidas en [25, 27, 30], para continuar el desarrollo de la teoría aquí presentada.

Capítulo 2

Marco Teórico

El objetivo de este capítulo es presentar un panorama general de las ideas y conceptos principales con los que se trata en este trabajo de investigación. Se presentan primero, informalmente, algunos conceptos generales a cerca de la programación lógica, la semántica de modelos estables y los answer sets, así como unas primeras consideraciones teóricas antes de iniciar la presentación formal.

Después presentamos las definiciones formales del lenguaje de la programación lógica, lógicas basadas en tablas de verdad multivaluadas y sistemas axiomáticos. Definimos y revisamos algunas propiedades simples de las lógicas intermedias. Esta presentación no es totalmente exhaustiva pero sí bastante clara y detallada, haciendo énfasis especial en las propiedades que serán importantes en la discusión posterior.

La última de las secciones presenta diversas clases de programas lógicos que se encuentran en la literatura, se define también formalmente lo que es una semántica y se presentan después las definiciones oficiales de las semánticas de modelos mínimos y answer sets. Mediante un ejemplo se muestra también como calcular los answer sets de un programa particular.

2.1. Conceptos Generales

En esta primera sección presentaremos de manera informal, más bien intuitiva, varios de los conceptos importantes con los que se relaciona este trabajo de investigación. Presentamos, de una manera muy general, el enfoque que propone el paradigma de la programación lógica. Describimos también los orígenes y las cualidades de la semántica de answer sets, así como algunas notas y consideraciones generales necesarias antes de hacer una presentación formal de estos temas.

2.1.1. El Enfoque de la Programación Lógica

El modelo usado tradicionalmente para enfrentar un problema con ayuda de la computadora consiste, primero, en realizar un análisis y estudio del problema para, después,

proponer un algoritmo que, ejecutado paso a paso, sea capaz de encontrar las soluciones del problema en cuestión. La computadora, en este caso, es utilizada simplemente como una herramienta de cálculo que realiza, de manera más rápida y eficiente, las operaciones y cálculos requeridos por el algoritmo propuesto.

La programación lógica ofrece un modelo alternativo para enfrentar y tratar de solucionar problemas auxiliados por la computadora. Este modelo, lo que pretende, es proveer a la computadora de una descripción del problema que tratamos y no propiamente de un mecanismo para construir soluciones. La computadora será entonces la responsable de analizar todos los casos posibles y determinar cuales de ellos representan soluciones al problema descrito.

El esquema de la programación lógica necesita entonces de un lenguaje que nos permita *describir* problemas a la computadora. Este lenguaje debe ser claro y específico, falto de ambigüedad. Además debe ser capaz de expresar construcciones matemáticas típicas. El lenguaje que, por supuesto, mejor se adapta a estos requerimientos es el lenguaje de la lógica matemática. Uno de los ejemplos más comunes para mostrar las posibles aplicaciones de la programación lógica es el problema de la coloración de un mapa.

Ejemplo 2.1. La siguiente es una variante típica del problema de colorear un mapa que puede describirse fácilmente mediante las siguientes reglas:

1. Cada ciudad debe ir coloreada por alguno de los colores: rojo, verde o azul.
2. No puede ocurrir que dos ciudades adyacentes tengan el mismo color.

Este mismo par de reglas se pueden escribir usando la sintaxis de los programas lógicos de la siguiente forma:

$$\begin{aligned} & \text{color}(X, \text{rojo}) \vee \text{color}(X, \text{verde}) \vee \text{color}(X, \text{azul}) \leftarrow \text{ciudad}(X). \\ & \leftarrow \text{color}(X, C) \wedge \text{color}(Y, C) \wedge \text{adyacente}(X, Y). \end{aligned}$$

El problema requiere además, por supuesto, de una descripción de la instancia particular de un mapa. Dicha descripción debe contener reglas de la forma $\text{ciudad}(x)$ y $\text{adyacente}(x, y)$ donde las variables x y y se pueden reemplazar, por ejemplo, por los nombres de las ciudades en el mapa.

Observe cuan fiel es la descripción del problema en el programa lógico del ejemplo anterior comparada con la que fue hecha primero en lenguaje natural. La descripción es, además, muy clara y concisa. Una solución algorítmica para este mismo problema requeriría, cuando menos, de una veintena de instrucciones; así como de un análisis previo de las propiedades y cualidades especiales de las soluciones del problema.

El modelo propuesto por la programación lógica presenta varias ventajas. Además de la claridad con que pueden ser descritos los problemas, nos ahorra el tener que realizar un estudio minucioso del problema antes de poder enfrentarlo. Otra de las ventajas que ofrece es que, en aplicaciones reales, las condiciones o restricciones para un problema pueden cambiar de un momento a otro. En el caso de la programación lógica un cambio de esta naturaleza sólo representa una modificación en la descripción del problema. Mientras que, en

el caso procedural, una de estas actualizaciones podría modificar alguna de las propiedades esenciales del problema y haciendo necesaria, incluso, la reescritura completa de un nuevo algoritmo para resolver el nuevo problema.

Una de las ventajas que, por su parte, mantiene el enfoque procedural es, por supuesto, la eficiencia y la rapidez con que se pueden encontrar soluciones. Es de esperarse que si proveemos a la computadora de métodos y construcciones específicas para encontrar las soluciones de un problema en particular, se puedan obtener mejoras significativas en la eficiencia. Muchos de los proyectos y avances recientes en la programación lógica consiste en implementar modelos híbridos que rescaten las mejores características de ambos enfoques.

2.1.2. Answer Sets y Modelos Estables

Para que un programa lógico tenga un significado y pueda ser interpretado por herramientas computacionales se le debe asignar una semántica. Una semántica es, intuitivamente, una manera de determinar el tipo de conclusiones que se pueden establecer a partir de un conjunto de reglas. Un problema debe codificarse entonces, en la forma de un programa lógico, buscando que la semántica del programa capture, precisamente, las soluciones del problema en cuestión.

La semántica de modelos estables (stable models), introducida por Gelfond y Lifschitz en [13], significó un importante adelanto en el área de la programación lógica. Una de las principales novedades de la semántica de modelos estables fue la facilidad con que permitía modelar el concepto de *negación como falla* en los programas lógicos. Este tipo de negación facilita la especificación de *conocimiento por omisión* (default knowledge) y establecer relaciones inerciales que son fundamentales para modelar sistemas dinámicos.

Los modelos estables, como fueron definidos en [13], proveen de un significado semántico a los programas lógicos que, sin embargo, están condicionados a una sintaxis particular muy restringida. La semántica de modelos estables consideraba, únicamente, reglas de la forma:

$$h_1 \vee \cdots \vee h_n \leftarrow b_1 \wedge \cdots \wedge b_m \wedge \neg b_{m+1} \wedge \cdots \wedge \neg b_{m+l}$$

donde cada h_i y b_i es una fórmula atómica con $n \geq 0$, $m \geq 0$ y $l \geq 0$. Este tipo de reglas, aunque representan sólo una clase restringida de programas lógicos, probaron ser de gran utilidad para poder enfrentar y resolver una amplia cantidad de problemas.

La semántica de answer sets, propuesta por Lifschitz, Tang y Turner en [18], generaliza la noción de modelos estables para una clase de programas más amplia. Los programas lógicos que se consideran en esta semántica permiten reglas de la forma $H \leftarrow B$, donde el par de fórmulas H y B pueden estar compuestas de conjunciones, disyunciones y negaciones arbitrariamente anidadas. En [11] se muestran evidencias claras de cómo este lenguaje puede ser de gran utilidad para representar y resolver problemas reales.

Así surgió la Programación con Answer Sets (Answer Set Programming, ASP) o también conocida como A-Prolog. Este paradigma representa la acumulación de una importante cantidad de avances en razonamiento no-monótono, inteligencia artificial y programación lógica durante los últimos 15 años.

El éxito de la semántica de los modelos estables, y posteriormente de los answer sets, se debió en gran medida a su capacidad para resolver problemas. Gracias a la existencia de implementaciones eficientes para calcular modelos estables como `dlv`¹ y `smodels`² la gama de problemas que podían enfrentarse con este nuevo paradigma creció rápidamente para incluir problemas combinatorios, establecer y resolver problemas de planeación, modelar el comportamiento de agentes lógicos y aplicaciones de inteligencia artificial en general.

Sin embargo ocurre que, de un modo ciertamente sorprendente, las primeras relaciones entre la lógica proposicional y la programación lógica estaban simplemente restringidas al uso de un lenguaje común. Las definiciones de las primeras semánticas, aunque ciertamente están basadas en nociones de la lógica matemática, están más bien propuestas a partir de métodos que simplemente *parecen razonables* para asignar modelos a un programa lógico dado. El estudio de propiedades o caracterizaciones de la semántica en términos de lógicas proposicionales es un tema de investigación mucho más reciente.

2.1.3. Notas y Consideraciones Preliminares

Una de las características innovadoras que presenta la semántica de answer sets es, precisamente, el uso de dos tipos de negaciones distintas. La negación *clásica* o *explícita*, que denotaremos con el conectivo \sim , permite especificar explícitamente conocimiento que sabemos de antemano es falso. Mientras que la negación *como falla*, denotada como \neg , tiene un significado un tanto distinto.

La intuición la negación como falla es que una fórmula $\neg F$ es cierta si, con la información disponible, no es posible mostrar que la fórmula F sea cierta. Observe que el no poder mostrar que un hecho es cierto no implica necesariamente que éste sea falso. Puede ocurrir que, por ejemplo, el hecho sea en efecto cierto pero la información disponible no sea suficiente para demostrarlo. Entre las ventajas que ofrece la negación como falla está, como lo habíamos mencionado antes, la facilidad con que se puede expresar conocimiento por omisión.

Ejemplo 2.2. Una de las aplicaciones donde se utilizan ambos tipos de negaciones es para describir situaciones que *normalmente* son ciertas, pero que pueden tener excepciones. Podemos escribir, por ejemplo, que “normalmente las aves vuelan” utilizando la regla:

$$\text{vuela}(X) \leftarrow \text{ave}(X) \wedge \neg \sim \text{vuela}(X).$$

De manera un poco más formal esta regla dice: “Si X es un ave y, con la información disponible, no es posible determinar que no vuela entonces asumiremos que sí puede volar”. Supongamos que tenemos la siguiente información acerca de algunas aves en nuestra base de conocimientos:

¹<http://www.dbai.tuwien.ac.at/proj/dlv/>.

²<http://www.tcs.hut.fi/Software/smodels/>.

```

ave(pato).
ave(pelícano).
ave(pingüino).
~vuela(pingüino).

```

El sistema podrá inferir correctamente que tanto el pato como el pelícano, como son aves, pueden volar. Sin embargo el sistema no tratará de usar el mismo razonamiento para demostrar que los pingüinos vuelan pues ya que sabe, de manera explícita, que no lo hacen.

Es importante hacer la aclaración de que, en diversas fuentes importantes en la literatura, es común encontrar al símbolo *not* para denotar a la negación como falla (aquí denotada como \neg) y al símbolo \sim para la negación explícita (aquí denotada \sim). La notación que aquí empleamos sigue de una corriente iniciada por trabajos recientes que buscan establecer relaciones entre las lógicas proposicionales y la propia programación lógica. En este caso resulta que la negación \neg utilizada comúnmente en la lógica proposicional coincide precisamente con la negación como falla de la programación lógica.

El uso de la negación explícita \sim en la semántica de answer sets es, de hecho, muy restringido. Sólo se permiten ocurrencias de esta negación dentro de fórmulas atómicas. Además de que no juega un papel realmente significativo en la definición de la semántica. En el Ejemplo 2.2 podríamos cambiar la expresión \sim vuela por un átomo nuevo: **no-vuela**, sin alterar el significado semántico del programa.

Es por esto que no consideraremos en la discusión principal de este trabajo a programas que hagan uso de la negación explícita. Sí discutiremos como reintegrar este tipo de negación dentro del esquema que proponemos e incluso, gracias a los resultados que proponemos, podremos considerar un nuevo esquema que permite el uso de la negación explícita sin ninguna restricción sintáctica. La negación explícita podrá usarse en este esquema no sólo para negar expresiones atómicas sino fórmulas arbitrarias en general.

Otra de las restricciones sintácticas necesarias para definir la semántica de answer sets, igual que en el caso de los modelos estables, es eliminar los símbolos funcionales del lenguaje. Esto es necesario porque, utilizando dominios infinitos, los answer sets ya no son necesariamente recursivamente numerables [20].

También, como es común en la literatura, se consideran únicamente teorías proposicionales finitas. Los programas con variables, como los mostrados en los Ejemplos 2.1 y 2.2, pueden incluirse en este esquema mediante un proceso de instanciación. Cada una de las reglas con variables es reemplazada por todas sus posibles instancias cambiando estas variables por los posibles valores constantes. Así, por ejemplo, el programa del Ejemplo 2.2 sería traducido a:

```

ave(pato).
ave(pelícano).
ave(pingüino).
~vuela(pingüino).
vuela(pato) ← ave(pato) ∧ ¬~vuela(pato).
vuela(pelícano) ← ave(pelícano) ∧ ¬~vuela(pelícano).
vuela(pingüino) ← ave(pingüino) ∧ ¬~vuela(pingüino).

```

Así, mediante este proceso de instanciación podemos restringir nuestra atención, cuando estudiamos la teoría de los answer sets, a programas lógicos en el caso proposicional.

2.2. Lógica Proposicional

En esta sección presentamos los conceptos y definiciones que requerimos sobre la lógica proposicional. Introducimos primero el lenguaje formal con el que se construyen las fórmulas lógicas y, después, discutimos brevemente algunas de las lógicas que, propiamente, se utilizan para dar significado a este lenguaje formal. Un tratamiento más completo se puede encontrar en referencias como [21, 39].

2.2.1. Lenguaje Formal

Consideramos formalmente un lenguaje provisto del siguiente alfabeto: un conjunto numerable \mathcal{L} cuyos elementos son llamados *átomos*; los conectivos $\wedge, \vee, \rightarrow, \perp$; y los símbolos auxiliares $(,)$. Los átomos en \mathcal{L} se denotan usualmente con letras minúsculas: a, b, c, \dots ; también se podrán usar nombres cortos como: `color`, `nodo`, etc. cuando se trate de átomos en algún contexto particular.

Los conectivos \wedge, \vee y \rightarrow son binarios, mientras que \perp es de orden cero, es decir no se usa para conectar fórmulas. El conector \perp es, por sí mismo, una fórmula. Las fórmulas se definen como es usual en lógica, formalmente consideramos el conjunto de *fórmulas de la lógica proposicional* –al que denotamos \mathcal{P} – que se define recursivamente como sigue:

1. $\perp \in \mathcal{P}$.
2. si $a \in \mathcal{L}$ entonces $a \in \mathcal{P}$.
3. si $F, G \in \mathcal{P}$ entonces $(F \wedge G), (F \vee G), (F \rightarrow G) \in \mathcal{P}$.
4. una expresión está en \mathcal{P} sólo si puede ser mostrado con las condiciones 1–3.

Otras notaciones se permiten como abreviación de las fórmulas proposicionales recién definidas. El símbolo \top se introduce, en particular, como una abreviación de $(\perp \rightarrow \perp)$; la fórmula $\neg F$ abrevia a $(F \rightarrow \perp)$; y $(F \leftrightarrow G)$ como abreviación de $((F \rightarrow G) \wedge (G \rightarrow F))$. La notación $(F \leftarrow G)$ es tan sólo una manera alternativa de escribir la fórmula $(G \rightarrow F)$.

Cuando no haya riesgo de ambigüedad podemos, manteniendo convenciones sobre la precedencia de los conectivos, eliminar paréntesis. Así, definimos al conector \neg como el de mayor precedencia, siguen los conectivos \wedge, \vee en el nivel medio y, finalmente, los de menor precedencia son \rightarrow, \leftarrow y \leftrightarrow . Los conectivos en una misma jerarquía se agrupan, en el orden en que aparecen, de izquierda a derecha.

Ejemplo 2.3. Presentamos algunos ejemplos sencillos de abreviaturas y las fórmulas que éstas representan:

abreviatura	fórmula
$(a \wedge b \wedge c) \rightarrow (a \vee b \vee c)$	$((a \wedge b) \wedge c) \rightarrow ((a \vee b) \vee c)$
$p \leftarrow a \wedge \neg q$	$((a \wedge (q \rightarrow \perp)) \rightarrow p)$
$\neg(a \wedge \neg b) \leftrightarrow (p \vee q)$	$((((a \wedge (b \rightarrow \perp)) \rightarrow \perp) \rightarrow (p \vee q))$ $\wedge ((p \vee q) \rightarrow ((a \wedge (b \rightarrow \perp)) \rightarrow \perp)))$

Una *teoría* es, en general, un conjunto de fórmulas. Sin embargo, para los propósitos particulares de este trabajo, consideraremos únicamente teorías finitas. Si T es una teoría definimos el *lenguaje* de T , denotado \mathcal{L}_T , como el conjunto de átomos que ocurren en T . Observe que, si asumimos que T es finita, siempre se tiene que \mathcal{L}_T es finito.

Una *literal* puede ser un átomo a (una literal positiva) o un átomo negado $\neg a$ (una literal negativa). Dada una teoría T definimos también la *teoría negada* $\neg T = \{\neg F \mid F \in \Gamma\}$.

2.2.2. Lógicas Multivaluadas

Las lógicas multivaluadas surgen como una generalización de las tablas de verdad utilizadas para definir a la lógica clásica. De particular interés para los propósitos de este trabajo son las lógicas multivaluadas G_i , con valores en de verdad en el conjunto $\{0, 1, \dots, i-1\}$, definidas por Gödel.

En la lógica G_i el valor de verdad $i-1$ se conoce como el *valor designado*. Una *interpretación* en G_i , es una función $I: \mathcal{L} \rightarrow \{0, 1, \dots, i-1\}$ que asigna un valor de verdad a cada átomo en el lenguaje. El dominio de esta función se extiende al conjunto completo de fórmulas proposicionales propagando la evaluación sobre sus conectivos:

- $I(A \rightarrow B) = i-1$ si $I(A) \leq I(B)$ y $I(A \rightarrow B) = I(B)$ en otro caso.
- $I(A \vee B) = \max(I(A), I(B))$.
- $I(A \wedge B) = \min(I(A), I(B))$.
- $I(\perp) = 0$.

Recuerde que \top , $\neg A$ y $A \leftrightarrow B$ fueron introducidos como abreviaciones. A partir de las definiciones anteriores podemos determinar también como se propaga la evaluación de fórmulas sobre estos conectivos:

- $I(\top) = i-1$
- $I(\neg A) = i-1$ si $I(A) = 0$ y $I(\neg A) = 0$ en otro caso.
- $I(A \leftrightarrow B) = i-1$ si $I(A) = I(B)$ y $I(A \leftrightarrow B) = \min(I(A), I(B))$ en otro caso.

Decimos que una interpretación es *definida* si asigna únicamente valores 0 ó $i-1$ a los átomos del lenguaje. Decimos que es *indefinida* si existe algún átomo al que se le asigne un valor intermedio.

Dada una interpretación I y una fórmula F decimos que I es un *modelo* de F si es de que I evalúa a F al valor designado, es decir $I(F) = i-1$. De manera similar podemos

decir que I es un *modelo* de la teoría T si I es un modelo de cada fórmula contenida en T . Equivalentemente podríamos definir $I(T) = \min_{F \in T} I(F)$ de modo que I es un modelo de la teoría T si $I(T) = i - 1$. Si una fórmula F es modelada por todas las interpretaciones (asignaciones de valores) posibles decimos que F es una *tautología*.

Observe que, en particular, G_2 corresponde exactamente a la lógica clásica usual. En la lógica clásica, que también denotamos C , las interpretaciones suelen relacionarse con subconjuntos del lenguaje. Es decir, a cada conjunto $M \subset \mathcal{L}$ se le asocia la interpretación que evalúa $M(a) = 1$ cuando $a \in M$ y $M(a) = 0$ si $a \notin M$. Nos tomamos la libertad de decir que un conjunto de átomos $M \subset \mathcal{L}$ es un *modelo clásico* de la teoría T , y se denota como $M \models T$, si la interpretación asociada es, en efecto, un modelo de T respecto a la lógica clásica.

La lógica G_3 , conocida también como HT o lógica de “Here and There”, será de especial utilidad en algunos de los resultados que presentamos más adelante. El nombre “Here and There” viene de una representación de la lógica G_3 en términos de semánticas de Kripke con dos mundos: “Aquí y Allá”.

Ejemplo 2.4. Las siguientes tablas muestran que, por ejemplo, algunas tautologías clásicas como $\neg\neg a \rightarrow a$ y $a \vee \neg a$ no son tautologías en la lógica G_3 .

$\neg\neg a \rightarrow a$	$a \vee \neg a$
0 2 0 2 0	0 2 2 0
2 0 1 1 1	1 1 0 1
2 0 2 2 2	2 2 0 2

Gödel definió también, a partir de las lógicas G_i , una lógica conocida como G_∞ . Una fórmula F es un *teorema* de la lógica G_∞ si, para toda i en $\{2, 3, \dots\}$, la fórmula F es una tautología en la lógica G_i .

2.2.3. Sistemas Axiomáticos

La lógica intuicionista, denotada I , surge como una alternativa a la lógica clásica. Pretende explicar el significado de los conectivos lógicos en términos de *conocimiento* o *demostrabilidad*, y no buscando una *verdad* o *certeza* absoluta como se hace en la lógica clásica de dos valores.

Esta lógica puede ser definida utilizando semánticas de Kripke o a partir de métodos de deducción natural [37, 38]. Nosotros utilizaremos una presentación de la lógica intuicionista basada en los sistemas axiomáticos de Hilbert que presentamos a continuación de manera formal. La lógica intuicionista se define a partir de los siguientes diez esquemas de axiomas:

- A1. $A \rightarrow (B \rightarrow A)$.
- A2. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$.
- A3. $A \wedge B \rightarrow A$.
- A4. $A \wedge B \rightarrow B$.
- A5. $A \rightarrow (B \rightarrow (A \wedge B))$.
- A6. $A \rightarrow (A \vee B)$.
- A7. $B \rightarrow (A \vee B)$.
- A8. $(A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow (A \vee B) \rightarrow C)$.
- A9. $(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A)$.
- A10. $\neg A \rightarrow (A \rightarrow B)$.

Recuerde que estos son “esquemas” de axiomas, y no axiomas propiamente. Un axioma se obtiene reemplazando las fórmulas A , B y C —que se dejaron indicadas en los esquemas— por fórmulas proposicionales. Así tenemos, por ejemplo, que la fórmula $a \wedge (b \vee c) \rightarrow (b \vee c)$ es una instancia del esquema de axioma 4. La lógica intuicionista utiliza además, como regla de inferencia, la regla de *modus ponens*. Esta regla permite obtener B si es que ya se tienen las fórmulas A y $A \rightarrow B$.

Una *prueba* para la fórmula F es una secuencia finita de fórmulas F_1, \dots, F_n donde la fórmula $F_n = F$ y cada F_k (con $1 \leq k \leq n$) es: (i) la instancia de un axioma, o bien (ii) se obtiene por una aplicación de modus ponens sobre dos fórmulas F_i, F_j con $i, j < k$ (es decir, si $F_j = F_i \rightarrow F_k$). Entonces decimos que F es un *teorema (intuicionista)* si existe una prueba, según la definición anterior, para F .

Ejemplo 2.5. Este es un ejemplo de la prueba del teorema $a \rightarrow a$:

- | | | |
|----|---|----------|
| 1. | $(a \rightarrow ((a \rightarrow a) \rightarrow a)) \rightarrow ((a \rightarrow (a \rightarrow a)) \rightarrow (a \rightarrow a))$ | Axioma 2 |
| 2. | $a \rightarrow ((a \rightarrow a) \rightarrow a)$ | Axioma 1 |
| 3. | $(a \rightarrow (a \rightarrow a)) \rightarrow (a \rightarrow a)$ | M.P. 1–2 |
| 4. | $a \rightarrow (a \rightarrow a)$ | Axioma 1 |
| 5. | $a \rightarrow a$ | M.P. 3–4 |

Se pueden construir sistemas axiomáticos similares a este para proveer definiciones alternativas para las lógicas que ya conocemos. Por ejemplo, si agregamos a los axiomas de la lógica intuicionista el nuevo esquema de axioma:

$$\text{A11. } (\neg A \rightarrow A) \rightarrow A.$$

obtenemos un sistema axiomático de prueba para la lógica clásica. O bien, si agregamos el esquema de axioma:

$$\text{A11'. } (\neg B \rightarrow A) \rightarrow (((A \rightarrow B) \rightarrow A) \rightarrow A).$$

lo que se obtiene es la definición de la lógica Sm, una representación más para la lógica que conocemos como G_3 o HT.

2.2.4. Lógicas Intermedias

Gödel mismo demostró que existe, al menos, una cantidad numerable de lógicas situadas entre la clásica y la intuicionista. Formalmente se tiene la siguiente proposición, donde el símbolo \subset denota la inclusión propia del conjunto de fórmulas demostrables (teoremas o tautologías) en cada lógica.

Proposición 2.1. [39] $I \subset G_\infty \subset \dots \subset G_{i+1} \subset G_i \subset \dots \subset G_3 \subset G_2 = C$.

Llamaremos *lógicas intermedias* a todas aquellas lógicas cuyo conjunto de fórmulas demostrables se encuentre entre la lógica intuicionista y la lógica clásica (inclusive). Una *lógica intermedia propia* es una lógica intermedia que no es la lógica clásica.

Un resultado importante también es que la lógica G_3 es, precisamente, la lógica intermedia propia más fuerte. Es decir, entre la lógica clásica y la lógica de 3 valores G_3 no se puede situar ninguna otra lógica (conjunto de fórmulas cerrado bajo modus ponens).

Como consecuencia de este resultado se podría pensar que la lógica intuicionista es menos poderosa que la lógica clásica pues, en efecto, prueba “menos” cosas. Sin embargo, mediante un resultado de Gödel, es posible recuperar a la lógica clásica dentro de la intuicionista.

Proposición 2.2. [21] *La fórmula F es una tautología clásica si y solo si $\neg\neg F$ es un teorema intuicionista.*

Observe que, si de alguna manera conseguimos implementar un método que determine si una fórmula es o no un teorema intuicionista, entonces tenemos también resuelto el caso para la lógica clásica. Parece ser, de hecho, que la lógica intuicionista y las lógicas intermedias propias son especialmente útiles en el contexto de la programación lógica debido a su capacidad para distinguir entre a y $\neg\neg a$.

2.2.5. Notación y Resultados Básicos

Utilizamos la notación estándar $\vdash_X F$ para denotar que F es una fórmula demostrable (un teorema o una tautología) en la lógica X . Si T es una teoría utilizamos el símbolo $T \vdash_X F$ para denotar $\vdash_X (F_1 \wedge \dots \wedge F_n) \rightarrow F$ para algunas fórmulas F_i contenidas en T . Esta no es la definición tradicional que se puede encontrar en la literatura, pero se puede mostrar que ambas definiciones son equivalentes mediante resultados bien conocidos como son, por ejemplo, el *Teorema de la Deducción*.

Un primer resultado que podemos demostrar, ya que hemos establecido esta notación, es una ligera generalización de la Proposición 2.2. Este resultado nos permite aplicar el mismo resultado de Gödel, aún cuando la prueba intuicionista se valga de un conjunto de premisas.

Proposición 2.3. *Sea T una teoría y F una fórmula. $T \vdash_C F$ si y solo si $T \vdash_I \neg\neg F$.*

Demostración. Por definición $T \vdash_C F$ si existe un conjunto de fórmulas $\{F_1, \dots, F_n\} \subset T$ tales que $\vdash_C (F_1 \wedge \dots \wedge F_n) \rightarrow F$. La Proposición 2.2 muestra que esto es posible si

y solo si $\vdash_I \neg\neg((F_1 \wedge \cdots \wedge F_n) \rightarrow F)$. Pero en intuicionismo se puede demostrar que $\vdash_I \neg\neg(A \rightarrow B) \leftrightarrow (A \rightarrow \neg\neg B)$ así que $\vdash_I (F_1 \wedge \cdots \wedge F_n) \rightarrow \neg\neg F$. Pero, de nuevo, esta es la definición de $T \vdash_I \neg\neg F$. \square

Una teoría T es *consistente*, respecto a la lógica X , si no ocurre que $T \vdash_X \perp$. Es fácil demostrar que, sin embargo, el hecho de que una teoría sea consistente no depende de la lógica particular que utilicemos.

Proposición 2.4. *Sean X y Y son dos lógicas intermedias. Una teoría T es consistente respecto a la lógica X si y solo si es consistente respecto a Y .*

Demostración. Basta demostrar que si $T \vdash_X \perp$ entonces $T \vdash_Y \perp$. Si $T \vdash_X \perp$ entonces, por la Proposición 2.1, también $T \vdash_C \perp$. Usando la Proposición 2.3 tenemos entonces que $T \vdash_I \neg\neg\perp$. Pero es fácil ver que $\vdash_I \neg\neg\perp \rightarrow \perp$ y, por lo tanto, $T \vdash_I \perp$. Finalmente, usando de nuevo la Proposición 2.1, concluimos que $T \vdash_Y \perp$. \square

Gracias a este resultado podemos decir simplemente que una teoría es consistente sin tener que especificar alguna lógica en particular. Observe también que el hecho de que una teoría sea consistente implica que esa teoría tiene al menos un modelo clásico.

Definimos también, si T y U son un par de teorías, el símbolo $T \vdash_X U$ para denotar que $T \vdash_X F$ para toda fórmula $F \in U$. Escribiremos $T \Vdash_X U$ para denotar el hecho de que T es consistente y $T \vdash_X U$. Finalmente decimos que dos teorías T_1 y T_2 son *equivalentes* respecto a la lógica X , y lo denotamos $T_1 \equiv_X T_2$, si es de que $T_1 \vdash_X T_2$ y $T_2 \vdash_X T_1$.

Los siguientes dos resultados, con los que cerramos esta presentación, están relacionados con la noción de teoría completas.

Proposición 2.5. [21] *Si M_1 y M_2 son conjuntos disjuntos de átomos y F una fórmula con lenguaje $\mathcal{L}_F \subseteq M_1 \cup M_2$ entonces $M_1 \cup \neg M_2 \vdash_I F$ ó $M_1 \cup \neg M_2 \vdash_I \neg F$.*

El esquema para demostrar esta proposición está muy bien explicado en [21], aunque no se presenta en el contexto de la lógica intuicionista. La demostración se basa en la idea de observar las premisas como $a \leftrightarrow \top$, cuando $a \in M_1$, y $a \leftrightarrow \perp$, cuando $a \in M_2$. Se pueden reemplazar todas las ocurrencias de átomos en F por los conectivos \top y \perp y reducir después la fórmula, siguiendo las mismas reglas de la evaluación en lógica clásica. En [21] se muestra como, a partir de esta interpretación clásica, es posible construir una prueba formal para F ó $\neg F$ según haya sido el caso.

Proposición 2.6. *Si M_1 y M_2 son conjuntos disjuntos de átomos y F una fórmula con lenguaje $\mathcal{L}_F \subseteq M_1 \cup M_2$ entonces $\neg\neg M_1 \cup \neg M_2 \vdash_I \neg\neg F$ ó $\neg\neg M_1 \cup \neg M_2 \vdash_I \neg F$.*

Demostración. Por la Proposición 2.5 sabemos ya que $M_1 \cup \neg M_2 \vdash_I G$, donde G es alguna de las fórmulas F ó $\neg F$. Entonces, para algún conjunto finito de átomos $\{a_1, \dots, a_n\} \subseteq M_1$, tenemos que $\neg M_2 \vdash_I (a_1 \wedge \cdots \wedge a_n) \rightarrow G$. Luego, como en intuicionismo $\vdash_I A \rightarrow \neg\neg A$, podemos llegar a que $\neg M_2 \vdash_I \neg\neg((a_1 \wedge \cdots \wedge a_n) \rightarrow G)$ y, distribuyendo la doble negación, $\neg M_2 \vdash_I (\neg\neg a_1 \wedge \cdots \wedge \neg\neg a_n) \rightarrow \neg\neg G$. Llegamos finalmente a $\neg\neg M_1 \cup \neg M_2 \vdash_I \neg\neg G$. Equivalentemente, según el caso, $\neg\neg M_1 \cup \neg M_2 \vdash_I \neg\neg F$ ó $\neg\neg M_1 \cup \neg M_2 \vdash_I \neg F$. \square

2.3. Programación Lógica

En esta sección revisaremos algunos de los conceptos y nociones generales de la programación lógica. Presentamos primero diferentes clases de programas lógicos que se encuentran comúnmente en la literatura. Definimos después, para poder dar un significado a los programas lógicos, la noción de *semántica*. Finalmente presentamos la definición oficial de la semántica de answer sets.

2.3.1. Clases de Programas Lógicos

Un *programa lógico* es, en general, una teoría finita. Restringiremos nuestra atención, como hemos mencionado antes, a programas lógicos en el caso proposicional. También es posible tratar programas lógicos con variables mediante un proceso de instanciación como se discutió en la Sección 2.1.3.

Una *clase de programas lógicos* es, simplemente, un conjunto de programas lógicos. Utilizaremos el símbolo Prp para denotar al conjunto de todas las teorías proposicionales. La clase más general de programas que consideraremos en este trabajo es precisamente Prp .

Otra clase importante que consideraremos es, por ejemplo, la clase de los *programas positivos* donde no se permiten ocurrencias del conectivo \perp , a excepción de aquellas que aparezcan en construcciones de la forma $\perp \rightarrow \perp$ que pueden representarse como \top . El caso general de la negación $\neg F$, por lo tanto, no está permitido dentro de los programas positivos.

Los programas lógicos que se encuentran en la literatura se definen usualmente como conjuntos de cláusulas. Una *cláusula* es una fórmula de la forma $H \leftarrow B$ donde las fórmulas H y B se conocen como la *cabeza* (head) y el *cuerpo* (body) de la cláusula respectivamente. Observe que, por tradición en programación lógica, se utiliza el símbolo de implicación en su forma invertida. Así una cláusula $H \leftarrow B$ en un programa lógico puede leerse como: “*H si es de que B*”.

Hay dos casos particulares de cláusulas que son: los *hechos*, de la forma $H \leftarrow \top$, y las *restricciones* o *constraints*, de la forma $\perp \leftarrow B$. Los hechos y las restricciones se pueden abreviar también como H y $\leftarrow B$ respectivamente. Observe que, como los sistemas lógicos que consideramos (las lógicas intermedias) cumplen con la regla *modus ponens*, la abreviación H para la fórmula $H \leftarrow \top$ es consistente con las definiciones que hemos presentado.

Las *cláusulas aumentadas* son cláusulas donde H y B se construyen utilizando los conectivos \wedge , \vee y \neg anidados arbitrariamente. Note que, en particular, las implicaciones anidadas no se permiten dentro de una cláusula aumentada. Resulta que en las lógicas intermedias propias, así como en las semánticas que definiremos más adelante, las fórmulas $\neg A \vee B$ y $A \rightarrow B$ no son equivalentes, es por eso que esta restricción sobre la sintaxis de las cláusulas es importante.

Una *cláusula libre* es la que tiene una disyunción de literales en la cabeza y una conjunción de literales en el cuerpo. Se permite que la cabeza o el cuerpo de una cláusula libre sean vacíos para denotar una restricción o un hecho respectivamente.

Una *cláusula general* es una cláusula libre que no permite negación en la cabeza. Todas las literales que ocurren en la cabeza deben de ser átomos positivos. Finalmente una *cláusula disyuntiva* es una cláusula general cuya cabeza no es vacía, es decir, no es una restricción.

Ejemplo 2.6. A continuación presentamos algunos ejemplos de cláusulas.

$a \vee b \vee c \leftarrow p \wedge q \wedge \neg r.$	disyuntiva
$\leftarrow p \wedge \neg q.$	general (restricción)
$a \vee \neg b \leftarrow p \wedge \neg q.$	libre
$a \vee \neg a.$	libre (hecho)
$\neg(a \wedge \neg b) \leftarrow p \vee (\neg q \wedge r)$	aumentada
$a \leftarrow (p \rightarrow q)$	fórmula proposicional

Definimos un *programa aumentado* como una teoría que contiene únicamente cláusulas aumentadas, y la clase **Aug** como la clase que contiene a todos los programas aumentados. Similarmente se definen los *programas libres*, *generales* y *disyuntivos*; así como sus respectivas clases: **Free**, **Gen** y **Dis**. Observe que, en particular, estas clases recién definidas satisfacen la relación: $\text{Dis} \subset \text{Gen} \subset \text{Free} \subset \text{Aug} \subset \text{Prp}$.

Otra clase importante de programas, que se utiliza como una clase intermedia para definir la semántica de answer sets, es la clase de programas básicos. Un *programa básico* es un programa aumentado donde no la cabeza y el cuerpo de cada cláusula no contienen fórmulas negadas $\neg F$. Observe que los programas básicos no son siempre programas positivos. El conectivo \perp sí puede aparecer, en la cabeza o en el cuerpo de las cláusulas, siempre que no ocurra como consecuente de una implicación $F \rightarrow \perp$ produciendo una negación. Un programa básico puede, incluso, contener restricciones. La clase de programas básicos la denotamos como **Bas**.

2.3.2. Semánticas para Programas Lógicos

Una *interpretación* para un programa lógico P es un conjunto de átomos $M \subseteq \mathcal{L}_P$.³ Dada una clase de programas C , una *semántica* Sem es una función que asigna a cada programa lógico $P \in C$ un conjunto de interpretaciones para P . Un ejemplo sencillo de una semántica es la semántica de los modelos clásicos de un programa proposicional.

Definición 2.1. Para cada programa $P \in \text{Prp}$ se define la *semántica de modelos (clásicos)* de P como el conjunto

$$M(P) = \{M \subseteq \mathcal{L}_P \mid M \models P\} .$$

En algunas ocasiones es importante distinguir, dentro de los modelos de una semántica, aquellos que tienen la propiedad de ser mínimos. Así se puede definir, para cada semántica, una cierta semántica mínima.

³No confundir con las interpretaciones en G_i de la Sección 2.2.2. Las interpretaciones que aquí definimos no tienen, aún, ningún significado semántico.

Definición 2.2. Si Sem es una semántica definida para una clase de programas C entonces, para cada $P \in C$ se define la semántica mínima:

$$\text{Sem}^m(P) = \{M \in \text{Sem}(P) \mid (\nexists N \in \text{Sem}(P))(N \subset M)\} .$$

La semántica de modelos mínimos, M^m , es una semántica bien conocida y la cual ha probado ser útil en muchas aplicaciones. Observe que un programa puede tener uno, ninguno o varios modelo mínimos.

Ejemplo 2.7. Por ejemplo $M^m(\{b, a \leftarrow b\}) = \{\{a, b\}\}$ mientras que $M^m(\{a \wedge \neg a\}) = \emptyset$ y $M^m(\{a \vee b\}) = \{\{a\}, \{b\}\}$. En el último caso observe que $\{a, b\}$ también es un modelo del programa $\{a \vee b\}$, pero no es mínimo. Además $M^m(\{a \vee \neg a\}) = \{\emptyset\}$ es distinto al caso de un programa sin modelos mínimos. El programa $\{a \vee \neg a\}$ tiene exactamente un modelo mínimo y corresponde al conjunto vacío.

2.3.3. La Semántica de Answer Sets

Presentamos ahora la definición formal de la semántica de answer sets para programas aumentados. Esta definición es una adaptación de la que se presenta en [18], la única diferencia esencial es que aquí no consideramos la negación explícita o negación clásica (que hemos denotado con \sim). Veremos, al final de esta sección, como se puede recuperar este segundo tipo de negación dentro del esquema que proponemos.

Observe que, sin embargo, los autores de [18] utilizan el símbolo *not* en lugar de nuestro símbolo \neg . La notación *not* es común en el contexto de la programación lógica pero, en muchos de los nuevos trabajos que utilizan un enfoque basado en lógica matemática para estudiar a las semánticas y programas lógicos, se prefiere mantener el símbolo \neg pues corresponde de manera natural con la negación utilizada en la lógica proposicional.

Otra de las diferencias con respecto a [18] es que ellos consideran también un constructor del tipo *if-then-else*, pero éste es simplemente una abreviación de una fórmula proposicional. La definición se da en dos partes, primero para programas básicos y luego se extiende para programas aumentados arbitrarios.

Definición 2.3. [18] Para los programas básicos $P \in \text{Bas}$ se define la *semántica de answer sets* como $\text{AS}(P) = M^m(P)$.

Definición 2.4. [18] El *reducto* de una fórmula, cláusula o programa aumentado, relativo a un conjunto de átomos M , se define recursivamente como sigue:

- Si F es un átomo, \perp o \top entonces $F^M = F$.
- $(F \wedge G)^M = F^M \wedge G^M$.
- $(F \vee G)^M = F^M \vee G^M$.
- $(\neg F)^M = \perp$ si $X \models F$ y $(\neg F)^M = \top$ de otro modo.

$$\blacksquare (H \leftarrow B)^M = H^M \leftarrow B^M.$$

Finalmente $P^M = \{(H \leftarrow B)^M \mid (H \leftarrow B) \in P\}$.

Observe que P^M , para cualquier programa aumentado P , es siempre un programa básico. El reducto se encarga, precisamente, de eliminar todas las negaciones ($\neg F$) contenidas en P utilizando la información contenida en M .

Definición 2.5 (Answer Sets). [18] Para todo programa aumentado P definimos la *semántica de answer sets* como

$$\text{AS}(P) = \{M \subseteq \mathcal{L}_P \mid M \in \text{AS}(P^M)\}$$

A continuación presentamos en un ejemplo concreto como se calculan los answer sets de un programa lógico particular.

Ejemplo 2.8. Sea P el programa aumentado:

$$\begin{aligned} a &\leftarrow \neg\neg a. \\ \neg b &\leftarrow c \vee b. \end{aligned}$$

Verifiquemos primero si el conjunto $M = \{a\}$ es un answer set de P . Calculando el reducto para la primera cláusula tenemos

$$(a \leftarrow \neg\neg a)^M = a \leftarrow (\neg\neg a)^M = a \leftarrow \top.$$

El último paso de la transformación se obtiene del hecho que $M \models \neg\neg a$. Repitiendo esta transformación, ahora sobre la segunda cláusula, obtenemos que P^M consta de:

$$\begin{aligned} a &\leftarrow \top. \\ \top &\leftarrow c \vee b. \end{aligned}$$

En este punto es fácil observar que $M \models P^M$. Para probar que M es un answer set basta mostrar que también es un modelo mínimo de P^M . Esto se obtiene inmediatamente del hecho de que el único conjunto N , subconjunto propio de M , es el conjunto vacío \emptyset y, en particular, $\emptyset \not\models a \leftarrow \top$. De aquí concluimos que $M = \{a\}$ es un answer set de P^M y, por lo tanto, también M es answer set de P .

Observe que, por otra parte, si consideramos ahora el conjunto $M = \emptyset$, entonces se produce un reducto distinto P^M :

$$\begin{aligned} a &\leftarrow \perp. \\ \top &\leftarrow c \vee b. \end{aligned}$$

También en este caso $M \models P^M$, y como M no tiene subconjuntos propios, concluimos también que \emptyset es un answer set de P . Se puede mostrar, analizando el resto de los casos, que P tiene exactamente dos answer sets: $\text{AS}(P) = \{\emptyset, \{a\}\}$.

Una forma de recuperar la negación explícita dentro de la semántica de answer sets es agregar al lenguaje \mathcal{L} , nuevos átomos de la forma $\sim a$. Si usamos $\mathcal{L}^\sim = \mathcal{L} \cup \{\sim a \mid a \in \mathcal{L}\}$ para denotar a este conjunto extendido de átomos, un conjunto $M \subseteq \mathcal{L}^\sim$ es un answer set de un programa aumentado P si es un answer set de P , según la Definición 2.5, y M no contiene al conjunto $\{a, \sim a\}$ para ningún $a \in \mathcal{L}$.

Se puede pensar en $\sim a$ como un átomo con un nombre asignado de manera conveniente para que una herramienta hecha para calcular answer sets elimine automáticamente los modelos donde ambos átomos a y $\sim a$ aparecen simultáneamente.

Capítulo 3

Caracterización de Answer Sets

El objetivo principal de este capítulo es obtener una caracterización de la definición de answer sets, presentada en la Definición 2.5, en términos de lógicas intermedias. Esta caracterización nos permite reducir el problema de determinar si un conjunto de átomos es answer set de un programa lógico a determinar si una fórmula, construible en tiempo polinomial a partir del programa, es o no demostrable en alguna lógica intermedia propia.

Este resultado no sólo provee una nueva forma de enfrentar el cálculo de answer sets para un programa lógico, además ofrece una visión alternativa para estudiar y comprender mejor a esta semántica. Podemos también, gracias a esta caracterización, proponer una generalización de la noción de answer sets para programas proposicionales e, incluso, examinar otras posibilidades que pueden surgir al considerar otros tipos y variedades de lógicas.

David Pearce inició esta línea de investigación proponiendo, para el caso de programas disyuntivos, una caracterización de answer sets en términos de la lógica intuicionista. Demostró en [33] que una fórmula es consecuencia lógica de un programa disyuntivo en la semántica de answer sets si y solo si pertenece a cada extensión del programa, obtenida agregando literales negativas, que sea consistente e intuicionísticamente completa.

Cuando hay un programa P claro por contexto y si M es conjunto de átomos, usamos el símbolo \widetilde{M} para denotar al complemento $\mathcal{L}_P \setminus M$. Así podemos presentar de manera formal, y utilizando esta notación, el resultado obtenido por Pearce.

Teorema 3.1. [33] Sea $P \in \text{Dis}$ y $M \subseteq \mathcal{L}_P$. $P \in \text{AS}(P)$ si y solo si $P \cup \neg\widetilde{M} \Vdash_{\text{I}} M$.

Este mismo resultado funciona para la clase de programas generales (que permiten el uso de restricciones). Sin embargo deja de ser válido si permitimos programas libres con negación en la cabeza o alguna otra clase de programas más general.

Ejemplo 3.1. Consideremos el programa $P = \{a \vee \neg a\}$, según la Definición 2.5 este programa tiene dos answer sets: \emptyset y $\{a\}$. Si consideramos entonces $M = \emptyset$ la condición de Pearce usando lógica intuicionista se reduce a verificar la consistencia de la teoría $\{a \vee \neg a, \neg a\}$ la cual, en efecto, ocurre. Al considerar, sin embargo, $M = \{a\}$ la condición de Pearce requiere que $a \vee \neg a \vdash_{\text{I}} a$. ¡Pero esto no es posible ni siquiera en lógica clásica! Por lo tanto observamos

que la condición propuesta por Pearce no rescata correctamente todos los answer sets en el caso de los programas libres.

Posteriormente, en [17] y a partir de resultados presentados en [32], se evitó este problema reformulando la caracterización de answer sets en términos de la “lógica de equilibrio”. Esta lógica corresponde a ciertos modelos preferidos de la lógica HT.

Nosotros proponemos una solución alternativa, siguiendo el enfoque de [33], que consiste en permitir no sólo átomos negados para completar los programas lógicos, sino permitir también átomos doblemente negados. Esta propuesta tiene sentido ya que en todas las lógicas intermedias propias, en particular en la lógica intuicionista, las fórmulas a y $\neg\neg a$ no son equivalentes. En el caso del ejemplo anterior la condición intuicionista se convertiría en $a \vee \neg a, \neg\neg a \vdash_I a$, lo cual sí es posible y nos permite recuperar el answer set que había sido perdido. El resultado formal se presenta más adelante en el Teorema 3.2.

Como consecuencia de los resultados presentados en este capítulo podremos obtener importantes conclusiones. Podremos demostrar primero que nuestro enfoque y el propuesto en [17] son, en efecto, equivalentes. Esto nos permitirá obtener una buena cantidad de retroalimentación entre ambas construcciones teóricas.

A partir de estos resultados podremos también dar una interpretación natural de la semántica de answer sets, en el contexto de agentes lógicos, en términos de los conceptos de conocimiento y creencia.

Encontraremos también que la condición propuesta por Pearce en el Teorema 3.1 está caracterizando, de hecho, modelos que además de ser answer sets son modelos mínimos. A partir de estos resultados redescubrimos un resultado bien conocido y es que, para la clase de programas disyuntivos, los answer sets siempre son modelos mínimos.

3.1. Resultados Preliminares

Revisaremos en esta sección algunos resultados básicos que serán utilizados en la demostración del Teorema 3.2. Presentamos primero algunas propiedades de las lógicas intermedias y, después, una serie de reducciones que podemos aplicar a los programas lógicos para simplificar su estructura.

3.1.1. Lógicas Intermedias

Algunos de los siguientes resultados tienen relevancia por sí mismos dentro del área de las lógicas intermedias. Nuestro interés principal se centrará, sin embargo, en sus posibles aplicaciones más adelante dentro del contexto de la programación lógica y la semántica de answer sets.

El primero de ellos muestra como, si una teoría positiva ha sido construida sobre un lenguaje con un sólo átomo, entonces la demostrabilidad en lógica clásica y el resto de las lógicas intermedias coincide.

Lema 3.1. *Sea X una lógica intermedia y $T \in \text{Pos}$. Si el lenguaje $\mathcal{L}_T = \{a\}$ y $T \vdash_C a$ entonces $T \vdash_X a$.*

Demostración. Construimos primero una nueva teoría T' aplicando los siguientes reemplazos sobre la teoría T hasta que ya no pueda aplicarse ninguno de ellos:

- $\top \wedge \top$ por \top .
- $a \wedge a$, $a \wedge \top$ y $\top \wedge a$ por a .
- $a \vee a$ por a .
- $a \vee \top$, $\top \vee a$ y $\top \vee \top$ por \top .
- $\top \rightarrow a$ por a .
- $a \rightarrow a$, $a \rightarrow \top$ y $\top \rightarrow \top$ por \top .

Es claro que $T \equiv_X T'$, para cualquier lógica intermedia X . Observe que cada fórmula en T debe reducirse ya sea a \top o bien al átomo a , es decir $T' \subseteq \{\top, a\}$. Como debemos tener que $T' \vdash_C a$ sabemos entonces que $a \in T'$. Ahora, como $a \in T'$, es claro que también $T' \vdash_X a$ y, del hecho de que $T \equiv_X T'$, podemos concluir que $T \vdash_X a$. \square

El siguiente lema se basa en el anterior para mostrar que si una teoría positiva permite demostrar, usando lógica clásica, todos y cada uno de los átomos en su lenguaje; entonces también podrá demostrarlos utilizando cualquier otra lógica intermedia.

Lema 3.2. *Sea X una lógica intermedia y $T \in \text{Pos}$. $T \vdash_C \mathcal{L}_T$ si y solo si $T \vdash_X \mathcal{L}_T$.*

Demostración. Si suponemos que $T \vdash_C \mathcal{L}_T$, sea a un átomo en \mathcal{L}_T . Observe que podemos reemplazar todos los átomos que ocurren en la prueba de $T \vdash_C a$ para obtener una prueba válida de $T_a \vdash_C a$, donde el programa T_a se obtiene al reemplazar todos los átomos que ocurren en la teoría T por el átomo a . Usando el Lema 3.1 tenemos que $T_a \vdash_X a$. Si hacemos $E = \{x \leftrightarrow y \mid x, y \in \mathcal{L}_T\}$ entonces también $T_a \cup E \vdash_X a$. Ahora sólo observe que $T_a \cup E \equiv_X T$ y, por lo tanto, podemos concluir que $T \vdash_X a$.

El recíproco es consecuencia directa de la Proposición 2.1. \square

Los siguientes resultados son útiles para, en algunos casos particulares, eliminar premisas que no son necesarias en las pruebas de ciertos teoremas en lógicas intermedias. El lema siguiente muestra que se pueden eliminar las premisas que tengan un lenguaje ajeno al de la fórmula que se quiere demostrar.

Lema 3.3. *Sea X una lógica intermedia, sean T, U dos teorías y F una fórmula tales que sus lenguajes cumplan $\mathcal{L}_{T \cup \{F\}} \cap \mathcal{L}_U = \emptyset$. Si la teoría U es consistente y $T \cup U \vdash_X F$ entonces $T \vdash_X F$.*

Demostración. Como U es consistente, entonces U tiene al menos un modelo clásico M . En la secuencia de fórmulas que conforman la prueba de $T \cup U \vdash_X F$ se puede reemplazar cada átomo $a \in \mathcal{L}_U$ con \top cuando $a \in M$ y con \perp cuando $a \notin M$. Como T y F no contienen ninguno de estos átomos $a \in \mathcal{L}_U$, estas premisas y el resultado de la prueba no son alterados. Mientras que las premisas de U se mapean a tautologías o teoremas de la lógica utilizada dejando una prueba para $T \vdash_X F$. \square

El último resultado, que presentamos a continuación, muestra una propiedad característica de la lógica intuicionista. Nos indica que en una demostración intuicionista de una fórmula positiva a partir de una teoría positiva, el agregar premisas adicionales de la forma $\neg a$, para algún átomo a , es totalmente irrelevante. Observe que, de hecho, este resultado no se cumple para la lógica clásica.

Lema 3.4. [29] Sean $T \in \text{Pos}$, $M \subseteq \mathcal{L}_T$ y F una fórmula positiva. Si $T \cup \neg\neg M \vdash_{\text{I}} F$ entonces también $T \vdash_{\text{I}} F$.

La demostración de este resultado se encuentra en [29] y está basada en una representación de la lógica intuicionista en términos de sistemas de deducción natural.

3.1.2. Reducciones

En esta sección presentamos algunas reducciones de teorías y varias de sus propiedades en términos de lógicas intermedias. Estas reducciones serán de utilidad para reducir la complejidad de programas lógicos y poder aplicar algunos de los resultados que presentamos en la sección anterior.

La primera de estas reducciones tiene como objetivo evaluar el efecto de todos los conectivos \perp y \top dentro de un programa lógico. Después de aplicar la transformación se habrán eliminado todos los \perp que aparecen en el programa, excepto aquellos que sirvan para formar negaciones $F \rightarrow \perp$ o si la teoría completa fue reducida al conectivo \perp . Además, no aparecerá ninguna ocurrencia de \top .

Definición 3.1. Para cada fórmula F definimos la fórmula $\text{Redu}^\perp(F)$ realizando los siguientes reemplazos en F hasta que ya no se pueda realizar ninguno de ellos, donde A es una fórmula cualquiera:

- $A \wedge \top$ y $\top \wedge A$ por A .
- $A \wedge \perp$ y $\perp \wedge A$ por \perp .
- $A \vee \top$ y $\top \vee A$ por \top .
- $A \vee \perp$ y $\perp \vee A$ por A .
- $A \rightarrow \top$ y $\perp \rightarrow A$ por \top .
- $\top \rightarrow A$ por A .

Ahora, para una teoría T , definimos $\text{Redu}^\perp(T) = \{\perp\}$ si existe alguna fórmula $F \in T$ con $\text{Redu}^\perp(F) = \perp$. Si esto no ocurre entonces definimos:

$$\text{Redu}^\perp(T) = \left\{ \text{Redu}^\perp(F) \mid F \in T \text{ y } \text{Redu}^\perp(F) \neq \perp \right\}.$$

Se sigue inmediatamente de la definición que esta reducción construye siempre una nueva teoría que es equivalente, bajo cualquier lógica intermedia, a la original.

Proposición 3.1. Si T es una teoría y X una lógica intermedia, entonces $T \equiv_X \text{Redu}^\perp(T)$.

Demostración. Basta observar que todos los reemplazos realizados por la reducción son válidos en intuicionismo, en particular también en todas las lógicas intermedias. \square

La siguiente reducción permite, dado un conjunto de átomos M , asumir que los átomos contenidos en M son falsos ($\neg M$) y, usando esta información, reducir los programas lógicos. Una de las propiedades importantes de esta reducción es que, después de aplicarla, el programa reducido no contiene ya a ninguno de los átomos en M . Se prueba también que, agregando $\neg M$ como premisas, se preserva la equivalencia en lógicas intermedias.

Definición 3.2. Sea T una teoría y M un conjunto de átomos. Sea también T' la teoría obtenida reemplazando todas las ocurrencias de los átomos $a \in M$ dentro de T con el conectivo \perp . Definimos entonces la reducción $\text{Reduce1}(T, \neg M) = \text{Redu}^\perp(T')$.

Proposición 3.2. Si T es una teoría, M un conjunto de átomos y X una lógica intermedia, entonces $T \cup \neg M \equiv_X \text{Reduce1}(T, \neg M) \cup \neg M$.

Demostración. Se sigue inmediatamente de la Proposición 3.1 y del hecho de que, para cualquier átomo a , se tiene $\neg a \vdash_X a \leftrightarrow \perp$. \square

La última de las reducciones es similar a la anterior, dado un conjunto de átomos M , evalúa el efecto que tiene sobre el programa el agregar las premisas $\neg\neg M$. Ya que estamos interesados en preservar equivalencia dentro de las lógicas intermedias, esta reducción sólo puede tener efecto dentro de subfórmulas negadas. Es decir, sólo los átomos que ocurran dentro de alguna fórmula $F \rightarrow \perp$ serán reemplazados.

Definición 3.3. Sea T una teoría y M un conjunto de átomos. Sea T' la teoría obtenida reemplazando todas las ocurrencias de átomos $a \in M$ dentro de T con el conectivo \top si es que la ocurrencia de dicho átomo aparece dentro de alguna fórmula de la forma $F \rightarrow \perp$. Definimos entonces la reducción $\text{Reduce2}(T, \neg\neg M) = \text{Redu}^\perp(T')$.

Proposición 3.3. Si T es una teoría, M un conjunto de átomos y X una lógica intermedia, entonces $T \cup \neg\neg M \equiv_X \text{Reduce2}(T, \neg\neg M) \cup \neg\neg M$.

Demostración. Se sigue también de la Proposición 3.1 y del hecho de que para cualquier fórmula F , si reemplazamos las ocurrencias del átomo a por \top para obtener F' , entonces se tiene $\neg\neg a \vdash_X \neg F \leftrightarrow \neg F'$. \square

Ejemplo 3.2. Consideremos la teoría T :

$$\begin{aligned} &(a \wedge b) \vee \neg(c \wedge d). \\ &b \wedge d \rightarrow b. \\ &(c \rightarrow (a \rightarrow b)) \wedge e. \end{aligned}$$

Para calcular $\text{Reduce1}(T, \{-b\})$ reemplazamos todas las ocurrencias del átomo b por el conectivo \perp y aplicamos sucesivamente las reducciones de Redu^\perp :

$$\begin{aligned} &(a \wedge \perp) \vee \neg(c \wedge d). & \perp \vee \neg(c \wedge d). & \neg(c \wedge d). \\ &\perp \wedge d \rightarrow \perp. & \implies \perp \rightarrow \perp. & \implies \top. \\ &(c \rightarrow (a \rightarrow \perp)) \wedge e. & (c \rightarrow (a \rightarrow \perp)) \wedge e. & (c \rightarrow (a \rightarrow \perp)) \wedge e. \end{aligned}$$

De modo que la teoría $T' = \text{Reduce1}(T, \{-b\})$ que se obtiene al final es:

$$\begin{aligned} &\neg(c \wedge d). \\ &(c \rightarrow (a \rightarrow \perp)) \wedge e. \end{aligned}$$

Si calculamos ahora $T'' = \text{Reduce2}(T', \{-a, \neg c\})$, empezamos por reemplazar las ocurrencias de a y c , que aparecen bajo el alcance de una negación, por el conectivo \top para después utilizar de nuevo Redu^\perp :

$$\begin{array}{l} \neg(\top \wedge d). \\ (c \rightarrow (\top \rightarrow \perp)) \wedge e. \end{array} \implies \begin{array}{l} \neg d. \\ (c \rightarrow \perp) \wedge e. \end{array}$$

Observe que, como resultado de las simplificaciones de Reduce2, se produjo una nueva negación en la última de las fórmulas y, por lo tanto, podemos aplicar de nuevo la reducción $T''' = \text{Reduce2}(T'', \{\neg a, \neg c\})$ para obtener:

$$\begin{array}{l} \neg d. \\ e. \end{array}$$

Esta es la teoría final simplificada que se puede obtener con las reducciones presentadas en esta sección.

Como se observó en el ejemplo anterior es posible que se necesite aplicar varias veces la reducción Reduce2 para simplificar por completo una teoría. Utilizaremos el símbolo $\text{Reduce2}^*(P, \neg\neg M)$ para denotar la teoría obtenida después de aplicar sucesivamente la reducción Reduce2 hasta que se alcance un punto fijo. Así podemos presentar una reducción que integre todos estos resultados y será de utilidad más adelante para proponer un método que se calcule los answer sets de un programa lógico.

Definición 3.4. Si T es una teoría, M_1 y M_2 conjuntos de átomos disjuntos definimos entonces $\text{Reduce}(T, \neg M_1, \neg\neg M_2) = \text{Reduce2}^*(\text{Reduce1}(T, \neg M_1), \neg\neg M_2)$

Proposición 3.4. Si T es una teoría, $M_1, M_2 \subseteq \mathcal{L}_T$ un par de conjuntos disjuntos y X una lógica intermedia, entonces $T \cup \neg M_1 \cup \neg\neg M_2 \equiv_X \text{Reduce}(T, \neg M_1, \neg\neg M_2) \cup \neg M_1 \cup \neg\neg M_2$.

Demostración. Se sigue inmediatamente de las Proposiciones 3.2 y 3.3. \square

Observe también que si, según la notación de la proposición anterior, $M_1 \cup M_2 = \mathcal{L}_T$ entonces la teoría $\text{Reduce}(T, \neg M_1, \neg\neg M_2)$ debe ser una teoría positiva. Esto es porque la reducción Reduce2 involucrada podrá seguir aplicándose de manera iterativa mientras siga existiendo negación en la teoría reducida.

3.2. Caracterizaciones

En esta sección podemos presentar, finalmente, caracterizaciones de diversas semánticas en términos de demostrabilidad en lógicas intermedias. Recuerde que, en estos enunciados y cuando hay un programa lógico P claro por contexto, utilizamos el símbolo \widetilde{M} para denotar al complemento $\mathcal{L}_P \setminus M$.

3.2.1. Modelos Mínimos

Iniciamos presentando una caracterización de la semántica de modelos mínimos, M^m , en términos de la lógica clásica.

Proposición 3.5. Sea $P \in \text{Prp}$ y $M \subseteq \mathcal{L}_P$ un conjunto de átomos. $M \in M^m(P)$ si y solo si $P \cup \widetilde{M} \Vdash_C M$.

Demostración. Supongamos primero que M es un modelo mínimo de P . Como M es modelo de P se sigue de inmediato que $P \cup \neg\widetilde{M}$ es consistente. Observe además, como M es un modelo mínimo de P , el único modelo de la teoría $P \cup \neg\widetilde{M}$ es precisamente M . De aquí se sigue que $P \cup \neg\widetilde{M} \vdash_C M$.

Supongamos ahora que $P \cup \neg\widetilde{M} \Vdash_C M$. Esto implica, en particular, que el programa $P \cup \neg\widetilde{M} \cup M$ es consistente. Como M es el único modelo para $\neg\widetilde{M} \cup M$ se obtiene que también M debe ser un modelo para P . Supongamos que existe otro conjunto $N \subset M$ que sea modelo de P . Sea a un átomo en $M \setminus N$, como entonces $a \in \widetilde{N}$, tenemos de inmediato que $P \cup \neg\widetilde{N} \vdash_C \neg a$. Pero, como $\widetilde{M} \subset \widetilde{N}$, ya sabíamos que $P \cup \neg\widetilde{N} \vdash_C a$. De aquí se sigue que la teoría $P \cup \neg\widetilde{N}$ es inconsistente y, por lo tanto N no puede ser un modelo de P . \square

3.2.2. Answer Sets

El siguiente par de lemas nos acercan a uno de los resultados principales que es obtener una caracterización de la semántica de answer sets en términos de lógica matemática. El primero de los lemas, que se presenta a continuación, muestra la utilidad de las lógicas intermedias para nuestro propósito. Este resultado no es cierto, de hecho, si utilizáramos lógica clásica en lugar de una lógica intermedia propia.

Lema 3.5. *Sea X una lógica intermedia propia y $P \in \text{Pos}$. Se tiene entonces que $P \Vdash_C \mathcal{L}_P$ si y solo si $P \cup \neg\neg\mathcal{L}_P \vdash_X \mathcal{L}_P$.*

Demostración. Es fácil verificar la consistencia, si P es consistente y $P \vdash_C \mathcal{L}_P$ entonces también $P \cup \mathcal{L}_P$ es consistente. Además, si $P \cup \neg\neg\mathcal{L}_P$ es consistente es inmediato que P es consistente. Ahora, si $P \vdash_C \mathcal{L}_P$ ya sabemos, por el Lema 3.2, que $P \vdash_X \mathcal{L}_P$, basta agregar las premisas $\neg\neg\mathcal{L}_P$ para obtener $P \cup \neg\neg\mathcal{L}_P \vdash_X \mathcal{L}_P$.

Para el recíproco supongamos que $P \cup \neg\neg\mathcal{L}_P \vdash_X \mathcal{L}_P$, como G_3 es la lógica intermedia propia más fuerte, también $P \cup \neg\neg\mathcal{L}_P \vdash_{G_3} \mathcal{L}_P$. Utilizando la representación axiomática de la lógica G_3 , presentada en la Sección 2.2.3, encontramos que $P \cup \neg\neg\mathcal{L}_P \cup AX \vdash_I \mathcal{L}_P$, donde AX es un conjunto de instancias del axioma A11'. También, sin pérdida de generalidad, podemos suponer que AX sólo contiene átomos del lenguaje \mathcal{L}_P . Construimos entonces la teoría $AX' = \text{Reduce2}^*(AX, \neg\neg\mathcal{L}_P)$ de modo que, por el Lema 3.3, $P \cup \neg\neg\mathcal{L}_P \cup AX' \vdash_I \mathcal{L}_P$. Observe que AX' debe ser un programa positivo pues, al aplicar la reducción usando el lenguaje completo \mathcal{L}_P , todas las fórmulas $F \rightarrow \perp$ deberán evaluarse y simplificarse totalmente.

Utilizando ahora el Lema 3.4 podemos eliminar las premisas $\neg\neg M$ para obtener una prueba de $P \cup AX' \vdash_I \mathcal{L}_P$. También es claro que, usando lógica clásica y por la Proposición 2.3, tenemos $P \cup AX' \vdash_C \mathcal{L}_P$. Basta observar que el conjunto AX' sólo contiene ahora tautologías clásicas y, por lo tanto, no son necesarias para la prueba. Así podemos concluir finalmente que $P \vdash_C \mathcal{L}_P$. \square

El siguiente resultado establece el vínculo entre la caracterización de answer sets que proponemos y la caracterización de modelos mínimos para el caso de programas básicos.

Lema 3.6. *Sea $P \in \text{Bas}$, M un conjunto de átomos y X una lógica intermedia propia. $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$ si y solo si $P \cup \neg\widetilde{M} \Vdash_C M$*

Demostración. Sea $P' = \text{Reduce1}(P, \neg\widetilde{M})$. Observe que, aplicando la Proposición 3.2, podemos intercambiar $P \cup \neg\widetilde{M}$ con $P' \cup \neg\widetilde{M}$ dentro de los enunciados de las pruebas lógicas. Además, como ahora el programa P' no contiene ya ninguno de los átomos en \widetilde{M} , podemos usar el Lema 3.3 para eliminar al conjunto $\neg\widetilde{M}$ de las premisas en dichos enunciados. Bastaría probar entonces que $P' \cup \neg\neg M \Vdash_X M$ si y solo si $P' \Vdash_C M$.

Para poder aplicar el Lema 3.5 sólo tenemos que verificar que se cumplen las hipótesis necesarias. Observe que P' debe de ser un programa positivo pues cada cláusula $A \leftarrow B$ en P , se debe reducir a una fórmula positiva ya sea de la forma $A' \leftarrow B'$, A' o incluso \top . Los dos casos restantes $\perp \leftarrow B'$ y \perp no podrían ocurrir por las condiciones de consistencia. En el caso $\perp \leftarrow B'$ la fórmula B' solo puede contener conjunciones y disyunciones de los átomos en M , pero como $M \vdash_X \neg\neg B'$ se provocaría inconsistencia. Es fácil ver también que, después de la aplicación de la reducción Reduce, tenemos $M = \mathcal{L}_{P'}$. \square

Después de haber presentado los resultados anteriores tenemos las herramientas necesarias para caracterizar la noción de answer sets en términos de lógicas intermedias. El siguiente teorema establece de manera formal este resultado.

Teorema 3.2. *Si $P \in \text{Aug}$, M es un conjunto de átomos y X es una lógica intermedia propia, entonces $M \in \text{AS}(P)$ si y solo si $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$.*

Demostración. Ahora $M \in \text{AS}(P)$ si y solo si, por la Definición 2.5 de answer sets, el conjunto $M \in \text{AS}(P^M) = \text{M}^m(P^M)$ si y solo si, por la Proposición 3.5, $P^M \cup \neg\widetilde{M} \Vdash_C M$ si y solo si, como P^M es un programa básico y por el Lema 3.6, $P^M \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$. Observe sin embargo que, como se tienen los dos conjuntos de premisas $\neg\widetilde{M}$ y $\neg\neg M$, también se tiene la equivalencia $P^M \cup \neg\widetilde{M} \cup \neg\neg M \equiv_X P \cup \neg\widetilde{M} \cup \neg\neg M$ y, por lo tanto, podemos concluir que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$. \square

Este resultado nos permite, además de ofrecer una nueva caracterización de la noción de answer sets, extender la aplicación de esta semántica para tratar con teorías proposicionales sin ninguna restricción. La definición original de answer sets, presentada en la Definición 2.5, está restringida a la clase de programas aumentados pues, en particular, no hay una manera clara o evidente de extender la operación del reducto para tratar con implicaciones anidadas.

La fórmula intuicionista del Teorema 3.2, sin embargo, no parece imponer ninguna restricción particular sobre la sintaxis o la forma que deben tener las cláusulas de los programas lógicos. Ésto nos permite proponer la siguiente generalización de la noción de answer sets para teorías proposicionales arbitrarias.

Definición 3.5. Si $P \in \text{Prp}$ definimos la *semántica de answer sets* como

$$\text{AS}(P) = \left\{ M \subseteq \mathcal{L}_P \mid P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_I M \right\}.$$

Observe que, también como consecuencia del Teorema 3.2, sabemos que, en la clase de programas aumentados, la lógica intuicionista I utilizada en esta definición puede ser reemplazada por cualquier otra lógica intermedia propia sin alterar los modelos que captura la semántica. El siguiente teorema muestra que esta misma afirmación sigue siendo válida aún en la clase de programas lógicos proposicionales.

Teorema 3.3. *Sean X y Y dos lógicas intermedias propias, sea $P \in \text{Prp}$ y $M \subseteq \mathcal{L}_P$. Se tiene entonces que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$ si y solo si $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_Y M$.*

Demostración. Si suponemos que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$ entonces, por la Proposición 2.1, se sigue que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{G_3} M$. Si hacemos $P' = \text{Reduce}(P, \neg\widetilde{M}, \neg\neg M)$ entonces, utilizando la Proposición 3.4, obtenemos que $P' \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{G_3} M$. Usando el Lema 3.3 podemos eliminar el conjunto de premisas $\neg\widetilde{M}$ y, como P' es un programa positivo, es posible aplicar el Lema 3.5 y llegar a que $P' \cup \neg\neg M \Vdash_I M$. Si reintegramos entonces el conjunto de premisas $\neg\widetilde{M}$ y cambiamos al programa P' por el original P de nuevo, por la Proposición 2.1, llegamos a que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_Y M$. \square

Observe por otra parte que, si utilizáramos la lógica clásica como base para la Definición 3.5, entonces lo que la semántica rescata es, precisamente, los modelos clásicos de la teoría P . En capítulos posteriores analizaremos otras opciones y posibilidades si consideramos en esta misma definición algunos otros tipos de lógicas.

Recordemos también que, a partir de resultados en [17, 32], se había podido establecer una caracterización de answer sets, para teorías proposicionales, en términos de la “lógica de equilibrio” basada en HT. Destacan los dos siguientes resultados.

Lema 3.7. [32] *Dada una teoría $T \in \text{Prp}$ y $M \subseteq \mathcal{L}_T$. $T \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_{\text{HT}} M$ si y solo si M es un modelo de equilibrio de T .*

Lema 3.8. [17] *Dada una teoría $T \in \text{Prp}$ y $M \subseteq \mathcal{L}_T$. M es un modelo de equilibrio de T si y solo si M es un answer set de T .*

A partir de nuestros resultados, en particular del Teorema 3.3, obtenemos también que la lógica de equilibrio se puede caracterizar por cualquier lógica intermedia.

Corolario 3.1. *Dada una teoría $T \in \text{Prp}$, $M \subseteq \mathcal{L}_T$ y X una lógica intermedia propia. $T \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$ si y solo si M es un modelo de equilibrio de T .*

Otra consecuencia importante, que se desprende de esta caracterización, es que justifica la siguiente interpretación de answer sets como modelos de creencias en sistemas de agentes lógicos. Supongamos que tenemos un agente lógica cuya base de conocimiento esta representada por un programa lógico consistente P . Podemos decir que el agente *sabe* una fórmula F si es de que $P \Vdash_I F$. Esta interpretación tiene sentido pues, según Brouwer [6], una fórmula F puede leerse en intuicionismo como “Yo sé F ”.

Siguiendo esta idea podemos proponer también un conjunto de creencias B para el programa P . En la siguiente definición decimos que una teoría T es (literal) completa en la lógica X si para cada átomo $a \in \mathcal{L}_T$ se tiene ya sea que $T \Vdash_X a$ o $T \Vdash_X \neg a$.

Definición 3.6. Dada una teoría $T \in \text{Prp}$, decimos que la teoría $B \in \text{Prp}$, con $\mathcal{L}_B \subseteq \mathcal{L}_T$, es un *conjunto de creencias* para P si la teoría $T \cup \neg\neg B$ es consistente y completa en la lógica intuicionista.

Observe que los answer sets de la teoría T determinan todos los posibles conjuntos, esencialmente distintos, de conjuntos de creencias para T . Regresando al contexto de los agentes lógicos podemos decir que un agente puede también *crear* un conjunto de fórmulas si el suponer estas fórmulas es (i) consistente y (ii) suficiente para explicar todos los hechos de su mundo. Este hecho de “suponer” un conjunto de fórmulas corresponde a la doble negación en la teoría intuicionista $T \cup \neg\neg B$.

Note también que, en esta perspectiva, el cambiar la lógica base ya no es irrelevante. Por el Teorema 3.3 sabemos que, no importa la lógica intermedia propia que elijamos, el agente siempre llegará a *crear* las mismas cosas. Sin embargo los conjuntos de las fórmulas que el agente realmente *sabe* cambiarán de una lógica a otra. Las lógicas multivaluadas G_i no son particularmente convincentes para modelar nociones de conocimiento y, es por esto, que se prefiere el uso de la lógica intuicionista para este tipo de enfoques.

3.2.3. Min-Sets

En esta última sección retomamos la condición planteada por Pearce en el Teorema 3.1 para mostrar que, de hecho, esta condición estaba caracterizando la noción de answer sets que son, además, modelos mínimos. Resulta que, como ya era conocido, los answer sets de programas disyuntivos son siempre modelos mínimos. Introducimos entonces la semántica de los *min-sets* que, en capítulos posteriores, probarán ser de gran relevancia en el contexto de la programación lógica.

Definición 3.7. Dado un programa lógico $P \in \text{Prp}$ definimos la semántica de los *min-sets* como $\text{MS}(P) = \text{AS}(P) \cap \text{M}^m(P)$.

Observe que los min-sets (MS) no necesariamente deben coincidir con los answer sets mínimos (AS^m). Estas dos semánticas son, como lo muestra el programa lógico del siguiente ejemplo, distintas en general.

Ejemplo 3.3. Considere el siguiente programa libre P :

$$a \vee \neg a.$$

$$b \leftarrow a.$$

$$b \leftarrow \neg b.$$

Este programa tiene dos modelos clásicos, $\text{M}(P) = \{\{b\}, \{a, b\}\}$, y es claro que sólo uno de ellos es modelo mínimo, $\text{M}^m(P) = \{\{b\}\}$. Observe, por otra parte, que la semántica de answer sets captura al modelo $\text{AS}(P) = \{\{a, b\}\}$ y por consiguiente $\text{AS}^m(P) = \{\{a, b\}\}$. Sin embargo la semántica de min-sets, en este caso, es vacía pues $\text{MS}(P) = \text{AS}(P) \cap \text{M}^m(P) = \emptyset$.

Utilizando resultados sencillos de la teoría de conjuntos se puede establecer el diagrama de la Figura 3.1. En este diagrama la existencia de una flecha $\text{Sem}_1 \longrightarrow \text{Sem}_2$ entre dos

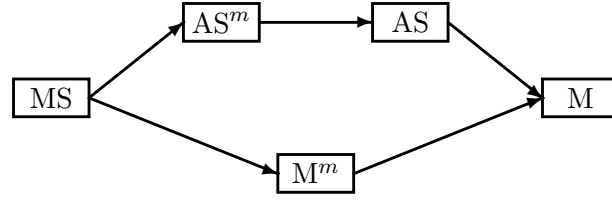


Figura 3.1: Comparación entre Semánticas

semánticas indica que, para todo programa lógico $P \in \text{Prp}$, se tiene $\text{Sem}_1(P) \subseteq \text{Sem}_2(P)$. Mientras que la carencia de dicha flecha indica que las semánticas son incomparables.

A partir de los resultados en las secciones anteriores podemos presentar también una caracterización para la semántica de los min-sets que corresponde, precisamente, con la condición que había propuesto Pearce en [33] para los programas disyuntivos.

Teorema 3.4. *Si $P \in \text{Prp}$, M es un conjunto de átomos y X es una lógica intermedia propia, entonces $M \in \text{MS}(P)$ si y solo si $P \cup \neg\widetilde{M} \Vdash_X M$.*

Demostración. Supongamos primero que $M \in \text{MS}(P) = \text{AS}(P) \cap \text{M}^m(P)$. Como el conjunto $M \in \text{M}^m(P)$ y por la Proposición 3.5 tenemos $P \cup \neg\widetilde{M} \Vdash_C M$. De las Proposiciones 2.3 y 2.1 se sigue que $P \cup \neg\widetilde{M} \Vdash_X \neg\neg M$. Pero como $M \in \text{AS}(P)$ y por la Definición 3.5 sabemos también que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$. De aquí se desprende entonces que $P \cup \neg\widetilde{M} \Vdash_X M$.

Para el recíproco suponemos que $P \cup \neg\widetilde{M} \Vdash_X M$. De la Proposición 2.1 tenemos inmediatamente que $P \cup \neg\widetilde{M} \Vdash_C M$ y, por la Proposición 3.5, $M \in \text{M}^m(P)$. También, como en general se tiene $\vdash_X a \rightarrow \neg\neg a$, se consigue mostrar que $P \cup \neg\widetilde{M} \cup \neg\neg M \Vdash_X M$ y, según la Definición 3.5, obtenemos $M \in \text{AS}(P)$. Así podemos concluir que $M \in \text{MS}(P)$. \square

Finalmente podemos establecer también una relación entre la semántica de answer sets y los min-sets que hemos estudiado en esta sección.

Corolario 3.2. *Sea $P \in \text{Prp}$ y $M \subseteq \mathcal{L}_P$. $M \in \text{AS}(P)$ si y solo si $M \in \text{MS}(P \cup \neg\neg M)$.*

Demostración. El resultado se desprende inmediatamente de las caracterizaciones provistas por los Teoremas 3.3 y 3.4. \square

Capítulo 4

Traducciones

En este capítulo presentamos algunas nociones de equivalencia entre programas lógicos como la equivalencia fuerte y la extensión conservativa. Revisamos como la equivalencia de programas en la semántica de answer sets se relaciona también, directamente, con equivalencia en la lógica de 3 valores G_3 . Utilizamos posteriormente estas nociones y sus respectivas caracterizaciones para construir traducciones entre clases de programas lógicos. Mostramos en particular una traducción que reduce teorías proposicionales arbitrarias a programas disyuntivos simples preservando propiedades importantes en la semántica.

4.1. Nociones de Equivalencia

En esta sección estudiaremos algunas nociones que permiten definir la idea de cuando dos programas lógicos son equivalentes. Definimos varios tipos de equivalencia y discutimos su importancia en aplicaciones de programación lógica.

4.1.1. Equivalencia Fuerte

Se pueden definir varias nociones de equivalencia entre programas lógicos. Una definición razonable, si estamos interesados en alguna semántica particular, podría ser que dos programas son *equivalentes* si ambos programas, compartiendo el mismo lenguaje, tienen exactamente los mismos modelos distinguidos por la semántica.

Definición 4.1. Sea Sem una semántica definida para la clase C y sean $P_1, P_2 \in C$ con lenguajes $\mathcal{L}_{P_1} = \mathcal{L}_{P_2}$. Decimos que los programas P_1 y P_2 son *equivalentes* en la semántica Sem , denotado $P_1 \equiv_{\text{Sem}} P_2$, si $\text{Sem}(P_1) = \text{Sem}(P_2)$.

Esta definición de equivalencia simple no tiene, sin embargo, algunas propiedades importantes que podríamos esperar de una relación de equivalencia. Particularmente estamos interesados en que, al reemplazar *localmente* una parte de un programa por otro conjunto “equivalente” de reglas, el programa original sea equivalente al programa modificado.

Esta propiedad de localidad nos permite concentrarnos en una sola parte del programa sabiendo que los cambios hechos no afectarán la semántica declarativa en el resto del programa. El siguiente ejemplo muestra que la equivalencia simple para la semántica de answer sets no cumple con esta propiedad.

Ejemplo 4.1. Consideremos los programas $P_1 = \{a \leftarrow \neg b\}$ y $P_2 = \{a, b \leftarrow b\}$. Estos dos programas son equivalentes ya que $\{a\}$ es el único answer set de ambos programas. Sin embargo, al reemplazar P_1 por P_2 dentro del programa $P = \{a \leftarrow \neg b, b \leftarrow a\}$, para obtener $P' = \{a, b \leftarrow b, b \leftarrow a\}$, perdemos esta equivalencia. Ahora P no tiene answer sets, mientras que el programa P' tiene exactamente un answer set: $\{a, b\}$.

Buscando una noción de equivalencia que sí cumpla con esta propiedad de localidad, se propuso en [17] el concepto de equivalencia fuerte.

Definición 4.2. [17] Sea Sem una semántica definida para la clase C y sean $P_1, P_2 \in C$ con lenguaje $\mathcal{L}_{P_1} = \mathcal{L}_{P_2}$. Decimos que los programas P_1 y P_2 son *fuertemente equivalentes* en la semántica Sem , denotado $P_1 \equiv_{\text{Sem}^*} P_2$, si para todo programa Q se tiene $P_1 \cup Q \equiv_{\text{Sem}} P_2 \cup Q$.

Ahora, si dos programas son fuertemente equivalentes, tenemos la seguridad que se puede reemplazar uno por el otro, dentro de algún otro programa más grande, sin modificar su significado semántico. También de la definición es claro que la equivalencia fuerte implica equivalencia, el Ejemplo 4.1 muestra que el recíproco de esta afirmación no necesariamente tendrá que ser cierto.

Una pregunta importante es entonces, dados dos programas lógicos, cómo determinar si son fuertemente equivalentes o no. Los mismos autores de [17] proporcionan una caracterización de la equivalencia fuerte para la semántica de answer sets, misma que presenta una interesante relación entre la programación lógica y las lógicas intermedias.

Teorema 4.1. [17] Si $P_1, P_2 \in \text{Aug}$, entonces $P_1 \equiv_{\text{AS}^*} P_2$ si y solo si $P_1 \equiv_{G_3} P_2$.

Observe que este resultado fue dado sólo para programas aumentados, bajo la definición de answer sets presentada en la Definición 2.5, a partir de su caracterización en términos de la lógica de equilibrio. Sin embargo es posible, a partir de las relaciones con nuestro enfoque como discutimos al final de la Sección 3.2.2, mostrar que el argumento utilizado en [17], es también válido para teorías proposicionales.

En esta sección presentamos, de cualquier modo, la demostración siguiendo nuestro mismo enfoque en lógicas intermedias para rescatar, al mismo tiempo, el resultado de equivalencia fuerte para los min-sets. Este resultado establece también fuertes indicios, al igual que la versión presentada en [17], para mostrar la utilidad de las lógicas intermedias en el contexto de la programación lógica.

En esta demostración consideramos tautologías, modelos e interpretaciones de 3-valores basadas en la lógica G_3 . Dada una interpretación I definimos la interpretación I' que evalúa, a cada átomo $a \in \mathcal{L}$, como $I'(a) = I(\neg\neg a)$. Observe que, mediante una inducción directa, se puede demostrar que, para cualquier fórmula F , $I'(F) = I(\neg\neg F)$. En particular, si I es un modelo de F también I' es un modelo de F y, si $I(F) = 0$, entonces $I(\neg\neg F) = 0$. El

siguiente resultado nos muestra cómo dos teorías son equivalentes en la lógica G_3 si y solo si tienen los mismos modelos.

Proposición 4.1. *Sean T_1 y T_2 dos teorías. $T_1 \equiv_{G_3} T_2$ si y solo si T_1 y T_2 tienen los mismos modelos.*

Demostración. Observe que el hecho de que $T_1 \equiv_{G_3} T_2$ es equivalente a que, para toda interpretación I , se tenga $I(T_1) = I(T_2)$. De esto se sigue inmediatamente que todos los modelos de T_1 son también modelos de T_2 y viceversa.

Demostraremos el contra recíproco de la otra implicación. Supongamos que $T_1 \not\equiv_{G_3} T_2$, entonces existe una interpretación I tal que $I(T_1) \neq I(T_2)$. Si I es modelo de T_1 (o de T_2) entonces es claro que no puede ser modelo de T_2 (resp. de T_1) y por lo tanto T_1 y T_2 tienen modelos distintos. Si I no es modelo de ninguna de las teorías T_1 o T_2 entonces el único caso posible es que $I(T_1) = 1$ y $I(T_2) = 0$ (o viceversa), pero entonces la interpretación I' sería un modelo de T_1 y no de T_2 . En cualquier caso T_1 y T_2 tienen modelos distintos. \square

El siguiente par de lemas son de utilidad para determinar, en algunos casos especiales, la relación que existe entre los modelos de 3 valores de una teoría y sus answer sets o min-sets correspondientes.

Lema 4.1. *Sea T una teoría con un único modelo I . Si $a \in \mathcal{L}_T$ y $I(a) = 2$ entonces se tiene $T \vdash_{G_3} a$.*

Demostración. Observe que la interpretación I debe de ser definida pues, de otro modo, I' sería también modelo de T contradiciendo la hipótesis de que tiene un solo modelo. Basta mostrar entonces que, para toda interpretación K , se tiene $K(T) \leq K(a)$. Si $K(a) = 2$ el resultado es trivial. Si $K(a) = 1$ entonces, como $I \neq K$ y I es el único modelo de T , se tiene $K(T) \leq 1$. Similarmente, si $K(a) = 0$, debemos tener que $K(T) = 0$ pues, de otro modo, K' sería un modelo de T distinto de I . \square

En el lema siguiente utilizamos, dada una interpretación I , la notación M_I para denotar al conjunto $M_I = \{a \in \mathcal{L} \mid I(a) > 0\}$. Observe que, por construcción, I es un modelo de los conjuntos $\neg \widetilde{M}_I$ y $\neg \neg M_I$. Para la demostración se consideran las caracterizaciones de las semánticas de answer sets y min-sets, en términos de la lógica intermedia G_3 , provistas por los Teoremas 3.3 y 3.4.

Lema 4.2. *Si $P \in \text{Prp}$ entonces:*

1. *Si P tiene un único modelo I entonces $M_I \in \text{AS}(P)$ y $M_I \in \text{MS}(P)$.*
2. *Si P no tiene modelos entonces $\text{AS}(P) = \text{MS}(P) = \emptyset$.*
3. *Si I es un modelo indefinido de P entonces $M_I \notin \text{AS}(P)$ y $M_I \notin \text{MS}(P)$.*

Demostración. (1) Si I es el único modelo de P entonces, por el Lema 4.1, tenemos que $P \Vdash_{G_3} M_I$. Es claro que podemos agregar las premisas $\neg\widetilde{M}_I$ y $\neg\neg M_I$ preservando consistencia, de modo que $P \cup \neg\widetilde{M}_I \cup \neg\neg M_I \Vdash_{G_3} M_I$ y también $P \cup \neg\widetilde{M}_I \Vdash_{G_3} M_I$. (2) Se sigue de que, como el programa no tiene modelos, tampoco tiene ningún modelo clásico y, por lo tanto, es inconsistente. (3) Como existe un átomo $a \in M_I$ tal que $I(a) = 1$ llegamos a la conclusión de que $P \cup \neg\widetilde{M}_I \not\vdash_{G_3} a$ y tampoco $P \cup \neg\widetilde{M}_I \cup \neg\neg M_I \not\vdash_{G_3} a$. \square

La prueba del resultado de equivalencia fuerte se basa en que, si tenemos dos programas que no son equivalentes en la lógica G_3 , podemos construir un nuevo programa P tal que, al agregarlo a los otros dos, genere programas con answer sets y min-sets distintos. La siguiente definición nos ayuda a construir dicho programa.

Definición 4.3. Si I es una interpretación definimos el programa P_I como el programa más pequeño que satisface:

- Si $I(a) = 0$ entonces $\perp \leftarrow a \in P_I$.
- Si $I(a) = I(b) = 1$ entonces $a \leftarrow b \in P_I$.
- Si $I(a) = 1$ entonces $a \leftarrow \neg a \in P_I$.¹
- Si $I(a) = 2$ entonces $a \in P_I$.

Observe que, por construcción, los únicos modelos de P_I son precisamente I y I' .

Lema 4.3. Sean $P_1, P_2 \in \text{Prp}$ dos programas tales que $\mathcal{L}_{P_1} = \mathcal{L}_{P_2}$. Si $P_1 \not\equiv_{G_3} P_2$ entonces existe $P \in \text{Gen}$ tal que $\text{AS}(P_1 \cup P) \neq \text{AS}(P_2 \cup P)$ y $\text{MS}(P_1 \cup P) \neq \text{MS}(P_2 \cup P)$.

Demostración. Si I es una interpretación que modela a P_1 y no a P_2 diremos que I es una interpretación testigo. Por la Proposición 4.1, y sin pérdida de generalidad, podemos asumir que existe al menos una interpretación testigo. Separamos la prueba en dos casos:

1. *Si existe una interpretación testigo definida.* Sea I dicha interpretación, como $I = I'$ es el único modelo de P_I entonces también I es el único modelo de $P_1 \cup P_I$. La Proposición 4.2 demuestra que $M_I \in \text{AS}(P_1 \cup P_I)$ y $M_I \in \text{MS}(P_1 \cup P_I)$. Mientras que el programa $P_2 \cup P_I$ no tiene modelos ya que la única interpretación que modela a P_I no modela a P_2 así que, también por la Proposición 4.2, $M_I \notin \text{AS}(P_2 \cup P_I)$ y $M_I \notin \text{MS}(P_2 \cup P_I)$.

2. *Si todas las interpretaciones testigo son indefinidas.* Sea I una de esas interpretaciones. Como I es indefinida sabemos, de la Proposición 4.2, que $M_I \notin \text{AS}(P_1 \cup P_I)$ y $M_I \notin \text{MS}(P_1 \cup P_I)$. Por otra parte la interpretación I' debe ser un modelo de P_2 pues, de otro modo, I' sería un modelo definido de P_1 y no de P_2 contradiciendo la hipótesis de este caso. Ahora recuerde que P_I sólo tiene dos modelos I y I' así que, como I no es modelo de P_2 , tenemos que I' es el único modelo de $P_2 \cup P_I$. Así concluimos, de nuevo por la Proposición 4.2 y como $M_I = M_{I'}$, que $M_I \in \text{AS}(P_2 \cup P_I)$ y $M_I \in \text{MS}(P_2 \cup P_I)$. \square

¹Este caso no es necesario si restringimos nuestra atención a la semántica de answer sets. Podemos omitir estas reglas ya que son equivalentes a las $\neg a$ que aparecen en la completación del programa al caracterizar los answer sets.

El lema anterior se encarga, precisamente, de una de las implicaciones de nuestro teorema principal. La otra de las implicaciones es una consecuencia inmediata de las definiciones de answer sets y min-sets en términos de lógicas intermedias.

Teorema 4.2. *Sean $P_1, P_2 \in \text{Prp}$ tales que $\mathcal{L}_{P_1} = \mathcal{L}_{P_2}$. Entonces $P_1 \equiv_{\text{AS}^*} P_2$ si y solo si $P_1 \equiv_{\text{MS}^*} P_2$ si y solo si $P_1 \equiv_{\text{G}_3} P_2$.*

Demostración. Si $P_1 \equiv_{\text{G}_3} P_2$ entonces también, para cualquier programa $P \in \text{Prp}$, tenemos que $P_1 \cup P \equiv_{\text{G}_3} P_2 \cup P$. Por lo tanto podemos reemplazar a $P_1 \cup P$ con $P_2 \cup P$ en las definiciones de answer sets y min-sets para obtener, respectivamente, que $P_1 \equiv_{\text{AS}^*} P_2$ y $P_1 \equiv_{\text{MS}^*} P_2$. Para la otra implicación supongamos que $P_1 \not\equiv_{\text{G}_3} P_2$, entonces el Lema 4.3 nos ayuda a construir un tercer programa que muestra $P_1 \not\equiv_{\text{AS}^*} P_2$ y $P_1 \not\equiv_{\text{MS}^*} P_2$. \square

4.1.2. Extensión Conservativa

Una de las aplicaciones importantes de las nociones de equivalencia presentadas en este capítulo es que nos permiten simplificar programas lógicos. Supongamos que tenemos un programa lógico P , con una estructura más o menos complicada, y queremos calcular los modelos distinguidos de alguna semántica. Sería de gran utilidad si podemos encontrar un programa más simple P' , equivalente a P , calcular la semántica de P' y recuperar entonces la semántica del programa original P .

Sin embargo la condición $\text{Sem}(P) = \text{Sem}(P')$ que impone la Definición 4.1 de equivalencia puede ser demasiado restrictiva. Sería suficiente si, conociendo la semántica $\text{Sem}(P')$ fuera posible recuperar los modelos en $\text{Sem}(P)$ mediante alguna relación sencilla y fácil de calcular. Un ejemplo de este tipo de equivalencia, presentada en [3], es conocida como extensión conservativa.

Definición 4.4. Sea Sem una semántica para la clase C y sean $P, P' \in C$ programas tales que $\mathcal{L}_P \subseteq \mathcal{L}_{P'}$. Decimos que P' es una *transformación conservativa* de P en la semántica Sem , y lo denotamos $P \xrightarrow{\text{Sem}} P'$, si se cumple la condición

$$\text{Sem}(P) = \{M \cap \mathcal{L}_P \mid M \in \text{Sem}(P')\} .$$

Si P_2 es una transformación conservativa de P_1 y, además, se satisface $P_1 \subseteq P_2$ decimos que P_2 es una *extensión conservativa* de P_1 . Baral presenta en [3], precisamente, la definición de extensión conservativa. La definición que presentamos aquí es, en cierto sentido, más general pues admite una familia más amplia de transformaciones y conserva las propiedades esenciales de la definición original.

Observe que la transformación conservativa permite agregar nuevos átomos al lenguaje del programa P para tratar de conseguir un programa más simple P' . La condición que define a la transformación conservativa dice que, si ignoramos estos nuevos átomos introducidos en los modelos de P' , obtenemos exactamente los modelos de la semántica del programa original P . A este conjunto de átomos nuevos $\mathcal{L}_{P'} \setminus \mathcal{L}_P$ le llamamos el conjunto de *átomos reservados* de la transformación. Observe también que, si la transformación no tiene átomos

reservados, la transformación conservativa coincide exactamente con la equivalencia simple de la Definición 4.1.

Una de las primeras propiedades que podríamos pedir a la transformación conservativa es, por supuesto, transitividad.

Proposición 4.2. *Sea Sem una semántica para la clase C y sean $P, Q, R \in C$ programas lógicos. Si $P \xrightarrow{\text{Sem}} Q$ y $Q \xrightarrow{\text{Sem}} R$ entonces $P \xrightarrow{\text{Sem}} R$.*

Demostración. Sea $M \in \text{Sem}(P)$ luego, como $P \xrightarrow{\text{Sem}} Q$, existe $M' \in \text{Sem}(Q)$ tal que $M = M' \cap \mathcal{L}_P$ y, como $Q \xrightarrow{\text{Sem}} R$, existe $M'' \in \text{Sem}(R)$ tal que $M' = M'' \cap \mathcal{L}_Q$. De aquí se tiene, como $\mathcal{L}_P \subseteq \mathcal{L}_Q$, que $M = M'' \cap \mathcal{L}_P$ para un modelo $M'' \in \text{Sem}(R)$. Similarmente, si $M \in \text{Sem}(R)$ entonces $M \cap \mathcal{L}_Q \in \text{Sem}(Q)$ y $(M \cap \mathcal{L}_Q) \cap \mathcal{L}_P = M \cap \mathcal{L}_P \in \text{Sem}(P)$. \square

Otra propiedad, un tanto más peculiar, que poseen las transformaciones conservativas es cierta forma de transitividad “inversa”. La proposición siguiente presenta de manera formal esta idea.

Proposición 4.3. *Sea Sem una semántica para la clase C y sean $P, Q, R \in C$ programas lógicos tales que $\mathcal{L}_P \subseteq \mathcal{L}_R$. Si $P \xrightarrow{\text{Sem}} Q$ y $R \xrightarrow{\text{Sem}} Q$ entonces $P \xrightarrow{\text{Sem}} R$.*

Demostración. Sea $M \in \text{Sem}(P)$ luego, como $P \xrightarrow{\text{Sem}} Q$, existe $M' \in \text{Sem}(Q)$ tal que $M = M' \cap \mathcal{L}_P$ y, como $R \xrightarrow{\text{Sem}} Q$, sabemos que $M' \cap \mathcal{L}_R \in \text{Sem}(R)$. Ahora, como $\mathcal{L}_P \subseteq \mathcal{L}_R$, tenemos que $M = (M' \cap \mathcal{L}_R) \cap \mathcal{L}_P$ para un modelo $M' \cap \mathcal{L}_R \in \text{Sem}(R)$.

Similarmente, si $M \in \text{Sem}(R)$ existe $M' \in \text{Sem}(Q)$ tal que $M = M' \cap \mathcal{L}_R$ y sabemos que $M' \cap \mathcal{L}_P \in \text{Sem}(P)$. Entonces $M \cap \mathcal{L}_P = (M' \cap \mathcal{L}_R) \cap \mathcal{L}_P = M' \cap \mathcal{L}_P \in \text{Sem}(P)$. \square

Las siguientes dos proposiciones establecen casos particulares de extensiones conservativas que, como veremos pronto y combinadas con el Teorema 4.2, servirán para justificar las propiedades semánticas de las traducciones en la siguiente sección.

Proposición 4.4. *Sea A un conjunto de átomos y sea $L = \{a \leftarrow a \mid a \in A\}$. Si $P \in \text{Prp}$ entonces $P \xrightarrow{\text{AS}} P \cup L$ y $P \xrightarrow{\text{MS}} P \cup L$.*

Demostración. Mostraremos que, de hecho, las semánticas satisfacen $\text{MS}(P) = \text{MS}(P \cup L)$ y $\text{AS}(P) = \text{AS}(P \cup L)$. Revisamos primero el caso de los min-sets.

Si $M \in \text{MS}(P)$ entonces, por el Teorema 3.4, $P \cup \neg(\mathcal{L}_P \setminus M) \Vdash_I M$. Como el programa $P \cup \neg(\mathcal{L}_P \setminus M)$ es consistente tiene un modelo clásico y, si extendemos ese modelo evaluando a cero todos los átomos en $A \setminus \mathcal{L}_P$, obtenemos que $P \cup \neg(\mathcal{L}_P \setminus M) \cup \neg(A \setminus \mathcal{L}_P)$ también es consistente. Además, como $M \subseteq \mathcal{L}_P$, se tiene que $(\mathcal{L}_P \setminus M) \cup (A \setminus \mathcal{L}_P) = \mathcal{L}_{P \cup L} \setminus M$. Luego, como L es un conjunto de teoremas intuicionistas, llegamos a que $P \cup L \cup \neg(\mathcal{L}_{P \cup L} \setminus M) \Vdash_I M$ y, de nuevo por el Teorema 3.4, concluimos que $M \in \text{MS}(P \cup L)$.

De manera similar, si $M \in \text{MS}(P \cup L)$, entonces $P \cup L \cup \neg(\mathcal{L}_{P \cup L} \setminus M) \Vdash_I M$ y, removiendo al conjunto de teoremas L , $P \cup \neg(\mathcal{L}_{P \cup L} \setminus M) \Vdash_I M$. Observe ahora que $M \subseteq \mathcal{L}_P$ pues, si existiera un átomo $a \in M$ con $a \notin \mathcal{L}_P$ entonces a no aparecería en el conjunto de premisas

del enunciado intuicionista anterior y no podría ser demostrado. De aquí se sigue, de nuevo, que $(\mathcal{L}_P \setminus M) \cup (A \setminus \mathcal{L}_P) = \mathcal{L}_{P \cup L} \setminus M$. Así llegamos a que $P \cup \neg(\mathcal{L}_P \setminus M) \cup \neg(A \setminus \mathcal{L}_P) \Vdash_I M$. Pero los átomos del conjunto $A \setminus \mathcal{L}_P$ ya no aparecen en ningún otro sitio dentro de la prueba intuicionista y, por el Lema 3.3, obtenemos que $P \cup \neg(\mathcal{L}_P \setminus M) \Vdash_I M$. Por lo tanto podemos concluir que $M \in \text{MS}(P)$.

Observe ahora que si $M \in \text{AS}(P)$ entonces, por el Corolario 3.2, $M \in \text{MS}(P \cup \neg\neg M)$. Pero, del resultado anterior, $\text{MS}(P \cup \neg\neg M) = \text{MS}(P \cup \neg\neg M \cup L)$. Así que también tenemos $M \in \text{MS}(P \cup L \cup \neg\neg M)$ y, por lo tanto, $M \in \text{AS}(P \cup L)$. \square

Proposición 4.5. *Sea $P \in \text{Prp}$, F una fórmula con $\mathcal{L}_F \subseteq \mathcal{L}_P$ y un átomo $x \notin \mathcal{L}_P$. Entonces se tiene que $P \xrightarrow{\text{AS}} P \cup \{x \leftrightarrow F\}$.*

Demostración. Tomemos primero $M \in \text{AS}(P \cup \{x \leftrightarrow F\})$, por la Definición 3.5 tenemos que $P \cup \{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \cup \{x\} \setminus M) \cup \neg\neg M \Vdash_I M$. Si hacemos $M' = M \cap \mathcal{L}_P = M \setminus x$ y

$$\Delta = \begin{cases} \{\neg\neg x\} & \text{si } x \in M \\ \{\neg x\} & \text{si } x \notin M \end{cases}$$

el enunciado intuicionista puede escribirse $P \cup \{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \setminus M') \cup \neg\neg M' \cup \Delta \Vdash_I M'$. Pero recuerde entonces, como $\mathcal{L}_F \subseteq \mathcal{L}_P$, $\neg(\mathcal{L}_P \setminus M') \cup \neg\neg M' \Vdash_I \neg\neg F$ (ó $\neg F$). Por razones de consistencia el número de negaciones de F en el enunciado anterior debe coincidir con las negaciones de x en Δ , así se sigue que $\{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \setminus M') \cup \neg\neg M' \Vdash_I \Delta$. De este modo el conjunto Δ no es necesario en la prueba intuicionista y podemos, eliminándolo, llegar a que $P \cup \{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \setminus M') \cup \neg\neg M' \Vdash_I M'$. Observe finalmente que x ahora sólo aparece en la fórmula $x \leftrightarrow F$ entre las premisas, podemos reemplazar entonces todas las ocurrencias de x en la prueba formal intuicionista por la fórmula F y obtener la prueba de $P \cup \{F \leftrightarrow F\} \cup \neg(\mathcal{L}_P \setminus M') \cup \neg\neg M' \Vdash_I M'$. Ahora es claro que podemos eliminar la tautología $F \leftrightarrow F$ para llegar a $P \cup \neg(\mathcal{L}_P \setminus M') \cup \neg\neg M' \Vdash_I M'$ y, por la Definición 3.5, concluimos que $M' \in \text{AS}(P)$.

Para la otra contención tomemos $M \in \text{AS}(P)$ de modo que, por definición, tenemos el enunciado intuicionista $P \cup \neg(\mathcal{L}_P \setminus M) \cup \neg\neg M \Vdash_I M$. Ahora tenemos, por la Proposición 2.5, que $M \cup \neg(\mathcal{L}_P \setminus M) \Vdash_I F$ (ó $\neg F$) y, a partir del enunciado intuicionista anterior, también tenemos que $P \cup \neg(\mathcal{L}_P \setminus M) \cup \neg\neg M \Vdash_I F$ (ó $\neg F$). Como $P \cup \neg(\mathcal{L}_P \setminus M) \cup \neg\neg M$ es consistente, y tiene al menos un modelo clásico, es fácil ver que $P \cup \{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \setminus M) \cup \neg\neg M$ también es una teoría consistente pues, según se haya probado F ó $\neg F$, podemos extender el modelo clásico asignándole a x el valor 1 ó 0 respectivamente. Observe ahora que, si tenemos $M \cup \neg(\mathcal{L}_P \setminus M) \Vdash_I F$, también tenemos que $P \cup \{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \setminus M) \cup \neg\neg M \Vdash_I x$. Haciendo finalmente $M' = M \cup \{x\}$ o $M' = M$ según el caso, y agregando las premisas necesarias $\neg\neg x$ ó $\neg x$, podemos concluir que $P \cup \{x \leftrightarrow F\} \cup \neg(\mathcal{L}_P \cup \{x\} \setminus M') \cup \neg\neg M' \Vdash_I M'$. Según la Definición 3.5, hemos llegado a $M' \in \text{AS}(P \cup \{x \leftrightarrow F\})$. \square

Observe que este último resultado no es válido para la semántica de los min-sets. El programa $P = \{a \vee \neg a\}$ tiene un min-set $\text{MS}(P) = \{\emptyset\}$, mientras que el segundo programa lógico $P' = \{a \vee \neg a, x \leftrightarrow \neg a\}$ tiene $\text{MS}(P) = \{\{x\}, \{a\}\}$. Aún ignorando al nuevo átomo

introducido x estamos obteniendo un modelo, $\{a\}$, adicional. La propiedad de agregar nuevos átomos para definir fórmulas y la minimalidad de la semántica son incompatibles. Este es un punto en el cual la semántica de answer sets se muestra, ya que sí permite introducir nuevos átomos para definir fórmulas, intuitivamente más correcta.

Las transformaciones conservativas, ya que tienen la libertad de extender el lenguaje de los programas lógicos, pueden realizar simplificaciones que, de otro modo, no serían posibles. Sin embargo no satisfacen, en general, la propiedad de localidad que tiene, por ejemplo, la equivalencia fuerte.

El mismo Ejemplo 4.1 presentado en la sección anterior muestra, ya que le equivalencia simple es un caso particular de la transformación conservativa, que no siempre es posible aplicar de manera local una de estas transformaciones sin modificar la semántica declarativa del programa. Entonces, de manera análoga a la equivalencia fuerte, podemos presentar la noción de una transformación conservativa fuerte.

Definición 4.5. Sea Sem una semántica para la clase C y sean $P, P' \in C$ programas tales que $\mathcal{L}_P \subseteq \mathcal{L}_{P'}$. Decimos que P' es una *transformación conservativa fuerte* de P en la semántica Sem , y lo denotamos $P \xrightarrow{\text{Sem}^*} P'$, si para cada programa Q , con la propiedad de que $\mathcal{L}_Q \cap (\mathcal{L}_{P'} \setminus \mathcal{L}_P) = \emptyset$, se tiene que $P \cup Q \xrightarrow{\text{Sem}} P' \cup Q$.

Observe que la condición $\mathcal{L}_Q \cap (\mathcal{L}_{P'} \setminus \mathcal{L}_P) = \emptyset$ establece que los programas Q , utilizados para complementar al programa P , no deben contener ninguno de los átomos reservados de la transformación. En una implementación del sistema esta condición podría garantizarse definiendo un conjunto especial de átomos cuyo uso esté reservado para transformaciones internas y que no estén disponibles para que el usuario final escriba programas lógicos con ellos. La transformación conservativa fuerte satisface, como lo podríamos esperar, transitividad.

Proposición 4.6. Sea Sem una semántica para la clase C y sean $P, Q, R \in C$ programas lógicos. Si $P \xrightarrow{\text{Sem}^*} Q$ y $Q \xrightarrow{\text{Sem}^*} R$ entonces $P \xrightarrow{\text{Sem}^*} R$.

Demostración. Sea T un programa con $\mathcal{L}_T \cap (\mathcal{L}_R \setminus \mathcal{L}_P) = \emptyset$. Observe que, como los lenguajes satisfacen $\mathcal{L}_P \subseteq \mathcal{L}_Q \subseteq \mathcal{L}_R$, entonces tenemos que $\mathcal{L}_T \cap (\mathcal{L}_Q \setminus \mathcal{L}_P) = \mathcal{L}_T \cap (\mathcal{L}_R \setminus \mathcal{L}_Q) = \emptyset$. Ahora, como $P \xrightarrow{\text{Sem}^*} Q$ y $Q \xrightarrow{\text{Sem}^*} R$, tenemos que $P \cup T \xrightarrow{\text{Sem}} Q \cup T$ y $Q \cup T \xrightarrow{\text{Sem}} R \cup T$. De la Proposición 4.2 se sigue que $P \cup T \xrightarrow{\text{Sem}} R \cup T$. \square

Observe también que las Proposiciones 4.4 y 4.5 pueden generalizarse inmediatamente al caso de transformaciones conservativas fuertes.

Proposición 4.7. Sea A un conjunto de átomos y sea $L = \{a \leftarrow a \mid a \in A\}$. Si $P \in \text{Prp}$ entonces $P \xrightarrow{\text{AS}^*} P \cup L$ y $P \xrightarrow{\text{MS}^*} P \cup L$.

Proposición 4.8. Sea $P \in \text{Prp}$, F una fórmula con $\mathcal{L}_F \subseteq \mathcal{L}_P$ y un átomo $x \notin \mathcal{L}_P$. Entonces se tiene que $P \xrightarrow{\text{AS}^*} P \cup \{x \leftrightarrow F\}$.

Demostración. Ambos resultados son ciertos ya que los dos enunciados consideran el caso $P \subseteq P'$, donde P' es la transformación conservativa de cada proposición. Si Q es un programa, con $\mathcal{L}_Q \cap (\mathcal{L}_{P'} \setminus \mathcal{L}_P) = \emptyset$, entonces las mismas Proposiciones 4.4 y 4.5 demuestran que $P \cup Q \xrightarrow{\text{AS}} P' \cup Q$ y $P \cup Q \xrightarrow{\text{MS}} P' \cup Q$ en cada uno de los casos. \square

4.2. Traducciones de Programas

En esta sección presentaremos y estudiaremos diversas traducciones entre programas lógicos que son de gran utilidad. Una traducción, como definiremos en seguida, es simplemente una función entre dos clases de programas lógicos. Nos interesará, por supuesto, determinar aquellas propiedades que son importantes respecto a la semántica o el contexto lógico particular que estemos considerando.

Las traducciones de programas pueden ser muy interesantes por varias y distintas razones. Las traducciones nos pueden ser útiles para simplificar la estructura de los programas lógicos [26, 34, 36], derivar programas correctos a partir de especificaciones [35], e incluso para realizar actualizaciones de programas y revisión de conocimiento para sistemas basados en agentes lógicos [1, 10].

Dadas dos clases de programas lógicos C y C' , una traducción de C a C' es simplemente una función $\text{Tr}: C \rightarrow C'$. En referencias como [15, 34] se han estudiado propiedades importantes de las traducciones de programas.

Definición 4.6. [15, 34] Dada una semántica Sem definida para la clase D y dadas dos clases $C, C' \subseteq D$ cerradas bajo uniones², una traducción $\text{Tr}: C \rightarrow C'$ es:

- *polinomial* si el tiempo requerido para calcular $\text{Tr}(P)$, para cualquier programa $P \in C$, es polinomial respecto al número de conectivos en P ;
- *fiel* si para todo programa $P \in C$ se tiene que $P \xrightarrow{\text{Sem}} \text{Tr}(P)$;
- *fuertemente fiel* si para todo programa $P \in C$ se tiene que $P \xrightarrow{\text{Sem}^*} \text{Tr}(P)$;
- *modular* si, dados $P_1, P_2 \in C$ se tiene que $\text{Tr}(P_1 \cup P_2) = \text{Tr}(P_1) \cup \text{Tr}(P_2)$; y
- *modular reductiva* si es modular, se satisface $C' \subseteq C$ y, para todo $P \in C'$, se tiene que $\text{Tr}(P) = P'$.³

La característica de ser polinomial (P) tiene que ver con el orden de complejidad que tendría una implementación actual de la traducción en un sistema de cómputo. Una traducción fiel (F) preserva, salvo la introducción de nuevos átomos al lenguaje, la semántica del programa original. Mientras que una traducción fuertemente fiel (S) nos garantiza,

²Una clase de programas C es cerrada bajo uniones si $P_1, P_2 \in C$ implica que $P_1 \cup P_2 \in C$

³Las definiciones originales de una traducción modular no coinciden en [15] y [34]. La traducción modular que presentamos aquí es la dada en [34], mientras que la traducción modular, más rica en propiedades, presentada en [15] la introducimos aquí con el nombre de *modular reductiva*.

además, que la traducción se puede aplicar de manera local a un subconjunto de reglas de un programa sin modificar la semántica declarativa.

El hecho de que una traducción sea modular (M) tiene que ver con el hecho de que puede aplicarse “*por partes*” a un programa lógico. Una traducción modular reductiva (R) agrega dos condiciones adicionales: la traducción debe de mapear programas de una clase a otra, en principio, más simple; además, si un programa se encuentra ya en la clase simple, no debe ser modificado por la traducción. Observe que estas propiedades no dependen de la semántica elegida y se refieren, más bien, a la *forma* de la traducción.

A manera de notación decimos que una traducción es PFM si es simultáneamente polinomial, fiel y modular. Análogamente podemos decir una traducción es PSM, si es además fuertemente fiel, o PSR, si la traducción es polinomial, fuertemente fiel y modular reductiva. Uno de los resultados más importantes de este trabajo es, precisamente, exhibir una traducción PSM de teorías proposicionales a la clase de programas lógicos disyuntivos. En las siguientes secciones se discuten diversos ejemplos de traducciones interesantes.

4.2.1. Reducción de Implicaciones Anidadas

Si una fórmula F tiene una subfórmula propia de la forma $A \rightarrow B$, donde A y B no contienen ya más implicaciones, decimos que $A \rightarrow B$ es una *implicación anidada elemental* de la fórmula F . Así podemos definir entonces la traducción PrpAug que elimina todas las implicaciones anidadas elementales de una teoría proposicional.

Definición 4.7. La traducción PrpAug: Prp \rightarrow Aug se define recursivamente, para cada teoría $T \in$ Prp, como sigue:

- Si no hay fórmulas $F \in T$ con implicaciones anidadas elementales, entonces ya tenemos que $T \in$ Aug y se define PrpAug(T) = T .
- Sea $F \in T$ una fórmula tenga una implicación anidada elemental $A \rightarrow B$. Tomemos un nuevo átomo $x \in \mathcal{L} \setminus \mathcal{L}_T$, sea F' la fórmula que se obtiene al reemplazar la ocurrencia de $A \rightarrow B$ en F por el átomo x , y sea T' la teoría obtenida al reemplazar F por la fórmula F' . Sea también el conjunto $\Delta = \{x \wedge A \rightarrow B, \neg A \vee B \rightarrow x, x \vee A \vee \neg B\}$. Definimos entonces: PrpAug(T) = PrpAug(T') \cup Δ .

Observe que en cada paso de la reducción se elimina exactamente una implicación anidada elemental y no se agrega ninguna nueva. Esto justifica que la recursión está bien fundamentada y la traducción bien definida.

Proposición 4.9. La traducción PrpAug: Prp \rightarrow Aug es PSR en la semántica AS.

Demostración. El hecho de que la traducción sea polinomial se sigue de que debe aplicarse exactamente una vez por cada implicación anidada en el programa, y este número esta acotado por el número de conectivos en el programa. Para justificar que el paso recursivo es fuertemente fiel observe que, por la Proposición 4.8, $P \xrightarrow{\text{AS}^*} P \cup \{x \leftrightarrow (A \rightarrow B)\}$. Luego del Teorema 4.2, a partir de la equivalencia $P \cup \{x \leftrightarrow (A \rightarrow B)\} \equiv_{G_3} P' \cup \Delta$, y por inducción

sobre el número de conectivos en P , podemos concluir que $P \xrightarrow{\text{AS}^*} \text{PrpAug}(P') \cup \Delta$. También es fácil ver que la traducción es modular reductiva ya que la traducción se va resolviendo fórmula por fórmula y, en el caso base, los programas que ya están en la clase de aumentados no son modificados. \square

Las implementaciones actuales de programación lógica suelen restringir el lenguaje a la clase de programas disyuntivos. Una práctica común era representar frases, que intuitivamente corresponden a la forma “ A implica B ”, con un nuevo átomo x y agregando al programa el par de reglas: $x \leftarrow B$, $x \leftarrow \neg A$. Esta práctica, sin embargo, a veces conducía a la aparición de modelos que, también intuitivamente, no se esperaban (o, similarmente, la falta de modelos que sí se esperaban) al calcular la semántica de los answer sets.

Es importante incluir la cláusula $x \vee A \vee \neg B$ en el conjunto Δ para representar correctamente la implicación anidada ya que, de otro modo, no se satisface la equivalencia necesaria en la lógica G_3 . Esta puede ser la explicación de porque las representaciones utilizadas anteriormente podían conducir a la obtención de resultados inesperados. Esto muestra la relevancia que tienen resultados como el Teorema 4.2, que ofrecen una clara relación entre la semántica de answer sets y la lógica matemática. La fórmula $x \vee A \vee \neg B$, necesaria para representar correctamente la implicación, fue descubierta, de hecho, examinando los modelos en G_3 de la fórmula original $x \leftrightarrow (A \rightarrow B)$.

4.2.2. Teorías Proposicionales a Programas Generales

La transformación presentada en la sección anterior nos permite eliminar las implicaciones anidadas de las teorías proposicionales para traducirlas a la clase de programas aumentados. En [34] se presenta una traducción PSM que nos permite simplificar programas aumentados a la clase más simple de programas generales⁴.

Componiendo este par de traducciones es posible reducir cualquier teoría proposicional, en tiempo polinomial y preservando las propiedades importantes de la semántica, a un programa general. Otra traducción muy sencilla y bien conocida, que veremos en la siguiente sección, puede terminar el trabajo y construir un programa puramente disyuntivo.

Será interesante presentar, de cualquier modo, una versión más general de la traducción presentada en [34] que integre, de una vez, los resultados que hemos podido conseguir en la sección anterior. La Definición 4.8, que presentamos en seguida, ofrece varias ventajas con respecto a la de [34]. Está primero el hecho, por supuesto, de que la traducción que aquí presentamos considera ya a la clase completa de teorías proposicionales como domino (y no sólo la clase de programas aumentados).

La traducción original esta dada, además, en función de cuatro sub-traducciones primitivas, mientras que la traducción que aquí proponemos realiza (basándonos principalmente en el modelo de su segunda sub-traducción) todos los cambios necesarios de una sola vez. La prueba de que la traducción es correcta, en particular de que es PSM, será también mucho

⁴Los autores de [34] usan el término “disyuntivo” pero la clase de programas que consideran es justamente la que aquí definimos como “generales”.

más sencilla que la presentada en [34] gracias a los resultados de equivalencia obtenidos en nuestro trabajo y presentados en la Sección 4.1.

Definición 4.8. Dada una teoría $T \in \text{Prp}$, definimos al conjunto \mathcal{F}_T como el conjunto de todas las subfórmulas que ocurren en T . Sea entonces $\Lambda: \mathcal{F}_T \rightarrow \mathcal{L} \setminus \mathcal{L}_T$ una función uno a uno que asigne un átomo nuevo y distinto a cada fórmula en \mathcal{F}_T ⁵. Definimos entonces la traducción $\text{PrpGen}: \text{Prp} \rightarrow \text{Gen}$ como $\text{PrpGen}(T) = \{\Lambda_F \mid F \in T\} \cup \Delta_P$, donde el programa general Δ_P se construye como sigue:

1. si \perp aparece en T se agrega la cláusula:

$$\perp \leftarrow \Lambda_{\perp}.$$

2. por cada átomo a en T se agrega el par de cláusulas:

$$\Lambda_a \leftarrow a. \quad a \leftarrow \Lambda_a.$$

3. por cada subfórmula $\neg F$ en T se agrega el par de cláusulas:

$$\Lambda_{\neg F} \leftarrow \neg \Lambda_F. \quad \perp \leftarrow \Lambda_{\neg F} \wedge \Lambda_F.$$

4. por cada subfórmula $F \wedge G$ en T se agregan las tres cláusulas:

$$\Lambda_{F \wedge G} \leftarrow \Lambda_F \wedge \Lambda_G. \quad \Lambda_F \leftarrow \Lambda_{F \wedge G}. \quad \Lambda_G \leftarrow \Lambda_{F \wedge G}.$$

5. por cada subfórmula $F \vee G$ en T se agregan las tres cláusulas:

$$\Lambda_{F \vee G} \leftarrow \Lambda_F. \quad \Lambda_{F \vee G} \leftarrow \Lambda_G. \quad \Lambda_F \vee \Lambda_G \leftarrow \Lambda_{F \vee G}.$$

6. por cada subfórmula $F \leftarrow G$ en T se agregan las seis cláusulas:

$$\begin{aligned} \Lambda_{F \leftarrow G} &\leftarrow \Lambda_F. & \Lambda_{F \leftarrow G} &\leftarrow \neg \Lambda_G. & \Lambda_{F \leftarrow G} \vee \Lambda_G \vee \Lambda_{\neg F}. \\ \Lambda_F &\leftarrow \Lambda_{F \leftarrow G} \wedge \Lambda_G. & \Lambda_{\neg F} &\leftarrow \neg \Lambda_F. & \perp &\leftarrow \Lambda_{\neg F} \wedge \Lambda_F. \end{aligned}$$

El teorema siguiente nos muestra como esta traducción de teorías proposicionales a programas generales satisface buenas propiedades semánticas y sintácticas como las hemos definido antes.

Teorema 4.3. *La traducción $\text{PrpGen}: \text{Prp} \rightarrow \text{Gen}$ es PSM en la semántica AS.*

Demostración. Es fácil ver que la traducción es polinomial pues el proceso definido arriba agrega una cantidad constante de fórmulas por cada conectivo en el programa original. Para justificar el hecho de que la traducción es fuertemente fiel observe que, de la Proposición 4.8, podemos obtener $P \xrightarrow{\text{AS}^*} P \cup \Phi_P$ donde $\Phi_P = \{\Lambda_F \leftrightarrow F \mid F \in \mathcal{F}_T\}$. Se puede probar también, por una inducción directa sobre el tamaño de las fórmulas, que $\Phi_P \equiv_{G_3} \Delta_P$. En este punto igual es claro que el programa $P \cup \Phi_P \equiv_{G_3} \{\Lambda_F \mid F \in P\} \cup \Delta_P$. A partir del

⁵Utilizaremos la notación Λ_F en lugar de la notación funcional usual $\Lambda(F)$.

Teorema 4.2 podemos concluir finalmente que $P \xrightarrow{\text{AS}^*} \{\Lambda_F \mid F \in P\} \cup \Delta_P$. El hecho de que la transformación sea modular se sigue de que la transformación se va resolviendo fórmula por fórmula. \square

Observe sin embargo que la traducción AugGen propuesta, igual que la que se encuentra en [34], no es modular reductiva. Esto es porque, aún cuando la traducción mapea programas de una clase complicada a una mucho más simple, los programas en la clase general también se ven modificados por esta traducción.

No debe ser demasiado complicado, sin embargo, construir una traducción PSR para reducir teorías proposicionales a la clase de programas generales aplicando la reducción con más cuidado y sólo para las subfórmulas que, efectivamente, sean las responsables de que la teoría no se encuentre aún en la forma de un programa general. Un ejemplo de como podría realizarse esta traducción esta dado en la Definición 4.7 que, minuciosamente, detecta las implicaciones anidadas que no están permitidas en los programas aumentados y las va eliminando de una en una.

Como para los propósitos formales de nuestra exposición no es necesario exhibir explícitamente dicha traducción PSR, nos contentaremos con saber que es posible construirla y que, seguramente, no debe de ser demasiado complicado construirla a partir de los resultados que ya hemos presentado.

Traducciones como la PrpGen que presentamos son un primer paso hacia la implementación de software que permita calcular answer sets de teorías proposicionales arbitrarias. Si tenemos una teoría T podemos utilizar alguno de las herramientas desarrolladas para programas generales, como DLV o `smodels`, calcular los answer sets de $\text{PrpGen}(T)$ y recuperar entonces, eliminando todos los nuevos átomos introducidos, los modelos originales de la teoría T . Utilizando un esquema similar fue creado `nlp` [34], una herramienta de software que calcula los answer sets de programas en la clase de los aumentados.

4.2.3. Eliminación de Restricciones

El único paso necesario para reducir teorías proposicionales a la clase de programas disyuntivos es el poder eliminar, dentro de los programas generales, las restricciones o *constraints* que aún pueden aparecer. Este resultado es, sin embargo, bien conocido en el contexto de la programación lógica.

Definición 4.9. Dado un programa general P , podemos separar a P en dos conjuntos $P = D \cup C$, donde D es un programa puramente disyuntivo y C el conjunto de restricciones en P . Definimos entonces $\text{GenDis}(P) = D \cup \{p \leftarrow B \wedge \neg p \mid (\perp \leftarrow B) \in C\}$, donde p es un átomo nuevo en $\mathcal{L} \setminus \mathcal{L}_P$.

El lema siguiente es una consecuencia directa del comportamiento de esta transformación, ver por ejemplo [3].

Lema 4.4. [3] *La traducción $\text{GenDis}: \text{Gen} \rightarrow \text{Dis}$ es PSR en la semántica AS.*

Observe que esta transformación simple pudo haber sido también integrada a las fórmulas en Δ_P de la Definición 4.8 para evitar el uso de restricciones en el programa traducido. Esto no se hizo así, sin embargo, para mantener lo más simple posible la demostración del Teorema 4.3.

Como conclusión final de todos los resultados presentados en esta sección tenemos el siguiente teorema, donde PrpDis corresponde como podríamos esperar a la composición de las traducciones PrpGen y GenDis.

Teorema 4.4. *La traducción $\text{PrpDis}: \text{Prp} \rightarrow \text{Dis}$ es PSM en la semántica AS.*

Demostración. Es consecuencia inmediata del Teorema 4.3 y el Lema 4.4. \square

Como hemos comentado antes, no debe ser demasiado complicado construir una traducción similar que sea también modular reductiva. Sin embargo no nos detendremos a dar una demostración formal de este punto.

La existencia de la traducción PrpDis conlleva, dentro de sí misma, una consecuencia importante para la teoría de los answer sets y la programación lógica. Suponga que tenemos un problema y sabemos que se puede resolver calculando los answer sets de una teoría T pero, desafortunadamente, lo único que poseemos es una máquina para calcular min-sets. Lo que podríamos hacer es construir el programa $\text{PrpDis}(T)$, lo cual se puede hacer en tiempo polinomial, y utilizar nuestra máquina de min-sets sobre $\text{PrpDis}(T)$. Como $\text{PrpDis}(T)$ es un programa disyuntivo, y la semántica MS coincide con la de AS precisamente en esta clase, lo que hemos obtenido son los answer sets del mismo $\text{PrpDis}(T)$. Ahora podemos fácilmente, intersectando los modelos obtenidos con el lenguaje de la teoría original \mathcal{L}_T , recuperar los answer sets de T y resolver nuestro problema.

Este procedimiento nos dice que la semántica de min-sets es, en cierto sentido, al menos tan expresiva como la semántica de los answer sets pues, en particular, todos los problemas que se puedan resolver utilizando answer sets serán también solubles utilizando min-sets. Una pregunta interesante es si la semántica de los min-sets se puede reducir también a la de answer sets. Si la respuesta es sí, lo cual es nuestra conjetura, entonces podemos concluir que ambos paradigmas tienen el mismo poder de representar y resolver problemas. Por el contrario, si la respuesta fuera no, entonces la semántica de los min-sets podría iniciar una nueva rama de investigación sobre el tipo de problemas que se pueden resolver mediante técnicas de programación lógica.

4.3. Propiedades de las Traducciones

En la sección anterior presentamos diversas traducciones entre programas lógicos y discutimos su importancia y posibles aplicaciones en la programación lógica. En esta sección estudiaremos con un poco más de detenimiento las propiedades de este tipo de traducciones. En particular veremos como las propiedades *sintácticas* de la traducción (M y R) pueden darnos información sobre sus propiedades *semánticas* (F y S).

Observe, en particular, que las condiciones de que una traducción sea fuertemente fiel y de que sea modular tienen cierta semejanza. Ambas nociones tratan de caracterizar, de maneras distintas, el hecho de que una traducción pueda aplicarse *por partes* a un programa. Una traducción fuertemente fiel puede aplicarse localmente a una *parte* del programa sin modificar su semántica declarativa. Una traducción modular también se puede aplicar a un programa *parte por parte* pero, para que la traducción mantenga propiedades semánticas, es necesario terminar la traducción para cada una de las *partes* que conforman al programa. El siguiente ejemplo hace más clara la diferencia entre las dos nociones.

Ejemplo 4.2. Sea C_1 la clase de programas disyuntivos que tienen exactamente un átomo en la cabeza y uno, o ninguno, átomos en el cuerpo. Sea C_2 la clase de programas disyuntivos que tienen exactamente dos átomos en la cabeza y dos, o ninguno, átomos en el cuerpo. Ejemplos de cláusulas en C_1 son: a , $a \leftarrow b$; y ejemplos de cláusulas en C_2 son $a \vee b$, $a \vee b \leftarrow c \wedge d$. Claramente C_1 y C_2 son clases cerradas bajo uniones.

Dado un programa lógico $P \in C_1$ tomamos, para cada átomo $a \in \mathcal{L}_P$, un nuevo átomo distinto $a' \in \mathcal{L} \setminus \mathcal{L}_P$. Definimos la traducción Hide: $C_1 \rightarrow C_2$, para cada cláusula en P , de la siguiente manera:

$$\text{Hide}(a) = \begin{cases} a' \vee a'. \\ a \vee a \leftarrow a' \wedge a'. \end{cases} \quad \text{Hide}(a \leftarrow b) = \begin{cases} a' \vee a' \leftarrow b' \wedge b'. \\ a \vee a \leftarrow a' \wedge a'. \\ b \vee b \leftarrow b' \wedge b'. \end{cases}$$

Es claro de la definición que la traducción es modular. No es modular reductiva pues, en particular, la condición $C_2 \subseteq C_1$ no se cumple. Observe que la traducción Hide está de hecho “escondiendo” al programa P reescribiéndolo con átomos en un nuevo lenguaje y agrega reglas, equivalentes a $a \leftarrow a'$, para recuperar los answer sets en el lenguaje del programa original P . Así tenemos que la traducción Hide es FM en la semántica AS.

Consideremos el programa, en la clase C_1 , $P = \{a \leftarrow b, b\}$. Ambos programas P y $\text{Hide}(P)$ tienen un sólo answer set que, restringido al lenguaje de \mathcal{L}_P , corresponde a $\{a, b\}$. Observe que, a pesar de que la traducción sea modular, si aplicamos la traducción solo a la primera regla obtendríamos el programa $\text{Hide}(a \leftarrow b) \cup \{b\}$ con un answer set que corresponde a $\{b\}$. Esto ocurre porque la implicación $a \leftarrow b$ ha sido “escondida” y aún cuando tenemos a b como hecho no nos sirve ya para concluir a . Esto muestra que la traducción no es fuertemente fiel, ya que no puede aplicarse localmente a partes aisladas del programa.

Era de esperarse, como las definiciones de modular y fuertemente fiel tienen distinta naturaleza, que fuera posible construir una traducción como Hide que sea FM y no fuertemente fiel. Observaremos que, sin embargo, en muchos casos interesantes de la semántica de answer sets el hecho de que una traducción sea FR sí es suficiente para garantizar que la traducción es fuertemente fiel.

Teorema 4.5. Si $\text{Tr}: C \rightarrow C'$ es una traducción FR en la semántica AS, donde las clases satisfacen $\text{Dis} \subseteq C' \subseteq C \subseteq \text{Prp}$, entonces Tr también es fuertemente fiel.

Demostración. Para demostrar que $P \xrightarrow{\text{AS}^*} \text{Tr}(P)$ tomemos $Q \in \text{Prp}$ que no contenga átomos reservados del conjunto $\mathcal{L}_{\text{Tr}(P)} \setminus \mathcal{L}_P$. Bastaría mostrar que $P \cup Q \xrightarrow{\text{AS}} \text{Tr}(P) \cup Q$. Sea $L = \{a \leftarrow a \mid a \in \mathcal{L}_{\text{Tr}(P)}\}$, de modo que $\mathcal{L}_P \subseteq \mathcal{L}_{\text{Tr}(P)} = \mathcal{L}_L$. Y construimos entonces, usando la traducción del Teorema 4.4, el programa disyuntivo $D = \text{PrpDis}(Q \cup L) \in \text{Dis}$ y, como la traducción es fuertemente fiel, $Q \cup L \xrightarrow{\text{AS}^*} D$.

Observe que, por construcción, ni P ni $\text{Tr}(P)$ tienen átomos del conjunto $\mathcal{L}_D \setminus \mathcal{L}_{(Q \cup L)}$ y, por la definición de fuertemente fiel, obtenemos que $P \cup (Q \cup L) \xrightarrow{\text{AS}} P \cup D$ y, similarmente, $\text{Tr}(P) \cup (Q \cup L) \xrightarrow{\text{AS}} \text{Tr}(P) \cup D$. Luego, a partir de la Proposición 4.4, tenemos también el par de relaciones $P \cup Q \xrightarrow{\text{AS}} P \cup D$ y $\text{Tr}(P) \cup Q \xrightarrow{\text{AS}} \text{Tr}(P) \cup D$.

Ahora, como $D \in \text{Dis} \subseteq C' \subseteq C$, podemos calcular la traducción $\text{Tr}(P \cup D)$ que, por la propiedad de ser fiel, nos lleva a $P \cup D \xrightarrow{\text{AS}} \text{Tr}(P \cup D)$. Pero también, como la traducción es modular reductiva y $D \in C'$, sabemos que $\text{Tr}(P \cup D) = \text{Tr}(P) \cup \text{Tr}(D) = \text{Tr}(P) \cup D$. Así obtenemos por transitividad que $P \cup Q \xrightarrow{\text{AS}} P \cup D \xrightarrow{\text{AS}} \text{Tr}(P) \cup D$. Finalmente llegamos, por la Proposición 4.3 y como $\text{Tr}(P) \cup Q \xrightarrow{\text{AS}} \text{Tr}(P) \cup D$, a que $P \cup Q \xrightarrow{\text{AS}} \text{Tr}(P) \cup D$. \square

Este teorema nos muestra un resultado interesante y es que, si tenemos una traducción fiel en el contexto de la semántica de answer sets, la propiedad sintáctica de ser modular reductiva es suficiente para implicar que la traducción tiene también la propiedad semántica de ser fuertemente fiel.

Este resultado fue dado, en particular, para la semántica de los answer sets pero puede ser generalizado a otras semánticas siempre que cumplan con las siguientes propiedades: (i) si existe un mecanismo para incrementar el lenguaje de un programa, agregando por ejemplo tautologías, sin modificar su significado semántico y (ii) sabemos ya de la existencia de una traducción fuertemente fiel de la clase en que está definida la semántica a una clase más reducida. Entonces todas las traducciones FR situadas entre las clases delimitadas por la traducción conocida son también fuertemente fieles.

Capítulo 5

Aplicaciones y Trabajo Futuro

En este capítulo discutimos diversas aplicaciones de los resultados que hemos presentado a lo largo de este trabajo y planteamos algunas líneas de investigación que podrían seguirse en un futuro.

5.1. Técnicas de Depuración

Como hemos comentado antes, existen diversas implementaciones de herramientas de software para calcular, de manera eficiente, answer sets para programas generales. Una limitante importante, sin embargo, desde el punto de vista de la ingeniería de software, es que no se cuentan aún con ningún tipo de herramientas para analizar o depurar el código de los programas lógicos. Es muy común en la práctica que al estar programando con answer sets que, debido a un error humano al escribir el programa, las herramientas para calcular answer sets no consigan encontrar, contrario a lo que esperábamos, ninguna solución para nuestro problema.

Utilizando la lógica intermedia G_3 , a partir de la caracterización de answer sets que presentamos en el Teorema 3.3, es posible aprovechar la naturaleza tri-valuada de G_3 para detectar, por ejemplo, las restricciones que son violadas y que, posiblemente, estén invalidando los answer sets estamos esperando. Este tipo de herramientas podrían ayudarnos a detectar las reglas que están causando problemas y corregir entonces los errores en nuestro programa.

Formalmente se definieron en [27] las semánticas *weak- G_3* y *strong- G_3* a partir de una cierta noción de minimalidad en un orden parcial entre las interpretaciones de tres valores para los programas lógicos¹. La semántica *strong- G_3* tiene la propiedad de que siempre asigna interpretaciones definidas (no asignan nunca el valor intermedio 1) y, a partir de este hecho, se puede mostrar que coinciden exactamente con la semántica de answer sets.

¹Aquí la noción de semántica difiere un poco de la que estuvimos considerando en este trabajo. Para este caso las semánticas son funciones que asignan, a cada teoría T un conjunto de interpretaciones en G_3 y no, como estuvimos haciendo, subconjuntos del lenguaje \mathcal{L}_T .

```

function GetAnswerSets( $P, M_1, M_2$ ): set of Models;
begin
   $P \leftarrow \text{Reduce}(P, \neg M_1, \neg\neg M_2)$ ;
  if  $\mathcal{L}_P \subseteq M_1 \cup M_2$  then
    return SolvePositive( $P, M_2$ );
  else begin
     $x \leftarrow \text{NextAtom}(P, M_1 \cup M_2)$ ;
    return GetAnswerSets( $P, M_1 \cup \{x\}, M_2$ )  $\cup$ 
      GetAnswerSets( $P, M_1, M_2 \cup \{x\}$ );
  end;
end;

```

Listado 5.1: Algoritmo para calcular Answer Sets

En la semántica *weak-G*₃, por su parte, podemos garantizar que cualquier programa consistente tiene siempre al menos un modelo. Este modelo puede contener además átomos evaluando a 1 que indican, de alguna manera, los átomos que quedaron indefinidos cuando se intentaron calcular los answer sets. La intuición de este proceso es que una herramienta para calcular answer sets no puede decidir, para estos átomos que quedan indefinidos, si son ciertos o falsos y, por lo tanto, está descartando un posible answer set.

En [27] se presentan también ejemplos de como, utilizando una traducción similar a GenDis, pueden eliminarse las restricciones de los programas lógicos e introducir nuevos átomos que, en la semántica *weak-G*₃, sirvan para detectar exactamente la restricción que está ocasionando conflictos.

5.2. Algoritmo para calcular Answer Sets

Se discutieron en la Sección 4.2 diversas ideas de como traducciones como PrpDis pueden utilizarse para enfrentar el problema de calcular answer sets para teorías proposicionales arbitrarias. Mencionamos incluso el caso de `nlp` que utiliza precisamente este enfoque para resolver el problema en el caso de los programas aumentados. En [30] se presenta un enfoque alternativo a partir de una adaptación del método de Davis-Putnam (DP) [8] para el problema de satisfacibilidad (SAT) en lógica proposicional clásica.

Hantao Zhang, y otros, desarrollaron una eficiente implementación del método DP llamada SATO [40]. Investigación reciente [2] muestra que en algunas clases de programas, donde los answer sets coinciden con la semántica *soportada* (supported semantics), una herramienta como SATO puede ser utilizada para calcular answer sets hasta diez veces más rápido, en algunos ejemplos, que utilizando el convencional `smodels`.

El sistema `dlv` para calcular answer sets utiliza, como reportan en [7], una variante del método DP para obtener una serie de *modelos candidatos* y emplean luego otro algoritmo más minucioso que revisa cuales de estos modelos candidatos son efectivamente answer sets.

El Listado 5.1 muestra el algoritmo que proponemos en [30], basado también en DP, que está construido a partir de las siguientes funciones primitivas:

- $\text{Reduce}(P, \neg M_1, \neg\neg M_2)$. Devuelve el programa reducido tal como se presentó en la Definición 3.4 de la Sección 3.1.2.
- $\text{SolvePositive}(P, M)$. Resuelve el caso donde P es un programa positivo y basta determinar si M es un answer set de P . Realmente sólo basta verificar la condición $P \vdash_X M$ en alguna lógica intermedia X . En caso afirmativo la función regresa $\{M\}$ y en caso negativo el conjunto vacío \emptyset .
- $\text{NextAtom}(P, M)$. Regresa un átomo del lenguaje \mathcal{L}_P que no aparezca en M . Como heurística el programa podría seleccionar, por ejemplo, el átomo mas frecuente que ocurra en P .

En [30] se discute también la validez del algoritmo propuesto y se presenta formalmente el siguiente teorema.

Teorema 5.1. *Para una teoría $T \in \text{Prp}$, la función $\text{GetAnswerSets}(P, \emptyset, \emptyset)$ termina y regresa el conjunto de answer sets de T .*

5.3. Inferencia Modal No Monótona

La lógica modal fue desarrollada como consecuencia del estudio de nociones como “necesario” y “posible”. Extendiendo la sintaxis de las fórmulas lógicas con los nuevos conectivos unarios \mathcal{K} y \mathcal{B} que, dándoles un significado semántico adecuado, permiten definir sistemas formales para trabajar con este tipo de nociones. Se han definido una gran cantidad de lógicas modales para definir conceptos como conocimiento, tiempo, obligación y algunos otros de manera similar.

Las fórmulas modales tienen, usualmente, una lectura muy natural y cercana al significado intuitivo que se les quiere dar. Esta es una de las razones por las cuales este tipo de lógicas han servido para dar fundamentos formales a diversas aplicaciones en representación del conocimiento, sistemas multi-agentes, etc.

La definición de answer sets, Definición 3.5, que presentamos en términos de la lógica intuicionista permite, gracias a un mapeo bien conocido de Gödel para la lógica intuicionista dentro de la lógica modal S4, definir una noción de answer sets, o conjuntos de creencias, para teorías proposicionales modales. En [25] estudiamos esta posibilidad y presentamos la siguiente definición donde L_T representa, dada una teoría T , el conjunto de literales $L_T = \mathcal{L}_T \cup \neg\mathcal{L}_T$ que pueden ocurrir en T .

Definición 5.1. Sea T una teoría modal. Para cada conjunto de literales $M \subseteq L_T$ definimos el *conocimiento aceptable* de M como $\text{AK}_M = \mathcal{K}M \cup \mathcal{K}\mathcal{B}\neg(L_T \setminus M)$. Así definimos entonces la semántica de *belief sets* como:

$$\text{BS}(T) = \{M \subseteq L_T \mid T \cup \mathcal{K}\mathcal{B}\text{AK}_M \Vdash_{\text{S4}} \text{AK}_M\}.$$

Esta definición, en el contexto de agentes lógicos, hace aún más clara y natural el fundamento lógico de, en este caso, los belief sets. Observe que ahora, dado un conjunto $M \subseteq L_T$, el conocimiento aceptable de M consiste, precisamente, en aceptar que nuestro agente *sabe* M y, para todo lo que no está contenido en M , *creerá* que no es cierto.

Si nuestro agente puede *suponer* (o saber que cree) este conocimiento aceptable de manera que (i) esto es consistente con la base de conocimientos y (ii) puede estar seguro ahora de este conocimiento aceptable, entonces podemos decir que M corresponde a un buen conjunto de creencias para el agente.

En la discusión de [25] también se discuten ideas de como generalizar este esquema para incluir lógicas multi-modales. Se reflexiona sobre como este tipo de semánticas podrían utilizarse para modelar sistemas lógicos donde varios agentes razonan, con el poder de la inferencia no monotónica, a cerca de los conocimientos y las creencias de cada quien. También se presenta un ejemplo de como este esquema podría utilizarse para modelar el acertijo del “Hombre sabio del Rey” cuya solución se basa, precisamente, en que los agentes sean capaces de razonar acerca de los conocimientos y creencias de sus compañeros.

Capítulo 6

Conclusiones

Este trabajo de investigación busca entender y generalizar diversos conceptos tradicionales en el campo de la programación lógica a través de nociones y relaciones con la lógica matemática. Este enfoque que, se originó de manera reciente en los trabajos de [32, 33], pensamos que ha alcanzado rápidamente un grado de madurez suficiente que nos permite, no sólo dar definiciones y caracterizaciones alternativas para los conceptos que ya se manejaban antes en el contexto de la programación lógica, sino entender con mucha mayor profundidad sus fundamentos lógicos e implicaciones en la práctica.

La caracterización de la semántica de answer sets presentada en el Capítulo 3 ofrece una manera natural de representar un sistema de razonamiento no-monótono (en este caso los answer sets) en términos de las lógicas intermedias propias. Una aplicación interesante, basada en este resultado y que discutimos en la Sección 5.1 a partir de resultados en [27], es un esquema planteado en la lógica G_3 que nos puede ayudar a presentar técnicas de depuración en la programación lógica. Por otra parte, la caracterización que proponemos nos invita a experimentar con otro tipo de lógicas para obtener nuevos y distintos resultados.

Como se muestra en la Sección 5.3, retomando la exposición de [25], se pueden utilizar las lógicas modales para modelar ambientes donde varios agentes pueden razonar acerca de las creencias y los conocimientos de cada uno con el poder de la inferencia no monotónica. Similarmente, en [31] se estudia la posibilidad de utilizar una lógica lineal para modelar sistemas de agentes en ambientes con recursos limitados. También en [32] se presenta un esquema, un tanto diferente, que emplea a las lógicas de Nelson para recuperar la negación explícita, con toda su funcionalidad, en el mismo esquema de answer sets.

El Capítulo 4, por su parte, presenta un esquema completo para estudiar y modelar nociones de equivalencia entre programas lógicos. Estudiamos la equivalencia fuerte, propuesta en [18], y las extensiones conservativas definidas en trabajos como [3, 15, 34]. La caracterización de equivalencia fuerte en términos de la lógica de tres valores G_3 , presentada originalmente en [17] y replanteada ahora en nuestro enfoque para la semántica de answer sets y min-sets, representa otra de las relaciones importantes que permite entender la programación lógica en términos de la lógica matemática.

Analizamos también algunos casos particulares de extensiones conservativas que, a partir

de los resultados en equivalencia fuerte, nos sirven para construir y demostrar propiedades de diversas traducciones entre programas lógicos. Construimos en particular una traducción de teorías proposicionales a la clase de programas aumentados y, siguiendo los resultados de [34], podemos completar la cadena para reducir los programas hasta la clase simple de programas disyuntivos.

La existencia de dicha traducción tiene varias implicaciones importantes. Primero, nos brinda un procedimiento con el cual podemos calcular los answer sets de teorías proposicionales arbitrarias. Podemos ahora traducir cualquier teoría proposicional a la clase de programas disyuntivos donde existen ya eficientes algoritmos y herramientas de software para resolver el problema. En [30], como explicamos en la Sección 5.2 se analizan otras posibilidades para implementar un algoritmo más eficiente para calcular answer sets de teorías proposicionales basados en las reducciones que presentamos en la Sección 3.1.2.

Otra de las implicaciones importantes que se siguen a partir de la construcción de esta traducción es que la semántica de los min-sets es, en cierto sentido, al menos tan expresiva como la semántica de answer sets. Como hemos visto, gracias a esta traducción, es posible reducir (en tiempo polinomial) el problema de calcular los answer sets de una teoría proposicional al problema de calcular los min-sets para un segundo programa. ¿Será posible reducir también el problema de calcular min-sets para teorías proposicionales al problema de calcular answer sets? Ésta es aún una pregunta abierta para investigación en un futuro y, como hemos argumentado, no importa cual sea la respuesta será de gran trascendencia para entender mejor los alcances y la expresibilidad de este tipo de semánticas.

Estudiamos también finalmente algunas propiedades sintácticas y semánticas de las traducciones, así como de las relaciones que pueden existir entre ellas. Demostramos en particular, y en el contexto de answer sets, que una amplia clase de traducciones fieles y modular reductivas son también fuertemente fieles. Y discutimos también cómo la misma prueba podría ser aplicada en otras semánticas a partir de algunas propiedades sencillas.

En este trabajo, como se hace comúnmente al estudiar answer sets, se restringe la atención únicamente a teorías proposicionales. Un problema interesante, y también abierto para un futuro, es tratar de generalizar estos resultados a teorías de primer orden y obtener entonces, quizá, métodos más eficientes para tratar programas con variables sin tener que recurrir a la instanciación completa de ellos.

La aportación principal de este trabajo de investigación es, sin embargo, esclarecer las relaciones que existen entre la programación lógica y la lógica matemática misma. De este modo esperamos obtener una buena cantidad de retroalimentación entre estas dos áreas del conocimiento.

Reconocimientos Numerosas discusiones y comunicaciones por correo electrónico con Michael Gelfond y David Pearce fueron de gran utilidad para clarificar muchas de las ideas y aplicaciones presentadas en este trabajo. Un agradecimiento especial, por supuesto, para Mauricio Osorio por todas sus enseñanzas. Gracias también a mis sinodales Maxim Todorov, asesor, y Andrés Ramos por su apoyo y sus consejos. Esta investigación fue patrocinada por el Consejo Nacional de Ciencia y Tecnología, CONACyT (proyectos 35804-A y 37837-A)

Bibliografía

- [1] José Julio Alferes, Luís Moinz Pereira, Halina Pryzmusinska, and Teodor Prymusinski. LUPS—a language for updating logic programs. *Artificial Intelligence*, 138:87–116, 2002.
- [2] Yuliya Babovich, Esra Erdem, and Vladimir Lifschitz. Fages’ theorem and answer set programming. In Chitta Baral and Mirek Truszczynski, editors, *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Breckenridge, Colorado, USA, April 2000.
- [3] Chitta Baral. *Knowledge Representation, reasoning and declarative problem solving with Answer Sets*. Cambridge University Press, Cambridge, 2003.
- [4] Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domain. In Jack Minker, editor, *Logic Based Artificial Intelligence*, pages 257–279. Kluwer, 2000.
- [5] C. Bell, A. Nerode, R. Ng, and V. S. Subrahmanian. Implementing stable semantics by linear programming. In Luís Moniz Pereira and Anil Nerode, editors, *Proceedings of the Logic Programming and Non-Monotonic Reasoning 1993*, pages 23–42, Lisbon, Portugal, 1993. MIT Press.
- [6] L. E. J. Brouwer. *Brouwer’s Cambridge Lectures on Intuitionism*. Cambridge University Press, Cambridge, 1981.
- [7] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Pruning operators for answer set programming systems. In *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning*, pages 200–209, 2002.
- [8] M. Davis and H. Putnam. A computing procedure for quantification theory. *ACM*, 7:201–215, 1960.
- [9] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181. Springer-Verlag, 1997.
- [10] Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. On updates of logic programs: Semantics and properties. Research Report 1843-00-08, INFSYS,

2000. A shortened version of this paper appears in *Theory and Practice of Logic Programming*, 2002.
- [11] Esra Erdem and Vladimir Lifschitz. Fages' theorem for programs with nested expressions. In *Proceedings of the 17th International Conference on Logic Programming*, pages 242–254, Paphos, Cyprus, December 2001. Springer.
 - [12] Michael Gelfond, M. Balduccini, and J. Galloway. Diagnosing physical systems in a prolog. In W. Faber Thomas Eiter and Mirek Truszczyński, editors, *Proceedings of the 6th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 213–226, Vienna, Austria, 2001.
 - [13] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *5th Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
 - [14] Michael Gelfond, Halina Przymusińska, and Teodor Przymusiński. On the relationship between circumscription and negation as failure. *Artificial Intelligence*, 38:75–94, 1989.
 - [15] Tomi Janhunen. On the effect of default negation on the expressiveness of disjunctive rules. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Logic Programming and Nonmonotonic Reasoning, 6th International Conference*, number 2173 in Lecture Notes in Computer Science, pages 93–106, Vienna, Austria, September 2001. Springer.
 - [16] Paolo Liberatore. Algorithms and experiments on finding minimal models. Technical Report 09-99, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, 1999.
 - [17] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.
 - [18] Vladimir Lifschitz, L. R. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
 - [19] J. Lobo and V. S. Subrahmanian. Relating minimal models and pre-requisite-free normal defaults. *Information Processing Letters*, 44:129–133, 1992.
 - [20] V. W. Marek and J. B. Remmel. On the foundations of answer set programming. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pages 124–131. AAAI Press, Stanford, USA, 2001.
 - [21] Elliott Mendelson. *Introduction to Mathematical Logic*. Wadsworth, Belmont, CA, third edition, 1987.
 - [22] Jack Minker and Donald Perlis. Computing protected circumscription. *The Journal of Logic Programming*, 2:235–249, 1985.

- [23] Juan Antonio Navarro. Answer set programming through G_3 logic. In Malvina Nissim, editor, *Seventh ESSLLI Student Session, European Summer School in Logic, Language and Information*, Trento, Italy, August 2002.
- [24] Juan Antonio Navarro. Properties of translations for logic programs. Accepted to appear at the Proceedings of the ESSLLI'03 Student Session, 2003.
- [25] Mauricio Osorio and Juan Antonio Navarro. Answer set programming and S4. Technical Report in progress, 2003.
- [26] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Equivalence in answer set programming. In A. Pettorossi, editor, *Logic Based Program Synthesis and Transformation. 11th International Workshop, LOPSTR 2001*, number 2372 in LNCS, pages 57–75, Paphos, Cyprus, November 2001. Springer.
- [27] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Debugging in A-Prolog: A logical approach (abstract). In P.J. Stuckey, editor, *Logic Programming. 18th International Conference, ICLP 2002*, number 2401 in LNCS, pages 482–483, Copenhagen, Denmark, August 2002. Springer.
- [28] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. A logical approach for A-Prolog. In Ruy de Queiroz, Luiz Carlos Pereira, and Edward Hermann Haeusler, editors, *9th Workshop on Logic, Language, Information and Computation (WoLLIC)*, volume 67 of *Electronic Notes in Theoretical Computer Science*, pages 265–275, Rio de Janeiro, Brazil, 2002. Elsevier Science Publishers.
- [29] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Applications of intuitionistic logic in answer set programming. Accepted to appear at the TPLP journal, 2003.
- [30] Mauricio Osorio, Juan Carlos Nieves, and Juan Antonio Navarro. Computing preferred safe beliefs. Technical Report in progress, 2003.
- [31] Mauricio Osorio, Jose Juan Palacios, and José Arrazola. Towards a framework for answer set programming as provability in linear logic. In Joaquin Borrego Diaz, editor, *Proceedings of the first IDEIA workshop (in conjunction with IBERAMIA 2002)*, Sevilla, Spain, 2002.
- [32] David Pearce. From here to there: Stable negation in logic programming. In D. M. Gabbay and H. Wansing, editors, *What Is Negation?*, pages 161–181. Kluwer Academic Publishers, Netherlands, 1999.
- [33] David Pearce. Stable inference as intuitionistic validity. *Logic Programming*, 38:79–91, 1999.

- [34] David Pearce, Vladimir Sarsakov, Torsten Schaub, Hans Tompits, and Stefan Woltran. A polynomial translation of logic programs with nested expressions into disjunctive logic programs: Preliminary report. In P. J. Stuckey, editor, *Logic Programming. 18th International Conference, ICLP 2002*, number 2401 in LNCS, pages 405–420, Copenhagen, Denmark, August 2002. Springer.
- [35] Alberto Pettorossi and Maurizio Proietti. Transformation of logic programs. In C. J. Hogger D. M. Gabbay and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter D, pages 697–787. Oxford University Press, 1998.
- [36] Ch. Sakama and K. Inoue. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.
- [37] A. S. Troelstra and Dirk van Dalen. *Constructivism in Mathematics: An introduction*, volume II. North-Holland, Amsterdam, 1988.
- [38] Dirk van Dalen. *Logic and Structure*. Springer, Berlin, second edition, 1980.
- [39] M. Zakharyashev, F. Wolter, and A. Chagrov. Advanced modal logic. In D. M. Gabbay and F. Guentner, editors, *Handbook of Philosophical Logic*, volume 3, pages 83–266. Kluwer Academic Publishers, Dordrecht, second edition, December 2001.
- [40] Hanato Zhang. SATO: A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, 22:1–3, March 1993.