Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación

Tesis presentada como requisito para obtener el título de:

Maestro en Ciencias de la Computación

Tesis:

"PROGRAMACIÓN LÓGICA EN MODELOS PROBABILÍSTICOS"

Presenta:

José Santiago Capilla Ávila

Asesor: Dr. José Luis Carballido Carranza
Co-Asesor: Dra. Claudia Zepeda Cortes
Co-Asesor: Hortensia J. Reyes Cervantes

Febrero 2013

Agradecimientos

A Dios por permitirme llegar a este día.

A mi padre porque sin su apoyo y comprensión no hubiera iniciado este proyecto.

A mi madre por sus consejos y regaños.

A la Dra. Claudia Zepeda por su orientación y apoyo.

Y finalmente a mi esposa por ayudarme a terminar este proyecto.

Dedicatoria

Este trabajo lo dedico a mi esposa e hijo personas que me motivan a seguir luchando cada día.

Índice

Capítulo 1 Introducción 3

Capítulo 2 Marco teórico 5

2.1 Programación lógica 5

2.2 Semánticas de programación lógica 10

2.2.2 Semántica Estable 11

2.2.3 Semántica P-estable 13

3.3 Answer Set Programming 14

2.3.1 Answer Set Solver 15

2.4 Estado del arte 16

2.5 Plog 18

2.5.1 Sintaxis 18

Capitulo 3 Analisis y Diseño 25

3.1 Análisis de Requisitos 25

3.2 Grounder 26

3.2.2 Grounder Gringo 27

3.2.3 Grounder lparser 27

3.2 solver 28

3.2.1 Solver Smodels 28

3.2.2 Solver ClaspD 28

3.3 Liimo 29

Capitulo 4 Implementación 31

4.1 Plog para semántica estable utilizando gringo y smodels 31

4.2 Plog para la semántica P-estable utilizando liimo 33

Capitulo 5 Pruebas 37

5.1 Probabilidad obtenida usando la semántica estable con Plog ……..37

5.2.2 Utilizando gringo y smodels 37

5.2.3 Utilizando gringo y claspD 39

5.2.4 Utilizando lparser y smodels..... 41

5.2.5 Utilizando lparse y claspD 44

5.2 Probabilidad obtenida usando la semántica P-estable con Plog 46

5.2.6 Utilizando liimo 47

Capitulo 6 Concluciones 49

Bibliografía 51

Anexo A 53

Anexo B 58

Capítulo 1

Introducción

En las ciencias de la computación un área de estudio importante es la representación del conocimiento para la Inteligencia artificial, la cual se debe realizar de manera precisa y sin ambigüedades para poder ser utilizados por sistemas automatizados; además se debe contar con reglas precisas para realizar inferencias. Inicialmente se usaba la lógica clásica, pues esta sirve como herramienta para representar el conocimiento, pues cuenta con una semántica bien definida y un mecanismo claro para hacer inferencias. Sin embargo no puedes representar el razonamiento no monótono, el cual puede representar características del mundo real en especial el razonamiento de sentido común.

Una lógica es no monótona si al añadir nuevos axiomas a una teoría basada en ella, dicha teoría puede perder alguno de los teoremas que ya tenía. Y el sentido común es así, pues con información nueva se pueden rechazar conclusiones antes aceptadas.

Una de las semánticas utilizadas más exitosas para modelar razonamiento no monótono es la semántica estable, presentada por Gelfond y Lifschitz en el artículo [6].
Esta semántica está definida a través de una transformación sencilla de la cual se habla en el capítulo 1. En el artículo [11], los autores definen la semántica P-estable para programas disyuntivos a través de una transformación similar a la utilizada por Gelfond y Lifschitz en su definición de la semántica estable.

Los autores del artículo [11] también han presentado resultados

que muestran que la semántica P-estable para programas normales es suficiente para expresar cualquier problema que pueda ser expresado con la semántica estable para programas disyuntivos.

Bajo esta premisa utilizaremos el paradigma de programación que permite transmitir este conocimiento a sistemas automatizados, que es la llamada Programación Lógica. Pues el significado depende de la semántica utilizada.

Por tanto en este documento nos centraremos en la semántica denominada P-estable y estable, pues se extenderá un sistema ya creado denominado p-log, para que pueda aceptar las dos semánticas antes mencionadas, cabe mencionar que este sistema está basado en el enfoque de Razonamiento Probabilístico basado en Answer Sets .

Capítulo 2

Marco Teórico

En este capítulo se presentan las definiciones formales y conceptos necesarios para entender la programación lógica. Asimismo explicar formalmente las semánticas de programación lógica y hacer especial énfasis a las utilizadas en este trabajo. Finalmente se expone el sistema que se desea extender.

2.1 Programación lógica

Un programa normal lógico es denotado como $\mathcal{L}$, es formado por reglas, estas a su vez están formadas por átomos, los cuales están formados por términos, y finalmente pueden ser constantes o variables. El conjunto de constantes y el conjunto de variables son disjuntos.

El lenguaje de lógica proposicional tiene un alfabeto que consiste de:

- Símbolos proposicionales:

  $p_0, p_1,$

- Conectivos: ∧, ∨, ←, ¬, y
- Símbolos auxiliares; (,).

Un átomo a es de la forma $p(t_i, \ldots, t_n)$, entonces si $n = 0$

simplemente se escribe p; donde t_i de 1 $hasta$ n son términos, llamados términos de a, la forma positiva del átomo se le llama literal positiva o la negación del átomo con la forma $\neg a$ es llamada literal negativa. También una variable es un término y una constante es un término.

Una regla normal es de la forma $a_h \leftarrow b_1 \wedge, \ldots, \wedge b_n \wedge \neg b_{n+1} \wedge, \ldots, \wedge \neg b_{n+m}$, donde a_h y cada b_i son átomos para $1 \leq i \leq n+m$.

El concepto de la flecha ← es semejante a la implicación en lógica clásica, solo que se lee en sentido inverso, es decir, derecha a izquierda, y la coma es análoga a la conjunción en lógica clásica (∧).

Una forma de abreviar la notación de una regla normal es la siguiente:

$$P = a_h \leftarrow B^+ \cup \neg B^- ,$$

Donde el conjunto $\{b_1, \ldots, b_n\}$ es denotado por B^+ y el conjunto $\neg\{b_{n+1}, \ldots, b_{n+m}\}$ es denotado por B^-. El átomo a_h se le denomina cabeza de P y se le denota como $H(r)$, el cuerpo de una regla puede ser vacía, en tal caso la regla es conocida como un hecho y puede denotarse simplemente como: $a \leftarrow .$ en caso contrario, es decir, que la cabeza de la regla sea vacía, entonces se conoce como una restricción y se denota por: $\leftarrow b, c.$

Entonces un programa normal lógico es un programa que consiste exclusivamente de reglas normales, es decir, si del lado izquierdo de la fecha hay más de un átomo se diría que es un programa disyuntivo.

Un programa encallado (grounded normal programs) es una clase particular de programa, y lo definimos a continuación.

Definición 1: Se dice que un término esta encallado si es una constante. Un término no está encallado si es una variable. Un átomo está encallado si todos sus términos están encallados. Un programa está encallado si todos los átomos de todas sus reglas están encallados.

Para entender mejor esta definición damos el siguiente programa:

$$P = \{ \quad x \leftarrow e(x,d), a(d). \\ \quad\quad a(d) \leftarrow x. \quad \}$$

Decimos que P está encallado si los términos $\{x, d\}$ son constantes. Si x fuera la única variable con dos posibles valores constantes b y c, entonces el programa encallado de P es el siguiente:

$$P = \{ \quad b \leftarrow e(b,d), a(d). \\ a(d) \leftarrow b. \\ \quad\quad c \leftarrow e(c,d), a(d). \\ \quad a(d) \leftarrow c. \quad \}$$

A lo largo de los siguientes capítulos utilizamos las abreviaciones $\mathcal{L}_p$ y $Prog_{\mathcal{L}}$ que se define a continuación.

Definición [Osorio M. 2010]: La signatura $\mathcal{L}_p$ de un programa encallado P es el conjunto de átomos encallados que ocurren en P. Y dado un conjunto de átomos $\mathcal{L}$, definimos $Prog_{\mathcal{L}}$ como el conjunto de todos los programas normales P tal que $\mathcal{L}_p = \mathcal{L}$.

Para ejemplificar la definición anterior se da el siguiente programa encallado P:

$$P = \{ \quad c \leftarrow b\,(c,d) \wedge a. \\ a(d) \leftarrow c. \\ a \vee c \quad \},$$

Entonces

$$\mathcal{L}_p = \{c, b(c,d), a, a(d)\}$$

Las siguientes definiciones de modelo, modelo mínimo y satisfactibilidad, entre otros, son conceptos elementales de lógica proposicional y van a ser muy utilizados en los capítulos siguientes.

Estas definiciones han sido tomadas del libro [Shawn H. 2006] y adaptadas a la notación que se utiliza en este documento.

Definición 2: Sea $\boldsymbol{M}$ un conjunto de átomos, decimos que $\boldsymbol{M}$ modela (en el sentido de lógica clásica) la regla normal $\boldsymbol{r}$ si y sólo si se cumple alguno de los siguientes puntos:

$$\boldsymbol{B^+(r) \setminus M \neq \emptyset}$$
$$\boldsymbol{B^-(r) \cap M \neq \emptyset}$$
$$\boldsymbol{H(r) \in M \neq \emptyset}$$

También decimos que un conjunto de átomos $\boldsymbol{M}$ modela la disyunción $\boldsymbol{\neg a_1 \vee, \ldots, \vee \neg a_n}$ si y sólo sí $\boldsymbol{\{a_1, \ldots, a_n\} \setminus M \neq \emptyset}$.

Decimos que $\boldsymbol{M}$ modela un programa $\boldsymbol{P}$ si y sólo si $\boldsymbol{M}$ modela todas las reglas de $\boldsymbol{P}$. Al conjunto de átomos $\boldsymbol{M}$ lo llamamos modelo de $\boldsymbol{P}$ y se denota esta relación como $\boldsymbol{M \models P}$.

Un ejemplo de esta definición es: Suponemos $\boldsymbol{P}$ como el siguiente programa encallado y proponemos una $\boldsymbol{M}$ que satisface todas las reglas de $\boldsymbol{P}$, sería el siguiente:

$$\boldsymbol{P = \{c \leftarrow b(c,d), a. \quad a(d) \leftarrow c. \quad a\ V\ c.\}}$$

Entonces

$\boldsymbol{M = \{c\}}$ Es un modelo de P.

Definición 3: Sean $\boldsymbol{P}$ y $\boldsymbol{P'}$ dos programas lógicos, si para todo conjunto de átomos $\boldsymbol{M}$ tal que $\boldsymbol{M \models P}$ también $\boldsymbol{M \models P'}$, decimos que $\boldsymbol{P'}$ es consecuencia lógica de P y lo escribimos como $\boldsymbol{P \models P'}$.

Ejemplificando la definición anterior, suponemos $\boldsymbol{P}$ como el siguiente programa:

$$\boldsymbol{P = \{\ c \leftarrow b(c,d), a.}$$

$a(d) \leftarrow c.\}$

Y P' el siguiente programa

$P' = \{c \leftarrow \ b(c,d), a.\}$

Entonces

$P \models P'$.

Definición 4: Decimos que M es un modelo mínimo de un programa P sólo cuando M es un Modelo de P y ningún otro modelo de P diferente de M es un subconjunto de M. $MM(P)$ es el conjunto de modelos mínimos de P.

Definición 5: Un programa P es una tautología si y sólo si todo conjunto de átomos es un modelo de P. P es una contradicción si y sólo si todo conjunto de átomos no es un modelo de P. P es satisfactible si y sólo si tiene al menos un modelo.

Para entender de mejor manera la definición anterior, se propone el siguiente ejemplo, donde P es el siguiente programa encallado y todo P es el modelo, es decir: $P = \mathcal{L}_p$:

$P = \{c \leftarrow b, c\}$

Entonces P es una tautología y por lo tanto es satisfactible. Ahora si P es el siguiente programa

$P = \{c \leftarrow b, c. \quad \neg c \leftarrow\}$

Por tanto, ***P*** es una contradicción, es decir, no es satisfactible. Cuando un programa ***P*** es insatisfactible lo denotamos como:

$$\boldsymbol{P} \models \bot$$

Definición [Osorio M. 2010] : Para cualquier programa ***P***, la parte positiva de ***P*** denotada por $\boldsymbol{POS(P)}$, es el programa compuesto exclusivamente de aquellas reglas en ***P*** que no tienen literales negativas.

Antes de abordar las definiciones de los modelos de la semántica p-estables y los modelos de la semántica estable, debemos dar la definición de la transformación RED que es necesaria para el cálculo de estos modelos**.**

Definición [Osorio M. 2006]: Sea *P* un programa normal y *M* un conjunto de átomos, entonces definimos la reducción de *P* respecto a *M* como:

$$RED(P,M) = \{H(r) \leftarrow B^{+} \cup \neg(B^{-}(r) \cap M) : (r) \in P\}$$

2.2 Semánticas de programación lógica

Al hablar de semántica entendemos que esta es la forma en que interpretamos un programa, mejor dicho damos sentido a la representación de este, pues indica cómo se debe entender un programa lógico.

Entonces un programa lógico necesita una semántica para esto, se le asigna un conjunto de interpretaciones. Una interpretación es cualquier subconjunto de la base de herbrand aumentada, con la condición de que en él no ocurran literales complementarias, es decir, que no ocurran las literales *a* y *–a* en dicha interpretación, a los que se le conoce como átomos, en la lógica clásica se le denomina proposición y solo puede tomar dos valores, los que aparecen en dicha colección toman el valor de verdad “cierto”

mientras que las que no aparecen toman el valor de verdad “falso”.

Así no todas las interpretaciones de un programa dado son aceptadas por una semántica, las que son aceptadas se les llaman modelos. Cabe mencionar que los modelos de un programa varían de una semántica a otra aunque en algunos casos coinciden.

Ya se mencionó varias veces con anterioridad, pero se debe destacar que un programa es un conjunto de reglas (o formulas) que son implicaciones, en nuestro caso se van a usar principalmente programas normales. Entonces bajo una cierta semántica a un programa $\boldsymbol{P}$ se le asignan una colección de interpretaciones.

A continuación se define cómo se obtienen estos modelos, asimismo se analizan unos ejemplos de programas bajo la semántica estable y la semántica p-estable

2.2.2 Semántica Estable

Si bien hablamos anteriormente de semántica, ahora con lo ya expuesto decimos que la representación obtenida da sentido al programa y esta a su vez se le da el nombre de modelo

Por tanto, el concepto de semántica estable de refiere al significado del modelo estable, o también conocido como answer set, entonces estos modelos determinan que respuesta es considerada correcta para una pregunta dada. A continuación se dan las definiciones formales para la semántica estable.

Definición [Gelfond M. 1988]: Sea $\boldsymbol{P}$ un programa cualquiera, para cualquier conjunto $\boldsymbol{M} \subseteq \mathcal{L}_{\boldsymbol{P}}$, sea ***POS (RED (P, M))*** el programa definitivo obtenido de $\boldsymbol{P}$, en donde mediante la eliminación de cada regla que tenga una literal $\neg \boldsymbol{l}$ en su cuerpo con $\boldsymbol{l} \in \boldsymbol{M}$, posteriormente, la eliminación de todas las literales $\neg \boldsymbol{l}$ en el cuerpo de las reglas restantes. Claramente $\boldsymbol{POS}(\boldsymbol{RED}(\boldsymbol{P},\boldsymbol{M}))$ no contiene $\neg$, entonces $\boldsymbol{M}$ es un modelo estable de $\boldsymbol{P}$ si y solo si $\boldsymbol{M}$ es un modelo clásico mínimo de $\boldsymbol{POS}(\boldsymbol{RED}(\boldsymbol{P},\boldsymbol{M}))$.

Para entender de una mejor manera la definición anterior se expone un ejemplo en el cual se va describiendo su desarrollo para la obtención de los modelos estables.

Ejemplo 1: Proponemos P de la siguiente manera:

$$P = \{\quad b \leftarrow \neg a.$$
$$a \leftarrow \neg b.$$
$$p \leftarrow \neg a.$$
$$p \leftarrow a. \quad \}$$

Sea una interpretación ***M = {a, p}***, entonces tenemos que ***RED (P, M)*** tiene el siguiente resultado:

$$RED(P,M) = \{\quad b \leftarrow \neg a.$$
$$a \leftarrow.$$
$$p \leftarrow \neg a.$$
$$p \leftarrow a. \quad \}$$

Una vez que se obtiene ***RED***, el paso a seguir es eliminar de ***RED (P, M)*** las reglas donde se encuentran las negaciones de las literales, denotado como ***POS (RED (P, M))***, entonces el programa resultante es el siguiente:

$$POS(RED(P,M)) = \{\quad a \leftarrow.$$
$$p \leftarrow a. \quad \}$$

Por lo que el siguiente paso es obtener el modelo mínimo, pero como ***M*** es el modelo resultante de la reducción ***POS (RED (P, M))***, ya es un modelo mínimo, entonces ***M*** es el modelo del programa ***P***. así que por la definición [Gelfond M. 1988], ***M = {a, p}*** es un modelo estable de ***P***.

Es trascendente mencionar que el tamaño de búsqueda para calcular los modelos estables es $2^{\mathcal{L}p}$, así que nosotros solo hicimos el cálculo de uno de los ocho posibles candidatos, se puede probar con todas las interpretación, pero se llegara a la conclusión que no son modelos estables.

2.2.3 Semántica P-estable

En este apartado iniciamos definiendo formalmente la semántica p-estable. Después se da un ejemplo de cómo obtener estos modelos a partir de un programa normal lógico.

Definición [Osorio M. *et al. 2010*] Sea P un programa normal y M un conjunto de átomos, decimos que M es un modelo p-estable de P si:

1. M es un modelo en lógica clásica de P y
2. La conjunción de los átomos en M es una consecuencia lógica (lógica clásica) de RED (P, M) (denotado como RED (P, M) $\models$ M).

Ejemplo 2 Supongamos un programa P de la siguiente manera:

$$P = \{a \leftarrow \neg c, b.$$
$$b \leftarrow c.$$
$$c \leftarrow \neg a.\}$$

Ahora hay que proponer un M = {b, c}, entonces la salida de RED seria la siguiente:

$$\mathrm{RED}(P,M) = \{a \leftarrow \neg c, b.$$
$$b \leftarrow c.$$
$$c \leftarrow .\}$$

Una vez hecha esta transformación, todos los conectivos ***not*** se tomarán como el conectivo tradicional de negación en lógica clásica.

Analizando el ejemplo anterior, vemos que ***c*** es un hecho, así que ***c*** es una consecuencia lógica de RED(P,M), por tanto se denota como ***RED(P,M)*** ╞ ***{c}***, la regla **b←c** nos dice que ***b*** es una consecuencia de ***c*** y por tanto ***RED(P,M)*** ╞ ***{b}***, entonces ***RED(P,M)*** ╞ ***{b,c}***, esto implica que ***M = {b,c}*** es una consecuencia lógica de RED(P,M), además M es un modelo de P, por lo tanto M es un modelo p-estable de P.

El siguiente teorema muestra la relación entre la semántica estable y p-estable para programas normales lógicos.

Teorema [Osorio M. *et al.2010*] Dado ***P*** un programa normal lógico y ***M*** un conjunto de átomos. Si M es un modelo estable de P entonces M es un modelo p-estable de P.

Como se puede observar el estar calculando los modelos estables sería una tarea titánica pues el espacio de búsqueda será polinomial, por tanto, surge un paradigma denominado Answer Set Programming el cual se aborda en la siguiente sección.

2.3 Answer Set Programming

La metodología ASP tiene más de 10 años, asimismo el uso de answer set solver para búsquedas fue identificado como un nuevo paradigma de programación en [Marek & Truszczy´nski 1999], así el término "answer set programming" fue usado por primera vez como el título de una sección de la colección donde aparece este artículo.

Normalmente se encuentra abreviado "ASP" y es una forma de programación declarativa orientada hacia problemas de búsqueda difícil (NP-duros), Se basa en la semántica de modelos estables de programación logia, así que los problemas son reducidos al cálculo de los modelos estables y los answer set solver son programas para la generación de modelos estables.

Cabe mencionar que los algoritmos de búsqueda usados en el diseño de muchos solver son mejoras del procedimiento DPLL (por sus siglas en ingles de Davis-Putnam-Logemann-Loveland), es decir, son procesos de búsqueda sistemática para encontrar una asignación satisfactoria para una formula booleana o de lo contrario demostrar que es insatisfactible [Gomes *et al.* 2008], además son similares a los algoritmos usados en solucionadores eficientes SAT.

ASP encapsula todas las aplicaciones de answer sets para la representación del conocimiento [Baral, C. & Gelfond, M. 2004], por tanto, significa que también incluye a solver que usan la representación del conocimiento en programas lógico, como es el caso de p-estable, o mm^r.

2.3.1 Answer Set Solver

En la actualidad existen un gran número de solver para computar los modelos de un programa lógico para cierta semántica, como en el caso de estable existen los siguientes: smodels, clasp, especialmente para programas normales sin disyunción, y para programas lógicos con disyunción están disponibles los siguientes: cmodels, claspD, dlv (este tiene una diferencia importante que más adelante se menciona) , noMoRe++, y pdmodels.

Se puede decir que estos solver son back-end dado que necesitan de un programa front-end, a excepción de dlv que no utiliza uno.

Entonces el front-end utilizado para smodels, cmodels, noMoRe++ y pdmodels es lparse el cual se encarga de traducir el archivo de entrada a un archivo libre de variables principalmente [Baral C. *et al*. 2007, Simons P. 2002, 37; Lin F. 2004], es decir, un programa lógico simple. También puede realizar otras funciones no mencionadas en este documento, pero no quiere decir que sea el único pues existe también gringo que es utilizado por clasp, claspD y coala no mencionado en este documento. Cabe mencionar que gringo y lparse se les denomina grounder, además de utilizar AnsProlog contracción de Answer Set Programming in Logic, se destaca que AnsProlog no solo define una sintaxis sino también una semántica, no en el sentido de los solver antes mencionados sino en el sentido de un compilador.

Para la semántica p-estable se implementó un solver denominado psolver, es un back-end el cual utiliza lparse. Este solver es utilizado para modelos con negación con falla, negación clásica, entre otros.

Entonces en este trabajo solo se utiliza los solver: claspD, smodel,

psolver, pues se tiene especial interés en las semánticas antes mencionadas. Se agrega un cuadro de comparación para los solver antes mencionado, tomando características como: tipo de licencia, que grounder utiliza o sino utiliza, implica si soporta variables, soporta disyunción, sistema operativo, mecanismo.

Funcionamiento	Incremental, inspirado en SAT	Propios algoritmos	Propios algoritmos
Disyunción	Sí, pero la semán	No	no
Conjuntos	No	No	No
Símbolos Funciona les	Si	No	No
Variables	Si, pero usando	Usando Lparse	Usando Lparse
Licencia	GPL	GPL	GPL
S.O.	Linux, Mac Os,	Linux, Mac Os,	Linux, Mac Os,
Solve	Clasp	Smodels	P-stable

Figura 2.1 Comparación de solvers utilizados en este trabajo.

En el anexo A se describe detalladamente las opciones de ejecución y el formato de ejecución de cada solver antes mencionado.

2.4 Estado del arte

Para poder entender el motivo de este trabajo se debe contar con el estudio de lo más reciente que se está haciendo en el área de razonamiento probabilístico, por tanto, se aborda el trabajo que se hizo por [Olvera A. 2011] el cual hace un estudio sobre una implementación, además de entender el sistema que cuenta con dos versiones y asimismo un documento que se realizó en el 2008 por Gelfond, que es lo más reciente que se ha hecho en el área.

En principio el trabajo de Olvera se toma como un manual de la implementación plog [Weijun Z. 2008] además de ejecutar varios ejemplos en el mismo, deja una interrogativa la cual es ¿se puede

usar para otras semánticas no solo para estables?, esta pregunta propone la idea de cambiar los módulos en donde se cargan los modelos.

Entonces se toma esa interrogativa lanzada al aire para llevarla a la realidad en este trabajo de tesis. Antes de empezar con las implementaciones, primero se entabla el estudio del cimiento de este trabajo el cual es un documento elaborado por [Gelfond M. *et al.* 2008], en donde propone un lenguaje declarativo que permite representar la información cuantitativa y cualitativa, cabe mencionar que la información cuantitativa se expresa como probabilidad, en este documento la base probabilística se fundamenta en el mapeo de redes bayesianas, pues se pueden obtener fácilmente de la coherencias de plog.

Y como todo lenguaje debe contar con una semántica la cual se define en dos partes: la primer parte define lo obtención de los mundos posibles y en la segunda parte se asignan las probabilidades a esos mundos, esto quiere decir, que consiste en una parte lógica y una parte probabilística. En este mismo documento se presentan varios ejemplos para ser ejecutados en la implementación, entonces estos mismos ejemplo se podrán usar para el desarrollo del nuevo sistema, al mismo tiempo se tendrá que hacer ejemplos con modelos p-estable.

Finalmente se introducirá la implementación para el lenguaje plog nombrada de la misma forma, como se mencionó al inicio hay dos versiones de esta implementación, bueno en realidad hay 7 implementaciones pero como es algo nuevo está en constante cambio, primero por los errores que se van encontrando al ir probando la implementación y segundo es por los conceptos que se van agregando para la funcionalidad. Esta implementación esta en C++ con código abierto, aunque cuenta con unos ejecutables para Windows, en donde utiliza unos programas lparser, smodels y una librería dinámica para hacer la concatenación de estos programas y el programa principal plog los dos primeros ejecutables sirven para separar la construcción lógica y obtener los modelos estables esta implementación tuvo éxito en una aplicación industrial para el diagnóstico de fallas en el sistema de control de reactivos (RCS) el cual es un sistema del transbordador que tiene la responsabilidad primaria para la maniobra de la aeronave mientras se encuentra en el espacio.[Balduccini M. et al. 2001], además de poder realizar actualización mediante nuevo conocimiento, por medio de agregación de nuevas reglas, la nueva versión denominada plog2.0 tiene la diferencia de poder tratar con las actualizaciones que se

consideran actualmente como inconsistencia.

2.5 Plog

En esta sección se habla del sistema Plog, que esta descrito en [Baral C. 2008]. Plog tiene semántica y sintaxis, quiere decir que es todo un lenguaje de programación, para efectuar su procesamiento combina una parte lógica y otra parte probabilística, además de ser un lenguaje declarativo, se habla más adelante sobre estas afirmaciones, se utiliza AnsProlog en gran medida para la sintaxis de los programas, y la semántica es arbitraria pues se utiliza de manera independiente. Se puede decir que Plog extiende la sintaxis de AnsProlog añadiendo el bloque probabilístico que proporciona una medida de creencia. A continuación se define la sintaxis de Plog.

2.5.1 Sintaxis

Se entiende que la entrada es un programa lógico incrementando con probabilidades, por tanto el programa contiene las siguientes secciones:

(i) Signatura de tipos.
(ii) Declaración.
(iii) Parte regular.
(iv) Reglas de selección random.
(v) Información probabilística.
(vi) Observaciones y de acciones.

Cada sentencia de P-log debe de ser terminada por un punto. Enseguida se explica cada parte antes menciona.

Signatura de tipos:

La signatura ordenada $\boldsymbol{\Sigma}$ de $\boldsymbol{\Pi}$ contiene un conjunto $\boldsymbol{O}$ de objetos y un conjunto $\boldsymbol{F}$ de símbolos funcionales. El conjunto $\boldsymbol{F}$ es la unión de dos conjuntos disjuntos, $\boldsymbol{F_r}$ y $\boldsymbol{F_a}$. Los elementos de $\boldsymbol{F_r}$ se llaman funciones de construcción, las

cuales pueden ser funciones predefinidas por ejemplo: “mod”. Los elementos de F_a son llamados atributos. Los términos se forman con elementos del conjunto O y F_r.

Los atributos tienen la forma $a(t')$, donde a es un atributo y t' es un vector de términos de los tipos que requiere a. Los atributos que tengan el rango {verdadero, falso} son atributos booleanos o relaciones.

Al igual que en la sintaxis de AnsProlog, una sentencia atómica p o $\neg p$ es una literal, las literales p y $\neg p$ son llamadas contrarias, asimismo las expresiones l y $not\ l$, donde l es una literal y not es una negación como fallo de l son llamadas literales extendidas.
Las literales de la forma $a(t) = verdadero$, $a(t) = falso$, y $\neg(a(t) = t_0)$, análogamente son escritas como: $a(t)$, $\neg a(t)$, y $a(t) \neq t_0$ respectivamente.

Si p es una relación unaria y X es una variable entonces una expresión de la forma $\{X: p(X)\}$ será llamada un término conjunto, las ocurrencias de X en esa expresión son llamadas atadas.

Declaración:

La declaración de un programa P-log es una colección de definiciones de tipos y declaraciones de tipos para los atributos.
Un tipo c puede ser definido listando sus elementos explícitamente, $c = \{x_1, \ldots, x_n\}$. o por un programa lógico T con un único modelo A. En este último caso $x \in c$ si y sólo si $c(x) \in A$.

El dominio y rango de un atributo a está dado por una declaración de la forma:

$$a: c_1 x \ \ldots \ x\, c_n \rightarrow c_0$$

donde el dominio se indica del lado izquierdo de la flecha y el rango del lado derecho de ésta.

Para atributos sin parámetros simplemente se escribe

***a*: *c*0.**

Así que para entender de mejor manera lo antes escrito, es con un ejemplo práctico, es cual es parte de un programa de entrada para Plog.

Se considera una situación en la que hay dos dados, cada dado tiene un dueño respectivamente se necesita un atributo que de la relación de pertenencia de dueño, solamente se lanza una vez cada uno los cuales toman valores de 1 al 6, además de esto se necesita un atributo que mapea los valores numéricos a los dados, también una relación que exprese par o impar, para el valor de cada dado, entonces se necesitan funciones aritméticas como la suma y el mod para definir la relación de paridad. A continuación se expone un fragmento del programa, en donde se exponen las necesidades antes mencionadas.

***dice* = {d_1, d_2}.**
***score* = {1, 2, 3, 4, 5, 6}.**
***person* = {mike, john}.**

Declaraciones de tipos.
$c = \{x_1, \ldots, x_n\}$

***roll*: *dice* → *score*.**
***owner*: *dice* → *person*.**
***even*: *dice* → *Boolean*.**

Declaraciones de tipos para atributos.
$a: c_1 \times \ldots \times c_n \rightarrow c_0$

Parte regular:

La parte regular de un programa P-log consiste en una colección de reglas del lenguaje *AnsProlog* sin disyunciones formadas usando las literales de ∑. Por ejemplo, asignar explícitamente de quien es el dado, e igualmente que dado es par, pero también un dado puede ser par y no al mismo tiempo impar. Así es como serían las sentencias en el programa Plog:

***owner* (d_1) = mike.**
***owner* (d_2) = john.**
***even* (D) ← *roll* (D) = S, S mod 2 = 0.**
¬*even* ← *not even* (D).

Las variables *D* el valor de dado y *S* toma el valor de *score* ya definidas en la sección de declaración respectivamente.

Reglas de selección *random*:

Esta sección del programa contiene las reglas que describen los posibles valores de los atributos *random*. Una selección *random* es una regla de la forma

a) $[r]\ random(\ a(t') :\ \{X: p(X)\})\ \leftarrow B.$

Donde ***r*** es un término usado para nombrar la regla y *B* es una colección de literales extendidas de $\sum$. El nombre ***[r]*** es opcional y se puede omitir si el programa contiene exactamente una selección *random* para ***a(t')***. Algunas veces se refiere a ***r*** como un experimento, se puede hacer la observación de que intuitivamente la regla a) dice que si *B* se mantiene, el valor de ***a(t')*** es seleccionado aleatoriamente del conjunto ***{ X : p(X)}*** ∩ ***range(a)}*** por el experimento ***r***, al menos que este valor este fijado por una acción deliberada. Si B en a) es vacío entonces se escribe:

b) $[r]\ random\ (a(t') :\ \{X: p(X)\}).$

Si ***{X: p(X)}*** es igual a *range(a)* entonces la regla a) se escribe:

c) $[r]\ random\ (a(t'))\ \leftarrow B$

Algunas veces se refiere al atributo $\boldsymbol{a(t')}$ como *random* y a $\{\boldsymbol{X}:\ \boldsymbol{p(X)}\}\ \cap\ \boldsymbol{range(a)}\}$ como el rango dinámico de ***a(t')*** vía la regla ***r***.

Para indicar que los valores del atributo *roll: dice* → *score* son *random* se tiene que agregar la declaración:

[r(D)] random(roll(D)) { *Tiene la forma:* ***[r] random(a(t'))*** porque {X: p(X)} es igual al rango de *a* y *B* es vacío.

Información probabilística:

La información sobre las probabilidades de que los atributos *random* tomen un valor en particular está dada por los átomos de probabilidad (pr-atoms) los cuales tienen la forma:

d) $pr_r(a(t') = y \mid_c B) = v$

donde $v \in [0, 1]$, ***B*** es una colección de literales extendidas, ***pr*** es un símbolo especial que no pertenece a $\sum$, ***r*** es el nombre de las reglas de selección *random* para ***a(t')***, y $pr_r(a(t') = y \mid_c B) = v$ dice que ***si el valor de a(t') está dado por un experimento r, y B se mantiene, entonces la probabilidad de que r cause a(t') = y es v***. Más adelante, cuando se hable de la semántica, se verá que si ***W*** es un mundo posible del programa, el cual también se definirá más adelante, que contiene d) y ***W*** satisface ***B*** y el cuerpo de la regla ***r***, entonces se referirá a ***v*** como la probabilidad causal del átomo ***a(t') = y*** en ***W***. Si ***B*** es vacío simplemente se escribe:

e) $pr_r(a(t') = y) = v$

Si el programa contiene exactamente una regla que genera los valores de ***a(t') = y*** el índice ***r*** se puede omitir.

Entonces dando el ejemplo de los dados, se agrega la información de probabilidad en donde se expone un dado bueno (honesto, que no esté cargado) y otro esté cargado, así que el dado que tiene *John* es honesto y el dado de *Mike* está inclinado a obtener el valor 6 con una probabilidad de 0.25.

p_r ***(roll (D) = S*** $\mid_c$ ***owner(D) = john) = 1/6.***
p_r ***(roll (D) = 6*** $\mid_c$ ***owner(D) = mike) = 1/4.***
p_r ***(roll (D) = S*** $\mid_c$ ***Y*** $\neq$ ***6, owner(D) = mike) = 3/20.***

Observaciones y acciones:

Las observaciones y acciones son de la forma:

$$obs(l).$$
$$do(a(t') = y).$$

donde ***l*** es una literal. Las observaciones son usadas para llevar registro de los resultados de los eventos ***random***. Por ejemplo, el programa puede contener ***{obs(roll(d_1) = 4)}*** registrando el resultado de lanzar el dado ***d_1***. Por otro lado, la declaración ***do(a(t') = y)*** indica que *a(t') = y* se hace verdadero como resultado de una acción deliberada. Por ejemplo, ***{do(roll(d1)= 4)}*** puede indicar que ***d_1*** simplemente se puso en la mesa en la posición descrita. Es importante mencionar que en el ejemplo de dado que se está utilizando no se agregarán ni observaciones ni acciones.

El programa completo quedaría de la siguiente manera. Se muestra dividido en bloques para su mejor comprensión.

dice = {d_1, d_2}.	% Hay dos dados: d_1 y d_2	Declaración
score = {1, 2, 3, 4, 5, 6}.	% Cada dado puede caer en cualquiera de los 6 valores	
person = {mike, john }.	% Hay dos personas: *mike* y *john*	
roll: dice → score.	% roll es un atributo que indica en que número cayó el dado	

Parte regular

: dice → person.	er es un atributo que indica qué es dueño de qué dado
even: dice → boolean.	% even es un atributo que indica si el valor en que cayó el dado es par

Información Probabilística

owner (d_1) = mike.	% mike es dueño de d1
owner (d_2) = john.	% john es dueño de d2
even (D) ← roll (D) = S, S mod 2 = 0.	% Notar que D es mayúscula, por lo que indica que es una variable, esta regla indica si un dado cayó en un número par**.**
¬even ← not even (D).	%Si no hay evidencia de que es par entonces **no** es par
[r (D)] random (roll (D)).	% El valor del atributo roll será aleatorio.
p_r (roll (D) = S $|_c$ owner(D) = john) = 1/6.	% La probabilidad de que el dado caiga en un número "Y" causado por ser propiedad de *john* es de 1/6
p_r (roll (D) = 6 $|_c$ owner(D) = mike) = 1/4.	% La probabilidad de que un dado caiga en el número 6 causado por ser propiedad de *mike* es de ¼.
p_r (roll (D) = S $|_c$ S ≠ 6, owner(D) = mike) = 3/20.	% La probabilidad de que un dado caiga diferente de 6, causado por ser propiedad de *mike* es de 3/20.

Capitulo 3

Análisis y Diseño

Como se mencionó anteriormente el objetivo de este trabajo es analizar y realizar una propuesta de fusión dentro de Plog para los ASP expuestos en el marco teórico, esto permitirá poder realizar pruebas para las distintas semánticas dentro de un mismo entorno que puede calcular las probabilidades de los mundos posibles.

En este capítulo describiremos el proceso de análisis realizado al software que llamaremos **plogExt1.0**, la información que obtuvimos de ello y las propuestas de fusión que sea considerado.

3.1 Análisis de Requisitos

Como se expone en el inicio del documento se debe crear un sistema que utilice las implementaciones de distintas semánticas, con el propósito de extender la implementación Plog. Entonces, dado que ya se analizó Plog como lenguaje de programación lógica. Ahora se analiza como extender Plog como implementación.

Para realizar el análisis, primero detallaremos la secuencia que se realiza para ejecutar un programa:

- Primero se tiene que escribir un programa que sea aceptado por Plog.
- Segundo se debe abrir una terminal (ya sea Linux o windows).
- Tercero ya estando en la consola se debe escribir la ruta

para acceder a la carpeta en donde se encuentra el archivo a ser ejecutado y

- Finalmente escribir la línea de comando necesaria para realizar la acción deseada.

Para realizar el proceso anterior se da por hecho que se realizó previamente la instalación de todo lo necesario para llevar a buen término la ejecución, como lo es la instalación del grounder, el solver y el mismo Plog.

Ahora que se describió a groso modo el proceso y dado que se analizó en la sección anterior la sintaxis y la semántica utilizada, se puede dar una propuesta para extender Plog.

Para alcanzar el objetivo de extender las semánticas que acepta Plog, además de los grounder, entonces la propuesta es la siguiente:

- La semántica que se agrega es P-estable por su aporte y para tener continuidad a la dirección de investigación del grupo, por tanto se entiende que se necesita un solver para la semántica.
- Agregar otro solver para la semántica estable, esto aporta versatilidad y poder hacer comparación, en este caso se propone ClaspD.
- Agregar otro grounder, se propone gringo por su compatibilidad con lparse, además de ser nativo de ClaspD.
- Cada solver y cada grounder tienen sus parámetros de ayuda, esto significa que se debe agregar a la interfaz gráfica un módulo para poder realizar la configuración necesaria para cada uno.
- Para los usuarios que les guste escribir las órdenes por línea de comando, se agrega una modulo para este fin.

3.2 Grounder

Grounder es el que interpreta un archivo nativo .plg y lo transforma en un programa normal lógico para que después el solver obtenga los modelos estables posibles.

Es importante mencionar que el grounder y el solver a utilizar se establece desde la línea de comandos así como los archivos .plg, .txt que será donde se obtendrá el archivo nativo y donde

almacenara la traducción de grounder así como los modelos estables resultantes obteniendo de este la probabilidad respectivamente.

3.2.2 Grounder Gringo

El grounder gringo es capaz de traducir los programas nativos .plg provenientes a programas equivalentes que el solver pueda interpretar. La salida del gringo puede ser conducido por el solver, es por ello que al momento de introducir las instrucciones en la línea de comandos es necesario establecer el solver a utilizar.

El solver calcula los conjuntos de respuesta y los encontramos en archivos .txt establecidos desde la línea de comandos en este caso es out.txt que contiene los arrojado por el grounder en este caso gringo y el archivo result.txt que contiene lo arrojado por el solver.

A continuación se presenta la serie de comandos necesaria para la línea de comandos utilizando el grounder gringo y el solver smoldes.

```
./plog -t paradojaSimpson.plg "gringo  out.txt|smodels  0 > result.txt"
```

3.2.3 Grounder lparser

El grounder lpaser al igual que gringo es capaz de traducir lo programas .plg provenientes a programas que el solver pueda interpretar y obtener de este los modelos estables, la diferencia se encuentra en que gringo es más actual que lparse.

Lparser también debe de ser establecido desde la línea de comandos .A continuación se presenta la serie de comandos pero en este caso utilizando lpaser.

```
./plog -t paradojaSimpson.plg "lparse --true-negation -W none out.txt|smodels 0 > result.txt"
```

3.2 solver

La funcionalidad de los solver se basa en un conjunto de respuestas de un modelo estable o en un conjunto de respuestas a partir de un programa lógico. En muchos casos los modelos estables de un programa son idénticos a los modelos de su finalización. Cuando un programa tiene varios modelos estables, el solver es capaz de generar todos ellos.

3.2.1 Solver Smodels

El solver smodels es una implementación de la semántica estable para la obtención de programas lógicos, este a diferencia de claspd no acepta disyunciones.

Al igual que el grounder el solver debe de ser establecido desde la línea de comandos.

A continuación se presenta las serie de comandos necesario para utilizar el grounder lparser con el solver smodels.

```
./plog -t paradojaSimpson.plg "lparse --true-negation -W
none out.txt|smodels 0 > result.txt"
```

Cabe mencionar que el solver smodels también puede ser utilizado con el grounder gringo como se muestra a continuación.

```
./plog -t paradojaSimpson.plg "gringo  out.txt|smodels  0 >
result.txt"
```

3.2.2 Solver ClaspD

El solver claspD al igual que smodels es se utiliza para la semántica estable para la obtención de programas normales lógicos la diferencia entre claspD y smodels está en que claspD es nativo de gringo y acepta disyunciones mientras que smodels no

Al igual que el grounder el solver debe de ser establecido desde la línea de comandos.

A continuación se presenta las serie de comandos necesario para utilizar el grounder lparser con el solver claspD.

```
./plog -t paradojaSimpson.plg "lparse --true-negation -W none out.txt|claspD -n 0 > result.txt"
```

Cabe mencionar que el solver claspD también puede ser utilizado con el grounder gringo como se muestra a continuación.

```
./plog -t paradojaSimpson.plg "gringo  out.txt|claspD -n 0 > result.txt"
```

3.3 Liimo

Liimo es la herramienta encargada de aplicar las nuevas transformaciones, preparar el programa lógico para ser procesado por exx y presentar los resultados de manera adecuada.

Permite elegir cual es la semántica bajo la que se desea procesar el programa lógico, se dispone de tres opciones:

-m Procesa el programa bajo la semántica MMr.

-p Selecciona la semántica p-estable.

-pr Usa también la semántica p-estable, pero agrega estratificado en los algoritmos de exx.

Ninguno de estos argumentos puede ser combinado entre sí. En liimo es necesario proporcionar siempre uno de estos argumentos o el cálculo no podrá realizarse.

Liimo permite trabajar con dos grounders gringo y lparse.

-gl Utiliza lparse.

-gg Utiliza gringo.

Este argumento no es obligatorio proporcionarlo, si no aparece ninguna de las opciones liimo utilizará lparse como grounder por defecto.

Descrito lo anterior se presentan la serie de instrucciones necesarias para utilizar liimo.

```
./plog -t paradojaSimpson.plg "cat out.txt | liimo -gl -pr > result.txt"
```

Capitulo 4

Implementación

Este capítulo describe de manera general el funcionamiento del software plogExt1.0, esto incluye la descripción de las clases y los métodos principales de ésta herramienta.

4.1 Plog para semántica estable utilizando gringo y smodels

Este comienza desde la línea de comandos, se tiene que cumplir con ciertos comandos siguiendo la secuencia necesaria para poder ejecutar el software **plogExt1.0.**

A continuación se muestra un ejemplo de los comandos necesarios.

```
./plog -t paradojaSimpson.plg "gringo
out.txt|smodels  0 > result.txt"
```

`./plog -t` :nos permite elegir el solver que nosotros queremos que se utilice en este caso contamos con dos opciones gringo y lparser , si no se coloca esta instrucción Plog ocupara el que tiene para calcular las probabilidades.

`paradojaSimpson.plg`: `es` el archivo fuente que está en Plog y de donde el grounder tomara el ejemplo del cual se quiere obtener la probabilidad.

`out.txt`: en este archivo se almacena la salida de la transformación hecha por Plog y que sirve como entrada para el grounder en este ejemplo es gringo que este a su vez crea otro

archivo que entiende el solver smodel o claspd o psolver.

`result.txt`: este archivo es donde se almacenaran los modelos de la semántica estable obtenida por el solver, en este ejemplo es el smodels.

La siguiente figura muestra la ejecución de los comandos presentados anteriormente así como la probabilidad resultante.

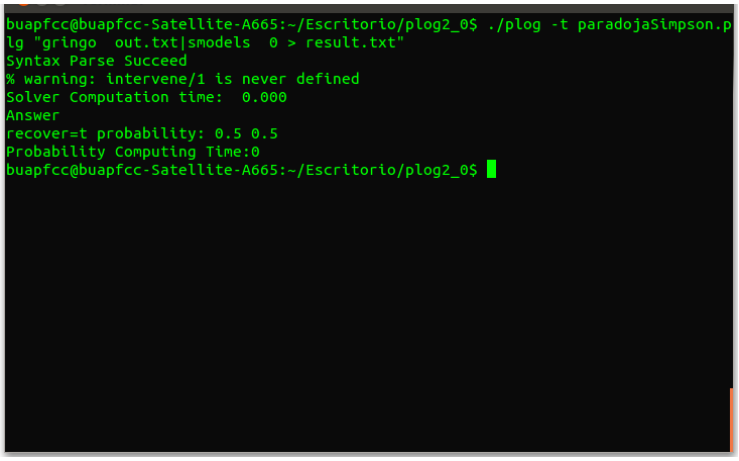

Figura 4.1: Muestra la probabilidad resultante utilizando gringo y smodels

A continuación se muestra el contenido de los archivos out.txt y result.txt.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
1{male(X_):boolean(X_)}1:-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
1{recover(X_):boolean(X_)}1:-not intervene(recover).
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
1{drug(X_):boolean(X_)}1:-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
```

```
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo result.txt

```
smodels version 2.34. Reading...done
Answer: 1
Stable Model: male(t) recover(t) drug(t)
Answer: 2
Stable Model: male(t) recover(f) drug(t)
Answer: 3
Stable Model: male(f) recover(f) drug(t)
Answer: 4
Stable Model: male(f) recover(t) drug(t)
False
Duration: 0.000
Number of choice points: 3
Number of wrong choices: 3
Number of atoms: 31
Number of rules: 34
Number of picked atoms: 11
Number of forced atoms: 0
Number of truth assignments: 66
Size of searchspace (removed): 4 (0)
```

4.2 Plog para la semántica P-estable utilizando liimo

Recordemos que liimo me permite elegir cual es la semántica bajo la que se desea procesar el programa lógico (véase capítulo 3, sección 3.3) y además cuenta con cat que son tuberías que trasmiten la información hacia liimo.

A continuación se nuestra la instrucción para liimo.

./plog -t paradojaSimpson.plg "cat out.txt | liimo -gl -pr > result.txt"

`./plog -t` :nos permite elegir el solver que nosotros queremos que se utilice en este caso liimo contiene a gringo y lparser.

`paradojaSimpson.plg`: es el archivo fuente que está en Plog y de donde el grounder tomara el ejemplo del cual se quiere obtener la probabilidad.

`out.txt:` en este archivo se almacena la salida de la transformación hecha por Plog y que sirve como entrada para el grounder en este ejemplo es gringo que este a su vez crea otro archivo que entiende el solver .

`result.txt:` este archivo es donde se almacenaran los modelos de la semántica estable obtenida por el solver, en este ejemplo es el smodels.

`Cat:` trasmite la información por medio de tuberías.

En la siguiente figura se muestra la línea de comandos vista anteriormente.

Figura 4.2: Muestra la probabilidad resultante utilizando liimo

A continuación se muestra el contenido de los archivos out.txt y result.txt.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
male( t) | male( f):-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
recover( t) | recover( f):-not intervene(recover).
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
drug( t) | drug( f):-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo result.txt:

```
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(1,5))
male(f) pa(r3,drug(t),di_(1,4)) recover(t) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(1,5))
male(f) pa(r3,drug(t),di_(1,4)) recover(f) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(3,5))
male(t) pa(r3,drug(t),di_(3,4)) recover(t) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(3,5))
male(t) pa(r3,drug(t),di_(3,4)) recover(f) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
```

Capitulo 5

Pruebas

En esta sección se presentas los resultados obtenidos al utilizar el sistema Plog extendido con la semántica Pestable, además de otro solver ClaspD y grounder que es Gringo, como se mencionó anteriormente dentro de cada fase del sistema el resultado de cada uno de ellos es un archivos de texto plano (.txt).

5.1 Probabilidad obtenida usando la semántica estable con Plog

Es importante recordar que Plog utiliza a otros sistemas para llegar a su meta, que en este caso es obtener la probabilidad de nuestros mundos posibles utilizando los grounder gringo y lparser así como los solver smodels y claspD en diferentes combinaciones.

5.2.2 Utilizando gringo y smodels

Para obtener la probabilidad de una semántica estables es necesario contar con un archivo base en este caso utilizamos el ejemplo de la paradojaSimposon.plg (véase en Anexo B)que es convertido a un programa normal lógico por Plog y es almacenado en out.txt este a su vez sirve como archivo de entrada a el grounder gringo (recordemos que este forma parte de Plog, véase capítulo 3 Análisis y Diseño) que convierte el archivo en instrucciones legibles que son transmitidas mediante tuberías a el solver smodels dando este como resultado un archivo más llamado result.txt que almacena los mundos posibles para la semántica estable.

A continuación se muestra el comando que se debe de ejecutar mediante el uso de la consola.

```
./plog -t paradojaSimpson.plg "gringo  out.txt|smodels  0 > result.txt"
```

La siguiente figura muestra claramente los comandos mencionados anteriormente y la probabilidad resultante.

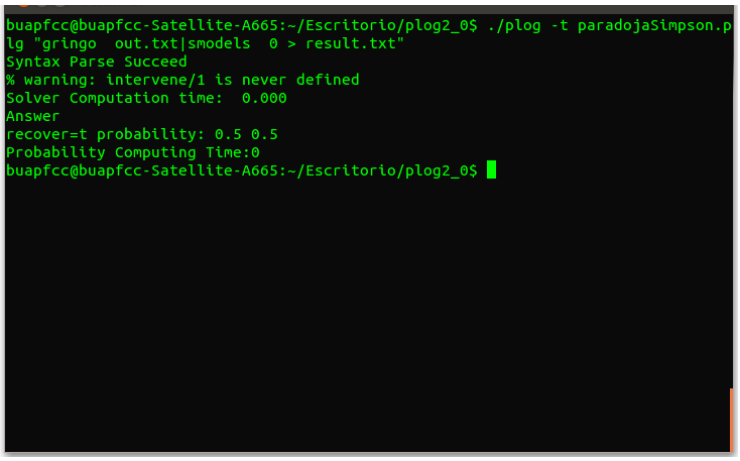

Figura 5.1 Muestra la probabilidad obtenida utilizando gringo y smodels

Como se mencionó anteriormente Plog da como resultado dos archivos de texto plano out.txt y resul.txt a continuación se presenta su contenido después de la ejecución del comando presentado anteriormente.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
1{male(X_):boolean(X_)}1:-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
1{recover(X_):boolean(X_)}1:-not intervene(recover).
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
1{drug(X_):boolean(X_)}1:-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
```

```
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo result.txt

```
smodels version 2.34. Reading...done
Answer: 1
Stable Model: male(t) recover(t) drug(t)
Answer: 2
Stable Model: male(t) recover(f) drug(t)
Answer: 3
Stable Model: male(f) recover(f) drug(t)
Answer: 4
Stable Model: male(f) recover(t) drug(t)
False
Duration: 0.000
Number of choice points: 3
Number of wrong choices: 3
Number of atoms: 31
Number of rules: 34
Number of picked atoms: 11
Number of forced atoms: 0
Number of truth assignments: 66
Size of searchspace (removed): 4 (0)
```

5.2.3 Utilizando gringo y claspD

Para obtener la probabilidad de una semántica estables es necesario contar con un archivo base en este caso utilizamos el ejemplo de la paradojaSimposon.plg (véase en Anexo B)que es convertido a un programa normal lógico por Plog y es almacenado en out.txt este a su vez sirve como archivo de entrada a el grounder gringo (recordemos que este forma parte de Plog, véase capítulo 3 Análisis y Diseño) que convierte el archivo en instrucciones legibles que son transmitidas mediante tuberías a el solver clapsD dando este como resultado un archivo más llamado result.txt que

almacena los mundos posibles para la semántica estable.

A continuación se muestra el comando que se debe de ejecutar mediante el uso de la consola.

```
./plog -t paradojaSimpson.plg "gringo  out.txt|claspD -n 0
> result.txt"
```

La siguiente figura muestra claramente los comandos mencionados anteriormente y la probabilidad resultante.

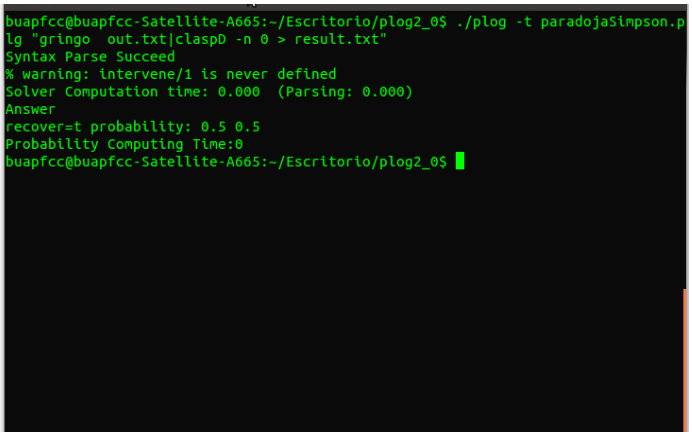

Figura 5.2 Muestra la probabilidad obtenida utilizando gringo y claspD

Como se mencionó anteriormente Plog da como resultado dos archivos de texto plano out.txt y resul.txt a continuación se presenta su contenido después de la ejecución del comando presentado anteriormente.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
1{male(X_):boolean(X_)}1:-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
1{recover(X_):boolean(X_)}1:-not intervene(recover).
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
```

```
1{drug(X_):boolean(X_)}1:-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo resul.txt

```
claspD version 1.1. Reading...done

Answer: 1
male(t) recover(t) drug(t)
Answer: 2
male(t) recover(f) drug(t)
Answer: 3
male(f) recover(t) drug(t)
Answer: 4
male(f) recover(f) drug(t)

Models      : 4
Time        : 0.000  (Parsing: 0.000)
```

5.2.4 Utilizando lparser y smodels

Para obtener la probabilidad de una semántica estables es necesario contar con un archivo base en este caso utilizamos el ejemplo de la paradojaSimposon.plg (véase en Anexo B)que es convertido a un programa normal lógico por Plog y es almacenado en out.txt este a su vez sirve como archivo de entrada a el grounder lparser(recordemos que este forma parte de Plog, véase capítulo 3 Análisis y Diseño) que convierte el archivo en instrucciones legibles que son transmitidas mediante tuberías a el solver smodels dando este como resultado un archivo más llamado result.txt que almacena los mundos posibles para la semántica

estable.

A continuación se muestra el comando que se debe de ejecutar mediante el uso de la consola.

```
./plog -t paradojaSimpson.plg "lparse --true-negation -W
none out.txt|smodels 0 > result.txt"
```

La siguiente figura muestra claramente los comandos mencionados anteriormente y la probabilidad resultante.

Figura 5.3 Muestra la probabilidad obtenida utilizando lparser y smodels.

Como se mencionó anteriormente Plog da como resultado dos archivos de texto plano out.txt y resul.txt a continuación se presenta su contenido después de la ejecución del comando presentado anteriormente.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
1{male(X_):boolean(X_)}1:-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
1{recover(X_):boolean(X_)}1:-not intervene(recover).
```

```
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
1{drug(X_):boolean(X_)}1:-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo result.txt:

```
smodels version 2.34. Reading...done

Answer: 1
Stable Model: drug(t) male(t) recover(f)
pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(3,5))
pa(r3,drug(t),di_(3,4)) pd(r1,male(f)) pd(r1,male(t))
pd(r2,recover(f)) pd(r2,recover(t)) pd(r3,drug(f))
pd(r3,drug(t))

Answer: 2

Stable Model: drug(t) male(t) recover(t)
pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(3,5))
pa(r3,drug(t),di_(3,4)) pd(r1,male(f)) pd(r1,male(t))
pd(r2,recover(f)) pd(r2,recover(t)) pd(r3,drug(f))
pd(r3,drug(t))

Answer: 3

Stable Model: drug(t) male(f) recover(t)
pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(1,5))
pa(r3,drug(t),di_(1,4)) pd(r1,male(f)) pd(r1,male(t))
pd(r2,recover(f)) pd(r2,recover(t)) pd(r3,drug(f))
pd(r3,drug(t))

Answer: 4

Stable Model: drug(t) male(f) recover(f)
pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(1,5))
pa(r3,drug(t),di_(1,4)) pd(r1,male(f)) pd(r1,male(t))
```

```
pd(r2,recover(f)) pd(r2,recover(t)) pd(r3,drug(f))
pd(r3,drug(t))
False

Duration: 0.000

Number of choice points: 3
Number of wrong choices: 3
Number of atoms: 28
Number of rules: 31
Number of picked atoms: 11
Number of forced atoms: 0

Number of truth assignments: 63
Size of searchspace (removed): 4 (0)
```

5.2.5 Utilizando lparse y claspD

Para obtener la probabilidad de una semántica estables es necesario contar con un archivo base en este caso utilizamos el ejemplo de la paradojaSimposon.plg (véase en apéndice --)que es convertido a un programa normal lógico por Plog y es almacenado en out.txt este a su vez sirve como archivo de entrada a el grounder lparser(recordemos que este forma parte de Plog, véase capítulo 3 Análisis y Diseño) que convierte el archivo en instrucciones legibles que son transmitidas mediante tuberías a el solver claspD dando este como resultado un archivo más llamado result.txt que almacena los mundos posibles para la semántica estable.

A continuación se muestra el comando que se debe de ejecutar mediante el uso de la consola.

```
./plog -t paradojaSimpson.plg "lparse --true-negation -W
none out.txt|claspD -n 0 > result.txt"
```

La siguiente figura muestra claramente los comandos mencionados anteriormente y la probabilidad resultante.

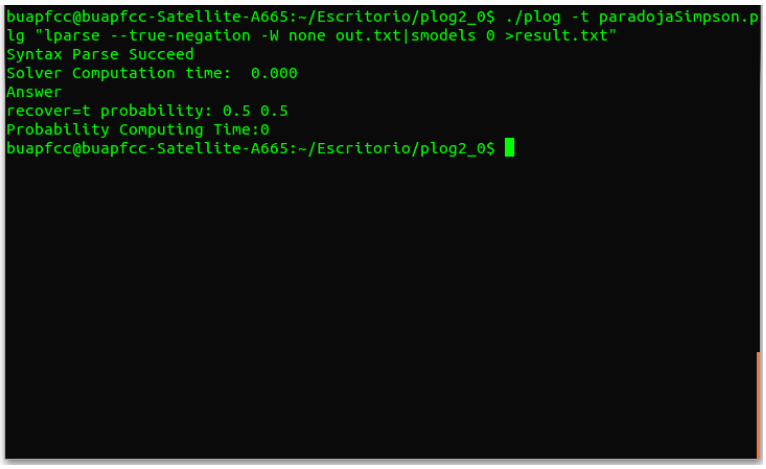

Figura 5.4 Muestra la probabilidad obtenida utilizando lparser y claspD.

Como se mencionó anteriormente Plog da como resultado dos archivos de texto plano out.txt y resul.txt a continuación se presenta su contenido después de la ejecución del comando presentado anteriormente.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
1{male(X_):boolean(X_)}1:-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
1{recover(X_):boolean(X_)}1:-not intervene(recover).
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
1{drug(X_):boolean(X_)}1:-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo result.txt:

```
claspD version 1.1. Reading...done

Answer: 1

drug(t) male(t) recover(t) pa(r1,male(t),di_(1,2))
pa(r2,recover(t),di_(3,5)) pa(r3,drug(t),di_(3,4))
pd(r1,male(f)) pd(r1,male(t)) pd(r2,recover(f))
pd(r2,recover(t)) pd(r3,drug(f)) pd(r3,drug(t))

Answer: 2

drug(t) male(t) recover(f) pa(r1,male(t),di_(1,2))
pa(r2,recover(t),di_(3,5)) pa(r3,drug(t),di_(3,4))
pd(r1,male(f)) pd(r1,male(t)) pd(r2,recover(f))
pd(r2,recover(t)) pd(r3,drug(f)) pd(r3,drug(t))

Answer: 3

drug(t) male(f) recover(t) pa(r1,male(t),di_(1,2))
pa(r2,recover(t),di_(1,5)) pa(r3,drug(t),di_(1,4))
pd(r1,male(f)) pd(r1,male(t)) pd(r2,recover(f))
pd(r2,recover(t)) pd(r3,drug(f)) pd(r3,drug(t))

Answer: 4

drug(t) male(f) recover(f) pa(r1,male(t),di_(1,2))
pa(r2,recover(t),di_(1,5)) pa(r3,drug(t),di_(1,4))
pd(r1,male(f)) pd(r1,male(t)) pd(r2,recover(f))
pd(r2,recover(t)) pd(r3,drug(f)) pd(r3,drug(t))

Models      : 4
Time        : 0.000  (Parsing: 0.000)
```

5.2 Probabilidad obtenida usando la semántica P-estable con Plog

Para obtener la probabilidad de una semántica p-estables es necesario utilizar liimo que es un sistema que cuenta con una versión propia de lparser y de gringo además de un solver llamado exx el cual se utiliza en la gramática p-estable.

5.2.6 Utilizando liimo

Para obtener la probabilidad de una semántica p-estables es necesario contar con un archivo base en este caso utilizamos el ejemplo de la paradojaSimposon.plg (véase en Anexo B)que es convertido a un programa normal lógico por Plog y es almacenado en out.txt este a su vez sirve como archivo de entrada a el grounder lparser(recordemos que este forma parte de Plog, véase capítulo 3 Análisis y Diseño) que convierte el archivo en instrucciones legibles que son transmitidas mediante tuberías a el solver exx dando este como resultado un archivo más llamado result.txt que almacena los mundos posibles para la semántica estable.

A continuación se muestra el comando que se debe de ejecutar mediante el uso de la consola.

```
./plog -t paradojaSimpson.plg "cat out.txt | liimo -gl -pr
> result.txt"
```

La siguiente figura muestra claramente los comandos mencionados anteriormente y la probabilidad resultante.

Figura 5.5 Muestra la probabilidad obtenida utilizando liimo.

Como se mencionó anteriormente Plog da como resultado dos archivos de texto plano out.txt y resul.txt a continuación se presenta su contenido después de la ejecución del comando presentado anteriormente.

Contenido del archivo out.txt:

```
boolean(t).
boolean(f).
#domain boolean(Y).
male( t) | male( f):-not intervene(male).
pd(r1,male(X_)):-not intervene(male),boolean(X_).
#show male(X_).
recover( t) | recover( f):-not intervene(recover).
pd(r2,recover(X_)):-not intervene(recover),boolean(X_).
#show recover(X_).
drug( t) | drug( f):-not intervene(drug).
pd(r3,drug(X_)):-not intervene(drug),boolean(X_).
#show drug(X_).
pa(r1,male(t),di_(1,2)).
pa(r2,recover(t),di_(3,5)):-male(t),drug(t).
pa(r2,recover(t),di_(7,10)):-male(t),drug(f).
pa(r2,recover(t),di_(1,5)):-male(f),drug(t).
pa(r2,recover(t),di_(3,10)):-male(f),drug(f).
pa(r3,drug(t),di_(3,4)):-male(t).
pa(r3,drug(t),di_(1,4)):-male(f).
:- not drug(t).

#hide.
#show pd(X,Y).
#show pa(X,Y,Z).
#show recover(t).
```

Contenido del archivo result.txt:

```
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(1,5))
male(f) pa(r3,drug(t),di_(1,4)) recover(t) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(1,5))
male(f) pa(r3,drug(t),di_(1,4)) recover(f) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(3,5))
male(t) pa(r3,drug(t),di_(3,4)) recover(t) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
drug(t) pa(r1,male(t),di_(1,2)) pa(r2,recover(t),di_(3,5))
male(t) pa(r3,drug(t),di_(3,4)) recover(f) pd(r1,male(f))
pd(r1,male(t)) pd(r2,recover(f)) pd(r2,recover(t))
pd(r3,drug(f)) pd(r3,drug(t))
```

Capitulo 6

Conclusiones

La principal aportación de este trabajo es la extensión del sistema Plog. La extensión consiste en agregar un solver para p-estable el cual aporta la capacidad de interpretar el sentido común del mundo real. Además se agregó otro solver para la semántica estable el cual es claspD. Al agregar este solver nos da la oportunidad de comparar resultados entre claspD y smodels nativo de Plog.

Al realizar la comparación observamos que los dos sistemas función al mismo nivel y su tiempo de ejecución casi es el mismo.
Finalmente se agregó otro grounder el cual se llama gringo que es el usado por omisión de claspD e igualmente se puede comparar con lparser y se llegó a que los dos están al mismo nivel.

Cabe destacar que cada uno de los programas mencionados son sistemas independientes y aun que su forma de compartir información es a partir de archivos, tiene cierto grado de complejidad pues se debe estandarizar las salidas y las entradas de cada uno de ellos para su fácil entendimiento.

Como se ha mostrado en el capítulo 5, las pruebas realizadas a la implementación de Plog extendido demuestran que los mundos posibles de la semántica estable son también aceptados por la semántica p-estable, esto es demostrado teóricamente en el capítulo 2, y visto en la práctica por Plog.

También para el usuario que utiliza estos sistemas resulta mucho más fácil el estar trabajando en un solo sistema y realizar una

ejecución de un solo paso y no varios.

Podemos notar que esta tesis aun deja el camino abierto para trabajos a futuro, como por ejemplo agregar la interfaz gráfica.
Poder hacer investigación sobre otras semánticas y agregarlas a Plog, además que aún hay mucho que se puede ejemplificar con este sistema.

Bibliografía

Baral C., Brewka G., and Schlipf J., editors. 2007.Proceedings of the Ninth International
Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07), volume 4483 of Lecture Notes in Artificial Intelligence. Springer-verlag

Balduccini, M., Gelfond, M., Nogueira, M., Watson, R., and Barry, M. 2001. An A-Prolog decision support system for the space shuttle - I. Proceedings of Practical Aspects of Declarative Languages. 169-183.

Gelfond M., Baral, C., Rushton N. 2008. Probabilistic reasoning with answer sets. In Proceedings of LPNMR-7}. edit. Springer-Verlag. 21-33.

Gelfond M., Baral, C., Rushton N. 2008. Probabilistic reasoning with answer sets. In Proceedings of LPNMR-7, 2004}. edit. Springer-Verlag. 21-33.

Gelfond, M., Baral, C. 2004. Reasoning in Answer Set Prolog. Disponible en: http://www.public.asu.edu/~cbaral/aquaint04-06/slides/baral-gelfond-baltimoreaug04.pdf.

Gelfond M. and Lifschitz V. 1988. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, 5th Conference on Logic Programming, pages 1070–1080. MIT Press.

Gelfond M. and Lifschitz V. 1991. Classical negation in logic programs and disjunctive databases. New Generation Computing, 9:365–385.

Gomes, C., Kautz P., Sabharwal H. and Selman B. 2008. Satisfiability solvers. In van Harmelen, F.; V.; Lifschitz, and Porter, B., eds., Handbook of Knowledge Representation. Elsevier.

Lin F. and Zhao Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence, 157(1-2):115–137.

Olvera A. 2011. Tesis de Licenciatura. Fundamentos de bases de

datos deductivas: razonamiento con incertidumbre. Universidad de las Américas. Puebla.

Osorio M., Carballido J. L. and Zepeda C. 2010. Expressing p-stable semantics based on stable semantics. ISBN: 978-1-4244-5353-5, pages 227-231 IEEE.

Osorio M., Arrazola J., and Carballido J. L.. 2008. Logical weak completions of paraconsistent logics. Journal of Logic and Computation, doi: 10.1093/logcom/exn015.

Osorio M., Navarro J.A., Arrazola J., and Borja V. 2006. Logics with common weak completions. Journal of Logic and Computation, 16(6):867–890.

Shawn H. A. 2006. First Course in Logic. Oxford University Press.

Simons P., Niemel¨a I. and Soininen T. 2002.Extending and implementing the stable model semantics. Artificial Intelligence, 138(1-2):181–234,

Weijun, Zhu. 2008. The P-log system [Software de cómputo]. Texas, E.U. Texas Tec University. Disponible en: http://www.cs.ttu.edu/~wezhu/

Anexo A

No se profundiza en los mecanismos de funcionamiento, pero si se exponen los parámetros de uso para la ejecución de los solver o de los grounder en su defecto.

La sinopsis de la línea de comando de lparse es el siguiente:

Uso: lparse [-1][-c *constante* = *numero*]
[-d all | facts | positive | none]
[-D] [-g *archivo*][-i][-n *número*]
[-r[1 | 2 | 3]][-t][-v][-w]
[-W *advertencia*][--dlp][--atom-file *archivo*]
[--allow-inconsistent-answers]
[--drop-quotes][--partial]
[--separate-weigh-definitions]
[--true-negation][--version]
$archivo_1$ $archivo_2$ ...

Entonces un ejemplo de su llamado en línea de comandos seria:

lparse [-opciones] file | smodels

Se da el significado de cada opción para lparse:

-1	Formato de saliva versión smodel 1.x.
-c const=n	Define un identificador que es una constante numérica con el valor de n.
-d all |facts| positive | none	Controla el dominio de los predicados que son emitidos. La opción por defecto es facts: • All: todas las literales de dominio se emiten. Las literales insatisfactible negativas no se eliminan del cuerpo. • Facts: todos los predicados de dominio en las reglas del cuerpo son eliminadas de la salida. • None: todas las literales de dominio se eliminan • Positive: el dominio de las literales negativas en las reglas del cuerpo se

	eliminan.
-D	Para depurar, uso para desarrolladores.
-g *archivo*	Lee una salida del archivo nombrado.
-i	Deshabilita las funciones internas.
-n *número*	Establece el número deseado de modelos.
-r[1 | 2 | 3]	Habilita la extensión de modelo regular, el parámetro numérico es opcional se utiliza para controlar que restricciones de integridad se eliminan.
-t	Imprimir salida como texto.
-v	Imprime información de la versión de lparse.
-w número	Establece el peso por defecto de la literal. Si esta opción no se da, el peso por defecto será 1.
-W advertencia	Controlar las advertencias emitidas por lparse. Los valores posibles son: all, arity, extended, library, unsat, weight, syntax, typo, and error.
--true-negation	Habilita la extensión de la negación clásica.
atom-file *arch*	La tabla de símbolos del programa le conecte la salida.
--allow-inconsistent-answers	Habilita la extensión de negación clásica y además también informes conjuntos de respuestas incoherente
--dlp	Usa la semántica de programación lógica disyuntiva con reglas de elección.
--drop-quotes	Quitar las comillas de cadenas cuando sea posible. Por ejemplo "a" se convierte **a** pero "123" no cambia.
--partial	Permite la extensión del modelo parcial. Este modificador es idéntico en comportamiento a la opción -r 2.

Figura A.1. Opciones para la ejecución de lparse.

También hay opciones para ejecutar smodels, es importante conocerlas para este trabajo. Dado que uno de los objetivos es poder hacer la unión de todos los solver mencionados con anterioridad para ser ejecutados con plog. A continuación se describen estas opciones:

Uso: smodels [number][-w][-nolookhead][backnump][-randomize]
[-internal][-tries *número*][-conflicts *número*]

[-seed número]

number	Determina el número de modelos estables a ser calculados, si se omite poner el número de modelos deseados, entones se indica que calcule todos los posibles.
-backjump	Permite retroceder para saltar por encima de varios puntos de elección.
-nolookahead	No utiliza lookahead en absoluto.
-sloppyheuristic	Reduce el uso de la heurística lookahead, esto puede acelerar a smodels si el programa tiene una estructura fácil.
-randomize	Hace una búsqueda completamente al azar
-internal	Simplifica el programa e imprime en un formato interno
-w	Calculo solo modelos wf(well-founded) del programa
-tries número	Hace una búsqueda al azar estocástica para un número de veces.
-conflicts número	Al hacer una búsqueda estocástica, se detiene cuando el número es encontrado
-seed número	Usa el número como la semilla para la parte aleatoria .

Figura A.2. Opciones para la ejecución de smodels.

Gringo no necesita que las opciones de ejecución y el nombre del archivo tengan algún orden específico, a continuación se presentan una forma de invocación y después se describen las opciones de ejecución.
gringo [opciones | nombre del archivo]

- help,-h	Imprimir información de ayuda y salir.
- version,-V	Imprimir la información de versión y sale
- verbose [= n],-V	Impresión adicional (progreso) la información durante el cálculo. Nivel de detalle, el nivel uno y dos no son utilizados actualmente por lo gringo. El nivel tres imprime representaciones internas de las reglas durante el procesamiento. (Esto puede ser utilizado para identificar errores semánticos en un programa de entrada o identificar cuellos de botella en el rendimiento).
- const, -c c = t	remplaza las ocurrencias de una constante c con un término t.
--text, -t	La salida del programa ground en formato texto (legible para las personas).
--reify	La salida del programa ground en formato de

	hechos.
--lparse,-l	la salida del programa ground en formato numérico lparse
--ground,-g	Activar el modo ligero para el procesamiento del ground del programa de entrada. (Esta opción se recomienda para omitir sobrecarga innecesaria si el programa de entrada es ya ground, pero conduce a un error de sintaxis).
--ifixed = n	Utilice n como el número de pasos fijos si el programa de entrada contiene sentencias #acumulative o # volative. (Esta opción permite el manejo de los programas escritos para iclingo).
--ibase	Procesar sólo la parte estática (que puede ser iniciada por una instrucción de #base) de un programa de entrada. (Esta opción puede ser usada para investigar la configuración básica de un problema incluyendo algunos vínculos mutables.)
--dep-graph = nombre del archivo	Esta opción se puede utilizar para obtener la dependencia del grafo en formato de archivo dot.
- shift	Elimina la disyunción por shifting (ver [Gelfond M. & Lifschitz V. 1991]).

Figura A.3. Opciones para la ejecución de gringo

La línea de comandos por defecto cuando se invoca gringo es:

gringo -- lparse

Es decir, gringo por lo general emite un programa ground en formato numérico de lparse, tratado por varios solucionadores de ASP.

Al introducirse en estos solver se encontró algo muy interesante, que clasp y claspD pueden utilizarse tanto gringo como con lparse, esto es interesante, pues están siguiendo el estándar propuesto por AnsProlog, y esto facilita el desarrollo de la implementación en el que se está trabajando.

A continuación se presenta la forma de ejecutar desde línea de comando de una forma abstracta el solver claspD.

claspD [number | options | filename]

Las mas importantes son:

Opciones	Descripción
--heuristic	Para seleccionar entre diversas heurísticas de decisión
--restarts	Para controlar el comportamiento de reinicio claspD
--deletion	para controlar el tamaño / actualización de los lemas de claspD
--help	Para poder identificar los demás comandos

Figura A.4 Opciones para la ejecución de claspD.

Ejemplo de la línea de comandos con lparse y después con gringo.

```
lparse --dlp program.lp | claspD 0
```

Esta línea ejecuta claspD con lparse para calcular todos los modelos posibles, la siguiente utiliza gringo

gringo programa.lp | claspD 0

Como se puede observar, es muy práctico por que el formato de archivo que leen los grounder es el mismo y por tanto el formato de salida que es el de entrada para los solver también es el mismo, así que para combinar tanto solver como grounder es más fácil para

la implementación que se está realizando y finalmente ser combinado con plog. En la siguiente sección se trata del sistema plog.

Anexo B

En este anexo se presenta el contenido de los archivos considerados como base para este trabajo.

I. paradojasimpson.plg

```
#domain boolean(Y).
boolean = {t,f}.
male, recover, drug : boolean.
[r1] random(male).
[r2] random(recover).
[r3] random(drug).
%probability information
[r1] pr(male = t)=1/2.
[r2] pr(recover = t | male = t,drug = t) =3/5.
[r2] pr(recover = t | male = t, drug = f)=7/10.
[r2] pr(recover = t | male = f, drug = t)=1/5.
[r2] pr(recover = t | male = f, drug = f)=3/10.
[r3] pr(drug = t|male = t)=3/4.
[r3] pr(drug = t|male = f)=1/4.
%Queries
?{recover = t} | obs(drug=t).
%?{recover = t} | obs(drug=f).
%?{recover = t} | do(drug=t).
%?{recover = t} | do(drug=f).
```

II. datos.plg

```
dice={d1,d2}.
#domain dice(D).
score={1..6}.
#domain score(S).
boolean = {t,f}.
person={mike,john}.
roll:dice->score.
owns(d1,mike).
owns(d2,john).
[r(D)] random(roll(D)).
[r(D)] pr(roll(D)=6|owns(D,mike))=1/4.
%The query could be:
?{roll(d1) = 6}.
```

III. MontyHall.plg

```
#domain doors(D).
doors={1,2,3}.
boolean = {t, f}.
prize,open,selected: doors.
can_open: doors -> boolean.
can_open(D) = f :- selected = D.
can_open(D) = f :- prize = D.
can_open(D) = t :- not can_open(D) = f.
[r1] random (prize).
[r2] random (selected).
[r3] random (open:{X:can_open(X) = t}).
%Queries:
?{prize=1}|obs(selected=1),obs(open=2).
%?{prize=3}|obs(selected=1),obs(open=2).
```

IV. rata.plg

```
boolean = {t, f}.
arsenic, death: boolean.
[r(1)] random (arsenic).
[r(2)] random (death).
[r(1)] pr(arsenic = t) = 2/5.
[r(2)] pr(death = t | arsenic = t) = 4/5.
[r(2)] pr(death = t | arsenic = f) = 1/10.
%Queries:
%?{arsenic = t}.
?{arsenic = t} | obs(death = t).
%?{arsenic = t} | do(death = t).
%?{death = t} | obs(arsenic = t).
%?{death = t} | do(arsenic = t).
```

V. ruleta.plg

```
#domain gun(G).
#domain boolean(B).
boolean={t, f}.
gun={1,2}.
pull_trigger:gun -> boolean.
fatal: gun -> boolean.
is_dead: boolean.
is_dead(t) :- fatal(G) = t.
is_dead(f) :- not is_dead(t).
pull_trigger(G) = t.
[r(G)] random(fatal(G)) :- pull_trigger(G) = t.
%probability information
[r(G)] pr(fatal(G) = t)=1/6.
%The query could be:
?{is_dead(t)}.
```