Proyecto 1
Atacando el problema SAT: Métodos sintácticos vs semánticos

Lógica Computacional 2019-2

Alumnos:

José Jhovan Gallardo Valdez
No. de cuenta: 310192815

Jorge Luis Garcia De Santiago
No. de cuenta: 311320219

Introducción

Hoy día con el continuo desarrollo y evolución de las tecnologías computacionales, económicas e incluso sociales en las que se encuentra inmerso el ser humano es que se deja de tener tiempo para razonar y entender las cosas de una manera rápida y lógica; donde se ve que con los avances tecnológicos el hombre ha dejado de darle sus propias soluciones a los problemas cotidianos que le afectan e incluso que tienen que solucionar puesto que es a través de las computadoras a las que se les da el trabajo para que sean ellas quienes resuelvan estos sin necesidad de la ayuda humana.

Igualmente, se hace necesario dar otro concepto, que para mí, es de gran importancia, como lo es La lógica matemática, ésta es una parte de la lógica y las matemáticas, que consiste en el estudio matemático de la lógica y en la aplicación de este estudio a otras áreas de las matemáticas, tiene estrechas conexiones con la ciencias de la computación y la lógica filosófica. La investigación en lógica matemática ha jugado un papel fundamental en el estudio de los fundamentos de las matemáticas, que es en lo que quiero ahondar.

La lógica matemática es la disciplina que trata de métodos de razonamiento. En un nivel elemental, la lógica proporciona reglas y técnicas para determinar si es o no valido un argumento dado. El razonamiento lógico se emplea en matemáticas para demostrar teoremas; en ciencias de la computación para verificar si son o no correctos los programas; en las ciencias física y naturales, para sacar conclusiones de experimentos; y en las ciencias sociales y en la vida cotidiana, para resolver una multitud de problemas. Ciertamente se usa en forma constante el razonamiento lógico para realizar cualquier actividad.

La relación de la lógica con la matemática desarrollo el intento de buscar un lenguaje en que los problemas derivados de la validez o invalidez de los razonamientos fuesen tratados como un simple cálculo, un problema que consistiría en mirar si estaba o no de acuerdo con la tabla. Con George Boole en 1847, se inició la construcción sistemática de la lógica matemática, él fue el primero en aplicar el álgebra a la lógica, dando origen a una lógica de clases y una lógica sentencial. A partir de este momento, la lógica matemática se ira construyendo a imagen y semejanza de las lenguas naturales.

¿Por qué es importante entonces estudiar ciencias de la computación? Vivimos en un mundo digitalizado, computarizado, programable y para poder entender esa realidad necesitamos la ciencia de la computación. El ingeniero que usa un computador para diseñar un puente debe entender cómo se computaron los estimados de carga máxima y qué tan confiables son. Los ciudadanos educados que depositan su voto mediante un sistema digital o participan en subastas de

eBay deben tener una compresión básica de los algoritmos que subyacen en esos sistemas, así como de los temas de seguridad y privacidad presentes, cuando se trasmite información y se almacena digitalmente.

La lógica está presente en la computación a través de los siguientes aspectos:

Es tan importante la relación lógica-computación que todo ordenador tiene una unidad en la cual se realizan las operaciones lógicas; es la unidad aritmético–lógica. En ella, se efectúan las operaciones lógicas de cualquier programa. Nos referimos a los operadores lógicos "y", "o", etc., los cuales trabajan en base a las tablas de verdad.

La lógica se hace presente en los programas. Cada uno de ellos es un conjunto formal y secuencial de operaciones, las cuales permiten realizar un trabajo. Decimos "formal " y con ello evidenciamos de la lógica forma, puesto que teóricamente, un mismo programa puede estar referido a varios contenidos, siempre y cuando tengan los mismos esquemas.

¿Por qué es importante SAT? Este es de suma importancia pues a partir de este se puede identificar la NP-completes de otros problemas NP que se sospechen puedan ser NP-completos. Esto de acuerdo al Teorema de Cook: Se dice que un problema A en NP es NP-completo cuando, para cada otro problema B en NP, A es más difícil de resolver que B. En la práctica se demuestra que un problema B que está en NP es NP-completo de la siguiente forma [ICS 161, 1996]: Se comienza con un problema A que se conoce es NP-completo (como por ejemplo SAT) y cuya característica es que todos los problemas en NP son reducibles a A. Si A es NP-completo y B está en NP, y A es reducible a B (se puede utilizar el algoritmo que resuelva B para resolver A, esto es, A es más fácil que B), entonces B es NP-completo. Y por lo tanto todos los problemas en NP son reducibles también a B. De aquí la importancia del problema SAT en la prueba de NP-completes para problemas NP.

Lógica proposicional

La lógica proposicional es el sistema lógico m as simple.

Se encarga del manejo de proposiciones mediante conectivos lógicos.

Una proposición es un enunciado que puede calificarse de verdadero o falso.

• El es bribón

• Tu eres caballero

• Hoy es jueves.

• Hay vida en Marte.

NO son proposiciones: ¿Llueve?, el perro azul, ¡ Viva Pancho Villa!, el cerro de la silla,

Juan Pérez, etc.

El lenguaje PROP

Definimos ahora un lenguaje formal para la lógica de proposiciones.

El alfabeto consta de:

Símbolos o variables proposicionales (un numero infinito) : p1, . . . , pn, ...

Constantes lógicas: $\bot$, $>$

Conectivos u operadores lógicos: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$

Símbolos auxiliares: (,)

El conjunto de expresiones o formulas atómicas, denotado ATOM consta de:

Las variables proposicionales: p1, . . . , pn, ...

Las constantes $\bot$, $>$.

Las expresiones que formaran nuestro lenguaje PROP, llamadas usualmente formulas, se

definen recursivamente como sigue:

Si $\varphi \in$ ATOM entonces $\varphi \in$ PROP. Es decir, toda formula atómica es una f formula.

Si $\varphi \in$ PROP entonces $(\neg\varphi) \in$ PROP

Si $\varphi, \psi \in$ PROP entonces $(\varphi \wedge \psi),(\varphi \vee \psi),(\varphi \rightarrow \psi),(\varphi \leftrightarrow \psi) \in$ PROP.

Son todas.

La última cláusula de la definición garantiza que el conjunto PROP es el mínimo conjunto

cerrado bajo las tres primeras reglas.

La definición anterior puede darse en la llamada forma de Backus-Naur para definir gramáticas como sigue:

$\varphi, \psi ::=$ V arP $| \bot | > | (\neg\varphi) | (\varphi \wedge \psi) | (\varphi \vee \psi) | (\varphi \rightarrow \psi) | (\varphi \leftrightarrow \psi)$

V arP ::= p1 | p2 | . . . | pn | . . .

Precedencia y Asociatividad de los Operadores Lógicos

Para evitar el uso de par´entesis al máximo definimos la siguiente precedencia de operadores

de mayor a menor:

¬

∧, ∨

→

↔

De manera que la operación a realizar primero es la dada por el conectivo de mayor precedencia. Veamos algunos ejemplos:

¬p → r es (¬p) → r

p → q ∧ r es p → (q ∧ r)

¬p → q ↔ r es

(¬p) → q

↔ r

p → q ∧ r ↔ s ∨ ¬t es

p → q ∧ r

↔

s ∨ (¬t)

p ∧ q → ¬q ∨ s es (p ∧ q) → ((¬q) ∨ s)

Obsérvese que los operadores ∧, ∨ tienen la misma precedencia de manera que expresiones

como p ∧ q ∨ r son ambiguas y no pueden ni deben ser utilizadas.

Acerca de la asociatividad, los operadores ∧, ∨, ↔ son asociativos de manera que expresiones

como p ∨ q ∨ r o ¬p ↔ p ∨ s ↔ t están libres de ambigüedades.

Por otra parte, el operador → no es asociativo, pero adoptaremos la convención usual de asociarlo

a la derecha, es decir expresiones del estilo φ1 → φ2 → φ3 se entenderán como φ1 → (φ2 → φ3)

Algunas funciones de importancia

Las siguientes funciones serán de utilidad más adelante. Su definición recursiva se deja como

ejercicio:

Profundidad de una fórmula: depth(φ) devuelve la profundidad o altura del ´árbol de análisis sintáctico de φ

Número de conectivos de una fórmula: con(φ) devuelve el número de conectivos de φ.

Variables de una fórmula: vars(φ) devuelve el conjunto o lista de variables que figuran en φ (sin repeticiones)

Atómicas en una fórmula: atom(φ) devuelve el número de presencias de fórmulas atómicas en φ.

Las siguientes relaciones entre algunas de las funciones recién especificadas deben verificarse mediante recursión estructural:

con(φ) < 2

depth(φ)

depth(φ) ≤ con(φ).

atom(φ) ≤ 2con(φ) + 1.

Significado de los conectivos lógicos

Veamos el significado de cada conectivo lógico:

La Negación

La negación de la formula P es la fórmula ¬P.

Símbolo utilizado: ¬

Correspondencia con el español: No, no es cierto que, es falso que, etc.

Otros símbolos: ∼P, P.

Semántica (tabla de verdad):

P ¬P

1 0

0 1

La Disyunción

La disyunción de las fórmulas P, Q es la fórmula P ∨ Q. Las fórmulas P, Q se llaman disyuntos.

Símbolo utilizado: ∨

Correspondencia con el español: o.

Otros símbolos: P + Q, P | Q.

Semántica (tabla de verdad):

P Q P ∨ Q

1 1 1

1 0 1

0 1 1

0 0 0

La Conjunción

La conjunción de las fórmulas P, Q es la fórmula P ∧Q. Las fórmulas P, Q se llaman conjuntos.

Símbolo utilizado: ∧

Correspondencia con el español: y, pero.

Otros símbolos: P & Q, P · Q o también P Q sin usar un operador explícito.

Semántica (tabla de verdad):

P Q P ∧ Q

1 1 1

1 0 0

0 1 0

0 0 0

La Implicación

La implicación o condicional de las formulas P, Q es la formula P → Q. La fórmula P es el

antecedente y la formula Q es el consecuente de la implicación.

Símbolo utilizado: →

Correspondencia con el español: P → Q significa: si P entonces Q; Q, si P; P solo si Q;

P es condición suficiente para Q; Q es condición necesaria para P.

Otros símbolos: P ⇒ Q, P ⊃ Q.

Semántica (tabla de verdad):

P Q P → Q

1 1 1

1 0 0

0 1 1

0 0 1

La Equivalencia

La equivalencia o bicondicional de las formulas P, Q es la fórmula P ↔ Q.

Símbolo utilizado: ↔

Correspondencia con el español: P es equivalente a Q; P si y solo si Q;

P es condición necesaria y suficiente para Q.

Otros símbolos: P ⇔ Q, P ≡ Q.

Semántica (tabla de verdad):

P Q P ↔ Q

1 1 1

1 0 0

0 1 0

0 0 1

Semántica formal de los conectivos lógicos

Definición 1 El tipo de valores booleanos denotado Bool se define como Bool = {0, 1}

Definición 2 Un estado o asignación de las variables (proposicionales) es una función

I : V arP → Bool

Dadas n variables proposicionales existen 2n

estados distintos para estas variables.

Definición 3. Dado un estado de las variables I : VarP → Bool, definimos la interpretación de las fórmulas con respecto a I como la función

I : PROP → Bool tal que:

I(p) = I(p) para p ∈ VarP, es decir

I (VarP) = I (VarP).

I (T) = 1

I (⊥) = 0

I (¬φ) = 1 syss I (φ) = 0.

I (φ ∧ ψ) = 1 syss I (φ) = 0 y I (ψ) = 1.

I (φ ∨ ψ) = 0 syss I (φ) =0 y I (ψ) = 0.

I (φ → ψ) = 0 syss I (φ) = 1 y I (ψ) = 0.

I(φ ↔ ψ) = 1 syss I (φ) = I (ψ).

Obsérvese que dado un estado de las variables I, la interpretación I generada por I está determinada de manera única por lo que de ahora en adelante escribiremos simplemente I en lugar de I. Este abuso notacional es práctica común en ciencias de la computación y se conoce como sobrecarga de operadores.

El siguiente lema es de importancia para restringir las interpretaciones de inter´es al analizar una fórmula.

Lema 1 (Coincidencia) Sean I1, I2 : PROP → Bool dos estados que coinciden en las variables

proposicionales de la fórmula φ, es decir I1(p) = I2(p) para toda p ∈ vars(φ). Entonces I1(φ) =

I2(φ).

Demostración.

Inducción estructural sobre φ a

El lema anterior implica que a´un cuando existen una infinidad de estados, dada una fórmula φ

basta considerar ´unicamente aquellos que difieren en las variables proposicionales de φ, a saber 2n

estados distintos si φ tiene n variables proposicionales.

2.1. Observaciones sobre la implementaci´on funcional

Estados: un estado I puede implementarse directamente como una función

```
i :: VarP ->Bool
```

o bien, dado que las listas son una estructura de datos muy conveniente en Haskell podemos

definir el tipo de Estados como

```
type Estado = [VarP]
```

con lo cual representamos a un estado mediante la lista de variables proposicionales que est´an

definidas como verdaderas. Por ejemplo si definimos I1(p) = I1(q) = 1 e I1(r) = 0 entonces

implementamos a dicho estado como i1=[p,q]. Similarmente el estado i2=[r,s,t] representa

al estado I2(r) = I2(s) = I2(t) = 1.

Función de interpretación : la función de interpretación I

?

se implementa mediante una función

interp :: Estado ->Prop ->Bool especificada por:

interp i φ = I

?

(φ)

donde el estado i representa al estado I de la teor´ıa.

Estados posibles: dada una fórmula φ con n-variables proposicionales, la función estados ::

Prop ->[Estado] devuelve la lista con los 2n

estados distintos para φ. Su definici´on es

estados phi = subconj (vars phi)

donde vars:: Prop ->[VarP] es la función que devuelve la lista de variables proposicionales

que figuran en φ (sin repetici´on) y subconj :: [a] ->[[a]] es una función que dada una

lista `, devuelve la lista con las sublistas de `. Por ejemplo:

subconj [1,2] = [[1,2],[1],[2],[]]

3. Conceptos Sem´anticos B´asicos

Dada una fórmula φ podemos preguntarnos ¿ Cu´antas interpretaciones hacen verdadera a φ?

las posibles respuestas llevan a las siguientes definiciones:

Definici´on 4 Sea φ una fórmula. Entonces

Si I(φ) = 1 para toda interpretación I decimos que φ es una tautologia o fórmula v´alida y

escribimos |= φ.

Si I(φ) = 1 para alguna interpretación I decimos que φ es satisfacible, que φ es verdadera

en I o que I es modelo de φ y escribimos I |= φ

Si I(φ) = 0 para alguna interpretación I decimos que φ es falsa o insatisfacible en I o que I

no es modelo de φ y escribimos I 6|= φ

Si I(φ) = 0 para toda intepretaci´on I decimos que φ es una contradicci´on o fórmula no

satisfacible.

Similarmente si Γ es un conjunto de fórmulas decimos que:

Γ es satisfacible si tiene un modelo, es decir, si existe una interpretación I tal que I(φ) = 1

para toda φ ∈ Γ. Lo cual denotamos a veces, abusando de la notaci´on, con I(Γ) = 1.

Γ es insatisfacible o no satisfacible si no tiene un modelo, es decir, si no existe una inter-

pretaci´on I tal que I(φ) = 1 para toda φ ∈ Γ.

Con respecto a las tablas de verdad tenemos las siguientes observaciones:

Una fórmula φ es satisfacible si en alguna linea de la tabla de verdad φ toma el valor 1. En

caso contrario, es decir si en todas las lineas toma el valor 0 es insatisfacible (contradicci´on).

Un conjunto de fórmulas Γ es satisfacible si existe alguna linea de la tabla de verdad en la que

todas las fórmulas de Γ toman el valor 1.

Proposici´on 1 Sea Γ un conjunto de fórmulas, φ ∈ Γ, τ una tautologia y χ una contradicci´on.

Si Γ es satisfacible entonces

• Γ \ {φ} es satisfacible.

• Γ ∪ {τ} es satisfacible.

• Γ ∪ {χ} es insatisfacible.

Si Γ es insatisfacible entonces

• Γ ∪ {ψ} es insatisfacible, para cualquier ψ ∈ PROP.

• Γ \ {τ} es insatisfacible.

Demostraci´on. Ejercicio a

Si Γ es satisfacible y ψ tambi´en nada se puede decir acerca de Γ ∪ {χ}. Analogamente si Γ es

insatisfacible y ψ no es tautologia nada se puede decir de Γ \ {ψ}.

5

Proposici´on 2 Sea Γ = {φ1, . . . , φn} un conjunto de fórmulas.

Γ es satisfacible syss φ1 ∧ . . . ∧ φn es satisfacible.

Γ es insatisfacible syss φ1 ∧ . . . ∧ φn es una contradicci´on.

Demostraci´on. Ejercicio a

3.1. Observaciones sobre la implementaci´on funcional

Para implementar los conceptos de la definici´on 4 nos serviremos de la función modelos ::

Prop ->[Estados] que dada una fórmula φ devuelve la lista de todos sus modelos. Esta función se

especifica como sigue:

modelos(φ) = [i|i ∈ estados(φ), interp i φ = True]

Es decir, modelos(φ) devuelve la lista con todos aquellos estados I tales que I(φ) = 1. Los conceptos

de la definici´on 4 se implementan comparando las listas estados(φ) y modelos(φ).

Tautolog´ıa: tautologia:: Prop ->Bool. φ es tautolog´ıa syss todos sus estados (interpreta-

ciones) son modelos.

tautologia(phi) = estados(phi) == modelos(phi)

Satisfacible en una interpretación : satisfen:: Estado ->Prop ->Bool. φ es satisfacible en

I (I |= φ) syss I(φ) = 1.

satisfen i phi = interp i phi == True

Satisfacible: satisf::Prop ->Bool. φ es satisfacible syss alg´un estado (interpretación) es

modelo, es decir, si la lista de modelos no es vac´ıa.

satisf(phi) = modelos(phi) /= [ ]

Insatisfacible en una interpretación : insatisfen:: Estado ->Prop ->Bool. φ es insatisfaci-

ble en I (I 6|= φ) syss I(φ) = 0.

insatisfen i phi = interp i phi == False

Insatisfacible o contradicci´on: contrad::Prop ->Bool. φ es insatisfacible o contradicci´on syss

ning´un estado (interpretación) es modelo, es decir, si la lista de modelos es vac´ıa.

contrad(phi) = modelos(phi) = [ ]

Extensi´on a conjuntos: las funciones que generan los estados y modelos para un conjunto

(lista) de fórmulas Γ y aquellas que verifican si Γ es satisfacible o insatisfacible se denotan:

• estadosConj :: [Prop] ->[Estado]

• modelosConj :: [Prop] ->[Estado]

• satisfenConj:: Estado ->[Prop] ->Bool

• satisfConj:: [Prop] ->Bool

• insatisfenConj:: Estado ->[Prop] ->Bool

• insatisfConj:: [Prop] ->Bool

la implementaci´on de las mismas es similar al caso de las funciones para fórmulas y se deja

como ejercicio.

4. Equivalencia de Fórmulas

Definici´on 5 Dos fórmulas φ, ψ son equivalentes si I(φ) = I(ψ) para toda interpretación I. En

tal caso escribimos φ ≡ ψ.

Es f´acil ver que la relaci´on ≡ es una relaci´on de equivalencia.

Proposici´on 3 Sean φ, ψ fórmulas. Entonces φ ≡ ψ syss |= φ ↔ ψ

Demostraci´on. Ejercicio a

Dado que ya implementamos la función tautologia podemos definir f´acilmente una función

equiv :: Prop ->Prop ->Bool que decida si dos fórmulas son equivalentes usando la proposici´on

anterior:

```
equiv phi psi = tautologia ( phi <-> psi )
```

La siguiente propiedad es fundamental para el razonamiento ecuacional con equivalencias l´ogicas.

Proposici´on 4 (Regla de Leibniz) Si $\varphi \equiv \psi$ y $p \in ATOM$ entonces $\chi[p := \varphi] \equiv \chi[p := \psi]$

Demostraci´on. Inducción estructural sobre χ. a

Las fórmulas equivalentes son de gran importancia pues nos permiten definir algunos conectivos

en t´erminos de otros, as´ı como simplificar fórmulas compuestas.

4.1. Algunas equivalencias l´ogicas

Conmutatividad:

$P \vee Q \equiv Q \vee P \quad P \wedge Q \equiv Q \wedge P$

Asociatividad:

$P \vee (Q \vee R) \equiv (P \vee Q) \vee R \quad P \wedge (Q \wedge R) \equiv (P \wedge Q) \wedge R$

Distributividad

$P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R) \quad P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$

Leyes de la negaci´on:

- Doble Negaci´on: $\neg\neg P \equiv P$
- De Morgan: $\neg(P \vee Q) \equiv \neg P \wedge \neg Q \quad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$
- $\neg(P \rightarrow Q) \equiv P \wedge \neg Q$
- $\neg(P \leftrightarrow Q) \equiv \neg P \leftrightarrow Q \equiv P \leftrightarrow \neg Q$.

Propiedades de la implicaci´on:

- Contrapositiva: $P \rightarrow Q \equiv \neg Q \rightarrow \neg P$
- Importaci´on/Exportaci´on: $P \wedge Q \rightarrow R \equiv P \rightarrow (Q \rightarrow R)$

Eliminaci´on de conectivos:

$P \rightarrow Q \equiv \neg P \vee Q \quad P \rightarrow Q \equiv \neg(P \wedge \neg Q)$

$P \leftrightarrow Q \equiv (P \rightarrow Q) \wedge (Q \rightarrow P)$ $P \leftrightarrow Q \equiv (P \wedge Q) \vee (\neg P \wedge \neg Q)$

$P \vee Q \equiv \neg P \rightarrow Q$ $P \wedge Q \equiv \neg(P \rightarrow \neg Q)$

Leyes simplificativas:

• Idempotencia: $P \vee P \equiv P$ $P \wedge P \equiv P$

• Absorci´on: $P \vee (P \wedge Q) \equiv P$ $P \wedge (P \vee Q) \equiv P$

• Neutros: $P \vee \bot \equiv P$ $P \wedge > \equiv P$

• Inversos:

◦ Tercero Excluido: $P \vee \neg P \equiv >$

◦ Contradicci´on: $P \wedge \neg P \equiv \bot$

• Dominancia: $P \vee > \equiv >$ $P \wedge \bot \equiv \bot$

Veamos un ejemplo de simplificaci´on de una fórmula mediante el uso de equivalencias l´ogicas y

la regla de Leibniz:

$(p \vee q) \wedge \neg(\neg p \wedge q) \equiv (p \vee q) \wedge (\neg\neg p \vee \neg q)$

$\equiv (p \vee q) \wedge (p \vee \neg q)$

$\equiv p \vee (q \wedge \neg q)$

$\equiv p \vee \bot$

$\equiv p$

4.2. Eliminaci´on de implicaciones y equivalencias

Las equivalencias l´ogicas que permiten eliminar los conectivos $\rightarrow$, $\leftrightarrow$ son de gran importan-

cia para la llamada l´ogica clausular que estudiaremos m´as delante. Se deja como ejercicio definir

recursivamente e implementar las siguientes funciones:

elimEquiv:: Prop ->Prop tal que

elimEquiv φ = ψ syss ψ ≡ φ y ψ no contiene el s´ımbolo ↔.

elimImp:: Prop ->Prop tal que

elimImp φ = ψ syss ψ ≡ φ y ψ no contiene el s´ımbolo →.

Obsérvese que estas funciones pueden definirse de diversas formas pero las m´as adecuadas son

usando las equivalencias usuales:

φ ↔ ψ ≡ (φ → ψ) ∧ (ψ → φ) φ → ψ ≡ ¬φ ∨ ψ

5. Consecuencia L´ogica

Definici´on 6 Sean Γ un conjunto de fórmulas y φ una fórmula. Decimos que φ es consecuencia

l´ogica de Γ si para toda intepretaci´on I que sastisface a Γ se tiene I(φ) = 1. Es decir si se cumple

que siempre que I satisface a Γ entonces necesariamente I satisface a φ. En tal caso escribimos

Γ |= φ

N´otese que la relaci´on de consecuencia l´ogica esta dada por una implicaci´on de la forma

I(Γ) = 1 ⇒ I(φ) = 1

lo cual informalmente significa que todo modelo de Γ es modelo de φ. De manera que no se afirma

nada acerca de la satisfacibilidad del conjunto Γ, simplemente se asume que es satisfacible y en tal

caso se prueba que la fórmula φ tambi´en lo es con la misma interpretación .

Obsérvese la sobrecarga del s´ımbolo |= que previamente utilizamos para denotar satisfacibilidad

I |= φ y tautolog´ıas |= φ.

Proposici´on 5 La relaci´on de consecuencia l´ogica cumple las siguientes propiedades:

Si φ ∈ Γ entonces Γ |= φ.

Principio de refutaci´on: Γ |= φ syss Γ ∪ {¬φ} es insatisfacible.

Γ |= φ → ψ syss Γ ∪ {φ} |= ψ.

Insatisfacibilidad implica trivialidad: Si Γ es insatisfacible entonces Γ |= φ para toda φ ∈

PROP.

Si Γ |= ⊥ entonces Γ es insatisfacible.

φ ≡ ψ syss φ |= ψ y ψ |= φ.

|= φ (es decir si φ es tautolog´ıa) syss ∅ |= φ (es decir φ es consecuencia l´ogica del conjunto

vac´ıo).

Demostraci´on. Ejercicio a

5.1. Observaciones sobre la implementaci´on funcional

La función consecuencia: [Prop] ->Prop ->Bool se espec´ıfica como:

consecuencia Γ φ = True syss Γ |= φ.

La implementaci´on se sirve del principio de refutaci´on dado en la proposici´on 5

consecuencia gamma phi = insatisfConj ((neg phi):gamma)

donde neg:Prop ->Prop es una función que construye la negaci´on de una fórmula.

Otra posibilidad es verificar que no existen modelos de Γ que no sean modelos de φ, es decir,

que la lista de modelos de Γ que no son modelos de φ es vac´ıa. Esto se puede hacer directamente

usando el mecanismo de listas por comprensi´on de Haskell:

```
consecuencia gamma phi = null [i | i <- estadosConj (phi:gamma),
satisfenConj i gamma,
not (satisfen i phi)
]
```

5.2. Correctud de Argumentos L´ogicos

Definici´on 7 Un argumento l´ogico es una sucesi´on de fórmulas φ1, . . . , φn llamadas premisas y

una fórmula ψ llamada conclusi´on. Dicha sucesi´on se escribe usualmente como

φ1

.

.

.

φn

∴ ψ

Un problema central de la l´ogica consiste en decidir si un argumento dado es correcto o no, para

lo cual la herramienta principal es la noci´on de consecuencia l´ogica.

Definici´on 8 Un argumento con premisas φ1, . . . , φn y conclusi´on ψ es l´ogicamente correcto si la

conclusi´on se sigue l´ogicamente de las premisas, es decir si {φ1, . . . , φn} |= ψ

Esta definici´on deja ver de inmediato una implementaci´on argcorrecto :: [Prop] ->Prop

->Bool dado que ya tenemos un programa (función) para decidir la consecuencia l´ogica.

```
argcorrecto [phi1,...,phin] psi = consecuencia [phi1,...,phin] psi
```

Es importante tener un m´etodo formal para decidir la correctud de argumentos del lenguaje

natural los cuales pueden parecer correctos y no serlo. Veamos un ejemplo sencillo considerando el

siguiente argumento:

Si hoy es viernes entonces ma˜nana es s´abado, ma˜nana es s´abado, por lo tanto hoy es viernes.

Palabras como “por lo tanto, as´ı que, luego entonces, de forma que, etc” se˜nalan la conclusi´on del

argumento.

La representaci´on l´ogica formal del argumento es:

v → s

s

∴ v

De manera que el argumento es correcto syss {v → s, s} |= v. Pero la interpretación I(v) =

0, I(s) = 1 confirma que el argumento es incorrecto.

Es importante resaltar que al analizar argumentos lo ´unico que interesa es su forma l´ogica y no

su significado, como lo muestra el siguiente ejemplo:

Si hoy tirila y Chubaka es kismi entonces Chubaka es borogrove y si hoy no tirila entonces hay

fefos. M´as a´un sabemos que no hay fefos y que Chubaka es kismi, luego entonces Chubaka es

borogrove.

Lo cual nos lleva a

$t \wedge k \rightarrow b$

$\neg t \rightarrow f$

$\neg f \wedge k$

$\therefore b$

Es facil ver que $\{t \wedge k \rightarrow b, \neg t \rightarrow f, \neg f \wedge k\} \models b$ por lo que el argumento es correcto.

5.3. M´etodos sem´anticos para demostrar la correctud de argumentos

Para mostrar la correctud del argumento l´ogico $\varphi 1, \ldots, \varphi n / \therefore \psi$ mediante interpretaciones, se

puede proceder de alguna de las siguientes dos formas:

M´etodo directo: probar la consecuencia $\varphi 1, \ldots, \varphi n \models \psi$.

M´etodo indirecto (refutaci´on): probar que el conjunto $\{\varphi 1, \ldots, \varphi n, \neg\psi\}$ es insatisfacible.

Por supuesto que existen otros m´etodos para mostrar la correctud de un argumento l´ogico, que

resultan m´as adecuados para la automatizaci´on, como son el c´alculo exhaustivo de una tabla de

verdad (sumamente ineficiente), el uso de tableaux o el m´etodo de resoluci´on binaria que veremos

m´as adelante.

5.4. Algunos argumentos correctos relevantes

P

$P \rightarrow Q$

$\therefore Q$

(Modus ponens)

$\neg Q$

$P \rightarrow Q$

$\therefore \neg P$

(Modus tollens)

$P \rightarrow Q$

$Q \rightarrow R$

$\therefore P \rightarrow R$

(Silogismo hipotetico)

$P \rightarrow R$

$Q \rightarrow R$

$\therefore P \vee Q \rightarrow R$

(P rueba por casos)

11

P

Q

$\therefore P \wedge Q$

(Introd. $\wedge$)

$P \wedge Q$

$\therefore P$

$P \wedge Q$

$\therefore Q$

(Elim. $\wedge$)

P

$\therefore P \vee Q$

Q

$\therefore P \vee Q$

(Introd. ∨)

P ∨ Q

¬Q ∨ R

∴ P ∨ R

(Resoluci´on binaria)

Forma Normal Conjuntiva

La llamada forma normal conjuntiva permite expresar cualquier fórmula proposicional como

una conjunción de disyunciones llamadas clausulas.

Una literal ` es una formula atómica (variable proposicional p, ⊥ o >) o la negación de una formula atómica

Una literal es negativa si es una negación, en otro caso decimos que es positiva.

Dada una literal ` definimos su literal contraria, denotada `

c

, como sigue:

`

c =

a si ` = ¬a

¬a si ` = a

Donde a es una formula atómica, es decir, una variable proposicional, > o ⊥.

El par {`, `c} se llama un par de literales complementarias.

Las literales constituyen los objetos básicos del lenguaje clausular, a partir de los cuales se

construyen las cláusulas.

Una cláusula C es una literal o una disyunción de literales.

Una formula φ est´a en forma normal conjuntiva (fnc) si y solo si es de la forma

$C1 \wedge C2 \wedge \ldots \wedge Cn$

donde para cada $1 \leq i \leq n$, Ci es una cláusula.

En particular se sigue que cualquier literal y cualquier clausula están en forma normal conjuntiva.

Los conceptos anteriores de literal, clausula y forma normal conjuntiva se definen de manera

breve mediante las siguientes gramáticas:

$a ::= \bot \mid > \mid p$ − − atómicas

$` ::= a \mid \neg a$ − − literales

$C ::= ` \mid ` \vee C$ − − clausulas

$F ::= C \mid C \wedge F$ − − formas normales conjuntivas

Cualquier fórmula proposicional φ puede transformarse algorítmicamente en una

formula ψ tal que $\varphi \equiv \psi$ y ψ está en forma normal conjuntiva. En este caso ψ se denota con

fnc(φ).

Demostración.

Basta aplicar las siguientes transformaciones en orden:

Obtener la forma normal negativa correspondiente.

Reescribir las conjunciones y disyunciones mediante las leyes distributivas:

$(\varphi \wedge \psi) \vee (\varphi \wedge \chi) \equiv \varphi \wedge (\psi \vee \chi)$

$(\varphi \vee \psi) \wedge (\varphi \vee \chi) \equiv \varphi \vee (\psi \wedge \chi)$

Opcionalmente se puede simplificar usando que

¬φ ∨ φ ≡ > > ∨ φ ≡ > > ∧ φ ≡ φ ⊥ ∨ φ ≡ φ ⊥ ∧ φ ≡ ⊥ φ ∨ φ ≡ φ

a

Ejemplo 2.1 Obtener la forma normal conjuntiva de φ = (¬p → q) → (¬r → s). Aplicando el

método recién descrito obtenemos:

(¬p → q) → (¬r → s) ≡

¬(¬¬p ∨ q) ∨ (¬¬r ∨ s) ≡

(¬p ∧ ¬q) ∨ (r ∨ s) ≡

(¬p ∨ r ∨ s) ∧ (¬q ∨ r ∨ s) = fnc(φ)

Programa: Amplia explicación del funcionamiento del programa.

El código del proyecto se divide en tres archivos principales Syntax.hs, Semantcs.hs y Tableux.hs.

En Syntax.hs se encuentran nuestra definición del tipo de dato Prop que corresponde a las fórmulas de lógica proposicional, además tiene métodos útiles para la transformación de estas fórmulas, es decir, para poder transformarlas a FNN y FNC, métodos que son usados en Semantics.hs y Tableux.hs.

En Semantics.hs se encuentran las funciones referentes a la semántica de la lógica de predicados. Es decir, encontramos funciones que nos permiten conocer información sobre la satisfacibilidad de fórmulas y de conjuntos de estas.

En Tableux.hs encontramos las funciones necesarias para construir tableux partiendo de lógicas proposicionales, y verificar si son satisfacibles o no.

El archivo Main.hs se construyen formulas aleatorias y se verifica su satisfacibilidad por dos métodos diferentes, usando semántica y tableux, e imprime el tiempo que le tomo a cada método verificar esto.

Conclusiones

La lógica proposicional tiene múltiples aplicaciones en las ciencias de la computación, en este proyecto en particular exploramos el problema de satisfacibilidad. Por lo que es importante poseer herramientas para atacarlo de la forma más eficiente posible, más aún considerando que es un problema NP completo. Gracias al código en el archivo Main.hs pudimos concluir que el método programado en Semantics.hs puede tomar mucho tiempo si las fórmulas que evaluamos tienen una gran cantidad de variables, por lo que Tableux puede ser más efectivo en esos casos, pero si tenemos pocas variables y muchos conectores, usar el método de Semantics seria más efectivo, pues no es necesario construir el Tableux.

Bibliografía

[1] Ben-Ari, M. Mathematical Logic for Computer Science, 3rd ed. Springer Publishing Com- pany, Incorporated, 2012.

[2] Halpern, J. Y., Harper, R., Immerman, N., Kolaitis, P. G., Vardi, M. Y., and Vianu, V. On the unusual effectiveness of logic in computer science. Bulletin of Symbolic Logic 7, 2 (2001), 213–236.

[3] Huth, M., and Ryan, M. Logic in Computer Science: Modelling and Reasoning About Systems. Cambridge University Press, New York, NY, USA, 2004.

[4] Lipovaca, M. Learn You a Haskell for Great Good!: A Beginner's Guide, 1st ed. No Starch Press, San Francisco, CA, USA, 2011.

[5] Marques-Silva, J. Practical applications of boolean satisfiability. 2008 9th International Workshop on Discrete Event Systems (2008), 74–80.

[6] Miranda Perea, F. E., and Viso Gurovich, E. Matemáticas discretas. La prensas de las ciencias, Ciudad de México, México, 2016.