Lógica Computacional 2017-2, nota de clase 12
Lógica Ecuacional

Favio Ezequiel Miranda Perea Araceli Liliana Reyes Cabello
Lourdes Del Carmen González Huesca Pilar Selene Linares Arévalo

8 de mayo de 2017

1. Introducción

Consideremos la siguiente proposición:

Proposición 1 $p \wedge q \equiv q \wedge p$

¿Qué significa exactamete que la proposición 1 es válida? ¿Es posible dar un contraejemplo a esa proposición? Para mostrar que la sentencia es válida en lógica proposicional, podríamos proceder realizando la tabla de verdad (tabla con cuatro renglones) y verificar que no existe un contraejemplo, es decir, una asignación de valores para las variables involucradas de tal forma que $p \wedge q \leftrightarrow q \wedge p$ se evalúe a falso.

Consideremos ahora la siguiente proposición:

Proposición 2 $x + y = y + x$

¿Podemos dar un contrajemplo a la proposición? ¿Cómo demostramos que ésta válida? ¿Podríamos realizar algo similar al caso de lógica proposicional? Desafortunadamente, los valores que podemos asignarle a las variables es infinito, lo cual significa que no podríamos terminar de completar la tabla de asignaciones. Por lo tanto, necesitamos una forma diferente de demostrar que una proposición como la anterior es válida sin recurrir a su tabla de asignaciones.

Hasta ahora hemos trabajado con lógica de proposiciones y de predicados y en ellos hemos usado una noción de equivalencia entre expresiones, $e_1 \equiv e_2$, que está fuertemente basada en la semántica formal de cada lógica.

Pero si se desean realizar diversas equivalencias resulta tedioso hacerlas formalmente y para ello hemos recurrido a utilizar sólo equivalencias demostradas previamente para justificar cada paso de alguna secuencia del estilo $e_1 \equiv e_2 \equiv \ldots \equiv e_n$.

Realizar este tipo de manipulaciones debe ser una característica útil de cualquier lógica o teoría y debe corresponder a una noción de igualdad $e_1 = e_2$, no necesariamente sintáctica pero que conserve la idea subyacente de objetos equiparables.
Veamos algunos casos en donde se ocupa un razonamiento puramente ecuacional para demostrar:

Lógica proposicional Las equivalencias de fórmulas proposicionales son usadas para simplificar o referirse a fórmulas cuyo valor booleano es el mismo, por ejemplo:

$$\begin{array}{lcll} (p \vee q) \wedge \neg(\neg p \wedge q) & \equiv & (p \vee q) \wedge (\neg\neg p \vee \neg q) & \text{distributividad negación} \\ & \equiv & (p \vee q) \wedge (p \vee \neg q) & \text{eliminación doble neg} \\ & \equiv & p \vee (q \wedge \neg q) & \text{asociatividad} \\ & \equiv & p \vee \bot & \text{absorción} \\ & \equiv & p & \text{neutro} \end{array}$$

Este razonamiento también debería expresarse utilizando igualdad en lugar de equivalencia.

Programa funcional Consideremos la definición de árboles binarios de números naturales en HASKELL, el aplanado de árboles es una función que dado un árbol devuelve una lista con todos los elementos en él:

```
data Tree = Leaf Nat | Node Tree Tree

flatten :: Tree -> [Nat]
flatten (Leaf n) = [n]
flatten (Node l r) = flatten l ++ flatten r
```

La función anterior es ineficiente, tiene complejidad cuadrática: si el número de nodos del subárbol izquierdo es m_1 entonces append y `flatten` están recorriendo esos m_1 elementos dos veces además de visitar cada hoja del subárbol derecho m_2.

Si quisiéramos mejorar la definición de `flatten` debemos eliminar el uso de append, para esto usaremos una especificación más general de aplanado de árboles utilizando una lista auxiliar:

```
flatten' t ns = flatten t ++ ns
```

Buscaremos una definición que la cumpla tal especificación por medio de inducción:

- Caso Base:

```
flatten' (Leaf n) ns
        = { especificacion de flatten' }
flatten (Leaf n) ++ ns
        = { aplicando flatten }
[n] ++ ns
        = { aplicando ++ }
n:ns
```

- Caso Inductivo: [1]

```
flatten' (Node l r) ns
        = { especificacion de flatten' }
(flatten l ++ flatten r ) ++ ns
        = { asociatividad de ++ }
flatten l ++ (flatten r ++ ns)
        = { hipotesis induccion para l }
flatten' l (flatten r ++ ns)
        = { hipotesis induccion para r }
flatten' l (flatten' r ns)
```

[1] Donde la hipótesis de inducción supone verdadera la propiedad para los subárboles.

Obteniendo una definición mejorada que no utiliza la definición de append y que usa una lista auxiliar que hace las funciones de **acumulador**:

```
flatten' :: Tree -> [Nat] -> [Nat]
flatten' (Leaf n) ns = n : ns
flatten' (Node l r) ns = flatten' l (flatten' r ns)
```

Finalmente, quisiéramos referirnos a la equivalencia en ambas definiciones, esto se puede describir como la igualdad de resultados al aplicar cualquiera de ellas:

para cualquier lista `l`, *se cumple que* `flatten l = flatten' l []`

El ejemplo anterior es un razonamiento sobre la especificación de estructuras de datos (árboles, listas, números naturales) y funciones de ellas (aplanado, concatenación, etc.). Una demostración utilizando el lenguaje de lógica de predicados resultaría en un ejercicio más extenso y con notación lógica que hemos usado hasta ahora: un lenguaje de predicados $\mathcal{L}$ para las estructuras, funciones y relaciones y un modelo para éste $\mathcal{M}_{\mathcal{L}}$. Así la pregunta anterior se traduce en demostrar que:

$$\mathcal{M}_{\mathcal{L}} \models \forall x(Tree(x) \rightarrow flatten(x) = flatten'(x, nil))$$

Para aligerar estas demostraciones, se puede usar un razonamiento ecuacional para decidir si una ecuación en particular se sigue de un conjunto de ‘ ecuaciones dado como por ejemplo en programas funcionales.
Por tanto requerimos de una noción formal para el manejo de igualdades que forme parte de cualquier lenguaje formal. En esta nota estudiaremos a fondo este concepto que también es útil en otras áreas de computación.

1.1. Reglas y sistemas de deducción

Para describir las reglas válidas que permiten *encadenar* razonamientos respecto a ecuaciones utilizaremos una representación que relaciona dos condiciones o enunciados mediante una línea horizontal. Estos esquemas son usados en muchas áreas matemáticas y computacionales ya pueden leerse en dos sentidos: de arriba hacia abajo y viceversa, dependiendo de un propósito en el desarrollo de una demostración: generar, derivar o deducir información.

Un sistema de deducción tiene reglas básicas o axiomas que son reglas que cuentan únicamente con la parte inferior para representar hechos o conclusiones sin premisas:

$$\frac{}{B}$$

También tienen reglas de deducción que cuentan con ambas partes, premisas y conclusiones, útiles en la generación de información:

$$\frac{A_1\ A_2\ \ldots\ A_m}{B}$$

Definición 1 *Un sistema de deducción para una teoría $\mathcal{T}$, es un sistema que consta de un número finito de reglas y axiomas que establecen la relación entre enunciados de $\mathcal{T}$.*
Una deducción o derivación de un enunciado $\mathcal{J}$ es una secuencia finita $\Pi = \langle \mathcal{J}_1, \ldots, \mathcal{J}_k \rangle$ tal que $\mathcal{J}_k = \mathcal{J}$ y cada $\mathcal{J}_i$, $1 \leq i \leq k$ es una instancia de una regla del sistema de deducción.

2. Lógica ecuacional

Un sistema de deducción para razonar respecto a ecuaciones es el llamado Cálculo de Birkhdef, el cual contiene cinco reglas:

$$\frac{}{\Gamma, H \vdash_{\mathcal{B}} H}\ \textsc{Hyp} \qquad \frac{}{\Gamma \vdash_{\mathcal{B}} t = t}\ \textsc{Refl} \qquad \frac{\Gamma \vdash_{\mathcal{B}} s = t}{\Gamma \vdash_{\mathcal{B}} t = s}\ \textsc{Sym} \qquad \frac{\Gamma \vdash_{\mathcal{B}} t = r \qquad \Gamma \vdash_{\mathcal{B}} r = s}{\Gamma \vdash_{\mathcal{B}} t = s}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash_{\mathcal{B}} t = s}{\Gamma \vdash_{\mathcal{B}} t\sigma = s\sigma}\ \textsc{Inst} \qquad \frac{\Gamma \vdash_{\mathcal{B}} t_1 = s_1 \quad \ldots \quad \Gamma \vdash_{\mathcal{B}} t_n = s_n}{\Gamma \vdash_{\mathcal{B}} f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)}\ \textsc{Congr}$$

Como mencionamos antes, el axioma básico de juicios hipotéticos es el que permite obtener una información conocida. El resto de las reglas establece las caracteríticas de la relación simétrica y transitiva. Las últimas dos reglas son útiles para la abstracción de patrones (instanciación) y para la construcción de nuevos términos (congruencia).

Estas reglas son suficientes para hacer demostraciones pero se puede extender con algunas otras reglas que factoricen usos de varias reglas:

$$\frac{\Gamma, H : t = s \vdash E[x := s\sigma]}{\Gamma, H : t = s \vdash E[x := t\sigma]}\ \textsc{Rewrite(H)} \qquad \frac{\Gamma, H : t = s \vdash E[x := t\sigma]}{\Gamma, H : t = s \vdash E[x := s\sigma]}\ \textsc{Rewrite(H)}$$

Veamos otros ejemplos:

Ejemplo 2.1 [Ecuaciones de términos] Recordemos la gramática para términos de cualquier lenguaje de primer orden:

$$t ::= x \mid c \mid f(t_1, t_2, \ldots, t_n)$$

Ejemplo 2.2 [otro ejemplo]

3. Formalización de teorías

El sistema de lógica ecuacional presentado ayuda a realizar demostraciones de ecuaciones de forma clara. Una aplicación de esto es la formalización de teorías que requieren de ecuaciones para la demostración de propiedades o especificaciones de los elementos en la teoría. Veamos dos ejemplos extendidos:

Ejemplo 3.1 [ejemplo de formalización (anillos, grupos, conjuntos)] Considere el siguiente conjunto de ecuaciones $\mathcal{A}$, que definen un anillo:

A1 $x + 0 = x$

A2 $x + (-x) = 0$

A3 $x + y = y + x$

A4 $x + (y + z) = (x + y) + z$

A5 $x \cdot 1 = x$

A6 $1 \cdot x = x$

A7 $x \cdot (y \cdot z) = (x \cdot y) \cdot z$

A8 $x \cdot (y + z) = x \cdot y + x \cdot z$

A9 $(x + y) \cdot z = x \cdot z + y \cdot z$

Demuestre lo siguiente:

1. $\mathcal{A} \vdash 0 + x = x$
2. $\mathcal{A} \vdash (-x) + x = 0$
3. $\mathcal{A}, x + z = y + z \vdash x = y$
4. $\mathcal{A}, z + x = z + y \vdash x = y$
5. $\mathcal{A} \vdash -(-x) = x$
6. $\mathcal{A} \vdash (-x) + (-y) = -(x + y)$
7. $\mathcal{A} \vdash 0 \cdot x = 0$
8. $\mathcal{A} \vdash x \cdot 0 = 0$
9. $\mathcal{A} \vdash x \cdot y + x \cdot (-y) = x \cdot 0$
10. $\mathcal{A} \vdash (-x) \cdot y = x \cdot (-y)$
11. $\mathcal{A} \vdash x \cdot (-y) = -(x \cdot y)$
12. $\mathcal{A} \vdash (-x) \cdot (-y) = x \cdot y$

Ejemplo 3.2 Considera una formalización para listas de elementos de cierto tipo A y la siguiente propiedad:

Cualquier lista no vacía puede descomponerse en dos listas y un elemento tal que ese elemento está en una posición intermedia en la lista original.

Las siguientes fórmulas de primer orden corresponden a la formalización de la propiedad:

1. Funcionamente:

$$\forall x\Big(L_A(x) \land x \neq nil \rightarrow \big(\exists y, z, w\big(L_A(y) \land L_A(z) \land A(w) \land x = app(y, cons(w, z))\big)\big)\Big)$$

2. Relacionalmente:

$$\forall x\Big(L_A(x) \land x \neq nil \rightarrow \big(\exists y, z, w\big(L_A(y) \land L_A(z) \land A(w) \land App(y, cons(w, z), x)\big)\big)\Big)$$

¿Cuáles son las herramientas necesarias para demostrar la validez de alguna de éstas fórmulas?

Este último ejemplo requiere de una regla con más poder que las incluidas en el sistema de deducción $\mathcal{B}$, es necesario esclarecer casos en los que la forma de un objeto es fundamental para la demostración es decir, el uso de inducción es inminente.