Lógica Computacional 2017-2, nota de clase 13
Lógica Ecuacional

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

22 de mayo de 2017

1. Introducción

Durante este curso hemos enfocado nuestra atención en el estudio de la validez de enunciados lógicos y métodos para hacerlo. Consideremos las siguientes proposiciones que involucran una noción de igualdad y analicemos su posible valor de verdad:

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

¿Qué significa exactamete que esta proposición sea válida? ¿Es posible dar un contraejemplo a esta 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.

Proposición 2 $x + y = y + x$

Suponiendo un significado aritmético a esta proposición, ¿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.

Aunado a lo anterior, hay una pregunta importante que nos debemos hacer: ¿Cuándo una igualdad se puede inferir a partir de un conjunto dado de ecuaciones? Por ejemplo, a partir de las ecuaciones:

$$\begin{aligned} x + y &= y + x \\ x + (y + z) &= (x + y) + z \end{aligned}$$

podemos concluir que $w_1 + (w_2 + w_3) = (w_3 + w_2) + w_1$ pero no podemos afirmar que $w_1 + w_1 = w_1 + w_2$.

En esta nota, estudiaremos la Lógica Ecuacional, una teoría que trabaja sobre expresiones que involucran al operador de igualdad, que reflejan la idea subyacente de objetos equiparables y no necesariamente de forma sintáctica. La lógica ecuacional juega un papel importante en el razonamiento sobre equivalencias lógicas, igualdades algebraicas, definiciones de programas, propiedades de estructuras de datos, entre otros.
Veamos algunos ejemplos en donde se ocupa un razonamiento puramente ecuacional para demostrar:

Lógica proposicional En esta lógica hemos usado una noción de equivalencia entre expresiones $e_1 \equiv e_2$, que está basada en la semántica formal. 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 $e_1 \equiv e_2 \equiv \ldots \equiv e_n$.

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:

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

El aplanado de árboles es una función que dado un árbol devuelve una lista con todos sus elementos:

```
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 respectivamente, además de que se visita cada hoja del subárbol derecho m_2 para obtener la lista de elementos del árbol original.

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 ++ }
```

[1] Donde la hipótesis de inducción supone verdadera la propiedad para los subárboles `l` y `r`.

```
flatten l ++ (flatten r ++ ns)
        = { hipotesis induccion para l }
flatten' l (flatten r ++ ns)
        = { hipotesis induccion para r }
flatten' l (flatten' r ns)
```

Hemos obtenido 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)
```

De este ejemplo quisiéramos referirnos a la equivalencia en ambas definiciones, esto se puede describir como la igualdad de resultados al aplicar cualquiera de ellas al mismo árbol:

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:

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

Estructura de datos Utilizando la formalización de listas polimórficas en lógica de predicados, queremos demostrar 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, donde los elementos de la lista son de cierto tipo A

1. Funcionamente:

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

2. Relacionalmente:

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

Nuevamente para este ejemplo se requiere de un modelo dado por la formalización vista en la nota de especificación formal.

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 y que presentamos a continuación.

1.1. Reglas y sistemas de deducción

Para describir las reglas que emplearemos en la Lógica ecuacional y que permiten *encadenar* razonamientos respecto a ecuaciones, utilizaremos una representación que relaciona dos 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:

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

En sentido de arriba hacia abajo, la regla de deducción anterior se puede leer como sigue: si sucede que se cumplen $A_1, A_2, ..., A_m$ entonces podemos concluir B. En sentido contrario (de abajo hacia arriba), la regla se puede interpretar como sigue: para demostrar B basta con demostrar que se cumplen $A_1, A_2, ..., A_n$.

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 $\mathcal{T}$.*

Definición 2 *Una deducción o derivación de un enunciado $\mathcal{J}$ usando un sistema de deducción, 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. También puede usarse una representación de árbol de derivación donde la raíz está en la parte inferior y es el enunciado $\mathcal{J}$ y cada hoja del árbol es una instancia de un axioma.*

Los enunciados sobre ecuaciones que deseamos representar dependen de una serie de ecuaciones o de información preestablecida, para ello usaremos de manera **implícita** esta "base de datos" en cada derivación.

2. Lógica ecuacional

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

$$\frac{}{H}\ \textsc{Hyp} \qquad \frac{}{t = t}\ \textsc{Refl} \qquad \frac{s = t}{t = s}\ \textsc{Sym} \qquad \frac{t = r \qquad r = s}{t = s}\ \textsc{Trans}$$

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

El axioma básico es el que permite obtener cierta información conocida o "información de la base de datos". El resto de las reglas hacen énfasis en ciertas ecuaciones y

1. establecen las características de una relación simétrica y transitiva (Refl, Sym, Trans)

2. abstraen patrones por medio de instanciación (INST)

3. construyen nuevos términos por medio de congruencia entre subtérminos (CONGR)

Estas reglas son suficientes para hacer demostraciones pero se pueden añadir algunas otras reglas que factoricen aplicaciones de varias reglas asegurando una demostración o derivación con menos pasos, de hecho se pueden utilizar las siguientes en lugar de las reglas de INST y CONGR:

$$\frac{t = s \qquad E[x := s\sigma]}{t = sE[x := t\sigma]}\ \textsc{Rewrite} \qquad\qquad \frac{t = s \qquad E[x := t\sigma]}{E[x := s\sigma]}\ \textsc{Rewrite <-}$$

En las reglas de reescritura $E[t]$ denota a una ecuación haciendo explícita una presencia de algún término t, por ejemplo si $E =_{def} x + 2 = f\, y$ podemos hacer explícitas cualquiera de las subexpresiones involucradas en E, usando $E[x], E[x + 2], E[f\, y]$.

3. Formalización de teorías

El sistema de lógica ecuacional presentado ayuda a realizar demostraciones de ecuaciones de forma clara. Así una derivación finita utilizará unicamente estas reglas y si se desea realizar un árbol de derivación, entonces las hojas serán instancias de los axiomas HYP o REFL. 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 [Estructura algebraica] Considera el siguiente conjunto de ecuaciones $\mathcal{A}$, que define la estructura de anillo [2]:

$$\begin{array}{rcllrcll}
0 + x & = & x & A1 & x^{-1} + x & = & 0 & A2 \\
x + y & = & y + x & A3 & (x + y) + z & = & x + (y + z) & A4 \\
x \cdot 1 & = & x & A5 & (x \cdot y) \cdot z & = & x \cdot (y \cdot z) & A6 \\
x \cdot (y + z) & = & x \cdot y + x \cdot z & A7 & (y + z) \cdot x & = & y \cdot x + z \cdot x & A8
\end{array}$$

Demuestra que $x + (y + z) = y + (x + z)$.
Recordemos que la base de datos es $\mathcal{A}$.

(1)	$(x + y) + z = x + (y + z)$	HYP A5
(2)	$x + (y + z) = (x + y) + z$	SYM (1)
(3)	$z = z$	REFL
(4)	$x + y = y + x$	HYP A4
(5)	$(x + y) + z = (y + x) + z$	CONGR + (4) (3)
(6)	$(y + x) + z = y + (x + z)$	INST (1)
(7)	$x + (y + z) = (y + x) + z$	TRANS (2)(5)
(8)	$x + (y + z) = y + (x + z)$	TRANS (7)(6)

[2] Un anillo es un conjunto R junto con dos operaciones $(R, +, \cdot)$, un elemento distinguido 0 y una operación unaria $^{-1}$.

Ejemplo 3.2 [Estructura de datos] Del ejemplo presentado al inicio:

> *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.*

consideremos aquí las listas de números naturales. La base de datos correspondiente para la prueba tiene la definición de listas así como las definiciones de funciones sobre listas.

Esta demostración puede realizarse considerando la hipótesis sobre la forma de la lista a saber $\ell \neq [\,]$ y donde el único caso a analizar es $\ell = n :: \ell'$. Exhibiremos la partición trivial donde $x = [\,]$:

- Demostrar que los elementos `x = []`, `z = n` y `y = l'` tales que `x++(z::y) = n::l'`, suponiendo que `l = n:: l'`

(**1**)	$x = [\,]$	Hyp
(**2**)	$z = n$	Hyp
(**3**)	$y = \ell'$	Hyp
(**4**)	$z :: y = n :: \ell'$	Congr :: (2)(3)
(**5**)	$z :: y = z :: y$	Refl
(**6**)	$x + +(z :: y) = [\,] + +(z :: y)$	Congr $+ +$ (5)(1)
(**7**)	$[\,] + +(z :: y) = z :: y$	Ins (`nil ++ x = x`)
(**8**)	$x + +(z :: y) = z :: y$	Trans (6)(7)
(**9**)	$x + +(z :: y) = n :: \ell'$	Trans (4)(8)

3.1. Guiar una demostración

Las demostraciones de los ejemplos **3.1** y **3.2** pueden resultar elaboradas y con algunos pasos que resultan obvios, por ejemplo los pasos en donde se presentan las hipótesis o el uso de reflexividad.

Además buscamos explotar nuestra herramienta de trabajo, la computadora. Para ello utilizaremos un asistente de pruebas que facilitará la formalización y las demostraciones sobre alguna teoría o sistema, por ejemplo en este caso nuestro sistema de razonamiento ecuacional.

Decimos que un asistente de pruebas es un sistema (software) que ayuda en el desarrollo de demostraciones formales mediante la interacción con el usuario. El asistente o demostrador puede resolver algunos pasos de las demostraciones pero el usuario siempre está ahí para guiar el proceso inclusive dar instrucciones para que se realice de manera casi automática.

El sistema de deducción $\mathcal{B}$ y algunos refinamientos del mismo resultan ser subconjuntos de tácticas del asistente de pruebas Coq [3]. Este asistente interactúa con el usuario mediante comandos y tácticas, estas últimas son funciones que ayudan a descomponer un enunciado a demostrar en partes que resultan ser más fáciles de demostrar y que una composición final de todas las subdemostraciones asegura la prueba del enunuciado original.

Las tácticas correspondientes a las reglas del sistema $\mathcal{B}$ son:

Regla	Táctica
Hyp	`assumption`
Refl	`reflexivity`
Sym	`symmetry`
Trans	`transitivity r`
Inst	`apply`
Congr	`rewrite` (n veces)
Rewrite	`rewrite`
Rewrite <-	`rewrite <-`

[3] `http://coq.inria.fr/`

El uso de estas tácticas sigue el sentido inverso (*backward reasoning*) en la construcción de demostraciones de Coq, es decir que las reglas del sistema $\mathcal{B}$ serán leidas y aplicadas de abajo hacia arriba. Por esta razón la regla de transitividad debe incluir el término `r` que permite pasar de `s` a `t`. También debe considerarse que no siempre es posible tener todas las ecuaciones de la base de datos en el contexto de prueba, así la aplicación de la táctica `assumption` será sustituida por las tácticas `apply` o `simpl`, la última realiza una simplificación de un término. Veamos otros ejemplos:

Ejemplo 3.3 [Propiedades de la concatenación de listas] Consideremos la implementación de listas en Coq y algunas funciones:

```
Require Import List.
Import ListNotations.

Section MyListSection.
Variable A : Type.

(* Recordatorio de las definiciones como aparecen en las bibliotecas de Coq
Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.

Definition app (A : Type) : list A -> list A -> list A :=
  fix app l m :=
  match l with
   | nil => m
   | a :: l1 => a :: app l1 m
  end.
Infix "++" := app (right associativity, at level 60) : list_scope.
*)
```

Para demostrar que la concatenación asociativa, agregaremos dos axiomas a la base de datos que describen la conmutatividad de la concatenación respecto a la lista vacía:

```
Axiom axapp_nil_l : forall l:list A, [] ++ l = l.
Axiom axapp_nil_r : forall l:list A, l ++ [] = l.
```

Y un teorema auxiliar:

```
Theorem aux_app_comm_cons :
   forall (x y:list A) (a:A), a :: (x ++ y) = (a :: x) ++ y.
Proof.
induction x.
(* l = [] *)
+ intros. rewrite axapp_nil_l. simpl app. reflexivity.
(* l = a :: l *)
+ intros. rewrite <- IHx. simpl app.
reflexivity.
Qed.
```

Así el teorema se enuncia y demuestra como sigue:

```
Theorem app_assoc :
   forall l m n:list A, l ++ m ++ n = (l ++ m) ++ n.
Proof.
induction l.
```

```
(* l = [] *)
+ intros. apply app_nil_l.
(* l = a :: l *)
+ intros. rewrite <- aux_app_comm_cons.
rewrite IHl.
reflexivity.
Qed.
```

Hay que observar que la táctica `intro` o `intros` hace que Coq incluya todas las hipótesis de un enunciado. De esta forma se agrega más información a la base de datos.

Ejemplo 3.4 Una formalización alternativa usando Coq del ejemplo **3.2** resulta en el siguiente código:

```
Theorem list_partition :
   forall l:list nat, l <> nil -> exists (z:nat) (x y:list nat), l=x++(z::y).
```

La base de datos correspondiente para la prueba tiene la definición de listas que aparece en el archivo `Coq.Init.Datatypes` y las definiciones de funciones sobre listas dadas en `Require Import List.`

Por otro lado, también se puede considerar una implementación de una función que calcule *todas* las particiones posibles para una lista dada:

```
Fixpoint aux_partition (n: nat) (l:list nat) : list (list nat*nat*list nat) :=
   match n with
   | 0 => nil
   | S n => ((firstn n l),(nth n l 0),(skipn (S n) l)) :: aux_partition n l
   end.

Definition partition (x:list nat) : list (list nat*nat*list nat) :=
   let l := length x in aux_partition l x.
```

Y después demostrar que esta implementación cumple con la especificación dada por la propiedad:

```
Theorem func_partition :
   forall (l:list nat) (z:nat) (x y:list nat),
   In (x,z,y) (partition l) <-> l = x++(z::y).
```

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 como el caso de la función `flatten`. Sin inducción no es posible realizar dichas demostraciones.

Pero este esquema de demostración es manejado de manera sofisticada por Coq facilitando la construcción interactiva de la prueba. Así también hacemos énfasis en que el sistema ecuacional estudiado en esta nota es un subsistema de tácticas que ofrece Coq y que el uso del asistente rebasa en gran medida al sistema $\mathcal{B}$ como lo muestra la demostración del teorema `list_partition`:

```
Proof.
intros.
case_eq l.
(* l=[] *)
+ intro. elim H. assumption.
(* l=n::l0 *)
+ intros. exists n. exists []. exists l0. simpl. reflexivity.
Qed.
```

En los siguientes temas nos serviremos de este poder para realizar demostraciones asistidas por computadora.