Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
Laboratorio de Lenguajes de Programación
Abril-Julio 2016

Proyecto 1: Implementación de un asistente de pruebas para la lógica proposicional

1. Introducción

Se requiere que usted implemente usando lenguaje de programación Haskell, un asistente de prueba para los estudiantes del curso de lógica simbólica de la carrera de Ingeniería de Computación. Un asistente de prueba es una aplicación que le permite al usuario verificar si en una demostración cada inferencia realizada es válida.

Existen en la actualidad asistentes de pruebas bastantes sofisticados, como es el caso de Isabelle/HOL [1] y coq [2]. En nuestro caso implementaremos un asistente bastante sencillo, que abarque sólo el capítulo 3 de lógica proposicional, del libro del curso de Lógica simbólica usado en esta universidad [3].

2. Marco Teórico

La lógica que trata el libro de Gries [3] se denomina lógica ecuacional. Las reglas de inferencia del sistema son: la reflexividad, simetría, transitividad, instanciación y regla de Leibniz.

La reflexividad, simetría y transitividad son las propiedades clásicas de una relación de equivalencia, por lo que no se profundizará al respecto. La instanciación es la regla que dice que si se tiene la premisa $X \equiv Y$ y la sustitución $[x_1, \ldots, x_n := t_1, \ldots, t_n]$ entonces se puede concluir que
$X[x_1, \ldots, x_n := t_1, \ldots, t_n] \equiv Y[x_1, \ldots, x_n := t_1, \ldots, t_n]$, es decir:

$$\frac{X \equiv Y}{X[x_1, \ldots, x_n := t_1, \ldots, t_n] \equiv Y[x_1, \ldots, x_n := t_1, \ldots, t_n]}$$

La regla de Leibniz dice que si se tiene una función $g.z$ y $X \equiv Y$, entonces la aplicación a $g.z$ en X como en Y es equivalente, es decir:

$$\frac{X \equiv Y}{g.X \equiv g.Y}$$

Sin embargo si la función g es computada por la expresión E, esta regla es equivalente a

$$\frac{X \equiv Y}{E[z := X] \equiv E[z := Y]}$$

de modo que en notación lambda la función g puede denotarse como

$$\lambda\, z.E$$

y la regla luciría como

$$\frac{X \equiv Y}{(\lambda\, z.E)X \equiv (\lambda\, z.E)\, Y}$$

para este proyecto usaremos esta última notación.

Las reglas de instanciación y Leibniz se pueden abreviar a una regla de inferencia general que engloba a las dos. Esta regla general tiene la siguiente forma:

$$< \textit{statement } (\#) \textit{ with } (\textit{'instantiation'}) \textit{ using leibniz Rule with Exp} > \qquad (*)$$

Por ejemplo Si el enunciado 3.59 es $p \Rightarrow q \equiv \neg p \vee q$, entonces la regla

$$< \textit{statement } 3.59 \textit{ with } p, q := p \vee q, r \textit{ using } \lambda\, z.z \wedge (a \vee b) >$$

significa que se debe instanciar 3.59 con $[p, q := p \vee q, r]$ y la ecuación resultante $p \vee q \Rightarrow r \equiv \neg(p \vee q) \vee r$ debe usarse para aplicar la regla de Leibniz con la función $\lambda\, z.z \wedge (a \vee b)$, el resultado de esta regla general que engloba dos operaciones es la ecuación

$$(p \vee q \Rightarrow r) \wedge (a \vee b) \equiv (\neg(p \vee q) \vee r) \wedge (a \vee b).$$

Por la regla de simetría la ecuación resultante anterior puede usarse en sentido inverso

$$(\neg(p \vee q) \vee r) \wedge (a \vee b) \equiv (p \vee q \Rightarrow r) \wedge (a \vee b)$$

Para demostrar una equivalencia $Ex1 \equiv Ex2$ en esta lógica formal, partimos de $Ex1$ y usando la regla general $(*)$ junto con la simetría (de forma implícita para usar la ecuación resultante en el sentido conveniente), transformamos $Ex1$ hasta convertirlo en $Ex2$. Implícitamente usamos transitividad cuando aplicamos la regla $(*)$ secuencialmente, por lo que tenemos un mecanismo de inferencia donde sólo se dice explícitamente los números de los teoremas, la instanciación y la regla de Leibniz.

Por ejemplo si el enunciado 3.30 es $p \equiv p \vee \mathit{false}$ y el enunciado 3.59 es el de los párrafos anteriores, entonces lo siguiente es una inferencia del enunciado $q \wedge (p \Rightarrow \mathit{false}) \equiv q \wedge \neg p$.

$q \wedge (p \Rightarrow \mathit{false})$
$\equiv <$ statement 3.59 with $p := p$ using $\lambda\, z.q \wedge z >$
$q \wedge (\neg p \vee \mathit{false})$
$\equiv <$ statement 3.30 with $p := \neg p$ using $\lambda\, z.q \wedge z >$
$q \wedge \neg p$

3. Definición del lenguaje de proposiciones lógicas

Sea un conjunto de variables *Var* que constan de todas las letras del abecedario, definamos los términos válidos de nuestro lenguaje:

- *true* es un término válido.
- *false* es un término válido.
- Si $t \in \mathit{Var}$ entonces t es un término válido.
- Si t es un término válido, entonces $\neg t$ es un término válido.
- Si $t1$ y $t2$ son términos válidos, entonces $t1 \vee t2$, $t1 \wedge t2$, $t1 \Rightarrow t2$, $t1 \Leftrightarrow t2$, $t1 \not\Leftrightarrow t2$, son términos válidos.

El operador $\neg$ tiene la máxima precedencia, luego siguen los operadores $\vee$ y $\wedge$ que tienen la misma precedencia, seguido de $\Rightarrow$ y de último $\Leftrightarrow$ y $\not\Leftrightarrow$ que tienen la misma precedencia. La asociación del operador $\Rightarrow$ es por defecto a la derecha, en cambio todos los demás operadores asocian por defecto a la izquierda

A diferencia de lo que propone Gries en su libro, la expresión $t1 \equiv t2$ no la consideraremos un término, sino un objeto de tipo *Equation* que se construye en base a dos términos. De esta forma una fórmula como

$$t1 \equiv t2 \equiv t3 \qquad (**)$$

no es válida en este proyecto ya que si asociamos a la derecha de la forma

$$t1 \equiv (t2 \equiv t3), \quad (***)$$

entonces $t2 \equiv t3$ no es de tipo término y no puede ser pasado como argumento al operador $\equiv$ de la izquierda, análogamente el término $(**)$ tampoco sería válido si asociamos a la izquierda. Por esta razón, en una ecuación de tipo *Equation* sólo ocurre el símbolo $\equiv$ una sola vez, de modo que si se quiere expresar $(***)$, entonces en este proyecto escribiremos

$$t1 \equiv t2 \Leftrightarrow t3$$

y si se quiere expresar

$$(t1 \equiv t2) \equiv t3$$

escribiremos

$$t1 \Leftrightarrow t2 \equiv t3.$$

El símbolo $\equiv$ se usará para representar la equivalencia en la metateoría, es decir, se usará a la hora de hacer una inferencia del estilo

*termino*1
$\equiv$ <Explicación sobre las reglas de inferencia usadas para inferir>
*termino*2

De modo que en los términos *termino*1 y *termino*2, no puede ocurrir el símbolo $\equiv$, sino más bien en su lugar, se usará $\Leftrightarrow$. El símbolo $\equiv$ se usará también a la hora de enunciar los teoremas, ya que todos serán de la forma $t1 \equiv t2$ (con la restricción de que no ocurre el símbolo $\equiv$, en los término $t1$ y $t2$).

4. Detalles de la implementación

4.1. Operadores

En Haskell usted debe implementar el conjunto de los términos válidos usando tipos recursivos, y sobre este tipo debe definir los operadores infijos, `\/`, `/\`, `==>`, `<==>`, `!<==>`, que se corresponderán a los operadores $\vee$, $\wedge$, $\Rightarrow$, $\Leftrightarrow$, $\not\Leftrightarrow$. Debe usar las reglas de asociación y precedencias descritas en la sección anterior. Adicionalmente, usted debe definir el operador infijo `===` que recibe como argumento dos términos $t1$ y $t2$ y devuelve un objeto de tipo *Equation*, este operador representará en Haskell a la equivalencia $\equiv$. La precedencia de `===` debe ser menor que la de todos los demás operadores.

4.2. Variables

Con la intención de que los enunciados de la lógica proposicional se escriban en Haskell lo más parecido posible a como lo hacemos en papel, usted definirá por cada letra τ del alfabeto (no incluimos en el alfabeto a ñ, á,é,í, ó, ú), una función constante de nombre τ que devuelve el objeto de tipo término, que representa a la variable de letra τ. Por ejemplo, si los términos de su implementación se denominan `Term` y el constructor para las variables es de tipo `Var Char`, entonces lo que se le está sugiriendo es que defina las funciones:

```
a :: Term
a = Var 'a'

b :: Term
b = Var 'b'

c :: Term
c = Var 'c'
```

```
d :: Term
d = Var 'd'
    :
    :
    :
```

Con estas definiciones podremos escribir en Haskell expresiones como `a\/b ==> c/\b`, de manera que esta última expresión es interpretada por el compilador, como el término $a \vee b \Rightarrow c \wedge b$ en el modelo concreto de su implementación (hecha con tipos recursivos).

4.3. True y False

Al igual como en la sección anterior, con motivo de que los enunciados de la lógica proposicional se escriban en Haskell lo más parecido posible a como se hace en el papel, usted definirá dos funciones constantes de nombres `true` y `false` que devuelven el término concreto de su implementación (hecha con tipos recursivos), que representa *Verdadero* y *Falso* respectivamente. De igual forma como se explicó en la sección anterior, con estas definiciones pueden escribirse en Haskell expresiones como `true\/b ==> c/\false` para que sean interpretadas como el término correspondiente en el modelo concreto de su implementación.

4.4. Sustitución

Se requiere que usted defina un tipo de dato *Sust* que represente a la sustitución $p := t1$, $p, q := t1, t2$ y $p, q, r := t1, t2, t3$ (sólo consideraremos hasta un máximo de tres sustituciones en paralelo) para cualquieras variables p, q, r y términos $t1$, $t2$ y $t3$. Un objeto de tipo *Sust* debe operar junto con un término al cual se le aplica la sustitución, de modo que usted programará una función llamada `sust` que recibe un término y un objeto de tipo *Sust* e implementa la sustitución en paralelo.

Para escribir en Haskell lo más parecido al papel, se definirá un operador infijo sobre términos `=:`, que devuelve un objeto de tipo *Sust*, esto es con el objetivo de que `t1=:p` represente a la sustitución $p := t1$ (el token `:=` es reservado en Haskell por lo que no puede usarse para definir un operador infijo, por eso usaremos `=:`). De igual manera debe programar también que la tupla en Haskell `(t1,t2 =: p, q)` de tipo $(Term, Sust, Term)$ y la tupla `(t1, t2, t3 =: p, q, r)` de tipo
$(Term, Term, Sust, Term, Term)$ sean de tipo *Sust* y correspondan a $p, q := t1, t2$ y $p, q, r := t1, t2, t3$ respectivamente.

4.5. Instanciación

Usted debe programar una función llamada `instantiate` que reciba un objeto de tipo *Equation*, uno de tipo *Sust* y devuelva una nueva ecuación *Equation*

con el lado izquierdo y derecho de la misma instanciada según la sustitución que se introdujo como argumento.

4.6. Regla de Leibniz

Usted debe programar una función llamada `leibniz` que dado una ecuación `t1===t2` de tipo *Equation*, un término E y una variable z (en el modelo concreto de su implementación de variable), devuelve una nueva ecuación de tipo *Equation*, resultante de aplicar la regla de Leibniz con la ecuación `t1===t2`, en la función $\lambda\, z.E$

4.7. Inferencia

Para poder realizar una inferencia, usted debe programar una función llamada `infer` que dado un número n, una ecuación de tipo *Equation*, una sustitución *sus*, una variable z y un término E, devuelva una nueva ecuación resultante de aplicar la regla de Leibniz

$$\frac{X \equiv Y}{(\lambda\, z.E)X \equiv (\lambda\, z.E)\, Y}$$

donde la premisa $X = Y$, es la ecuación resultante de aplicar instanciación (usando la función `instantiate`) en el teorema de número n del módulo de teoremas (ver siguiente sección), con la sustitución *sus*

4.8. Deducción de un paso

Por otro lado nos interesa hacer deducciones a partir de un término inicial *termino*1 y convertirlo en un término final *termino*2. Esta transformación la llamaremos deducción de un paso y tienen la forma de:

*termino*1
$\equiv$ <statement $'num'$ with $(t1, t2 =: p, q)$ using $\lambda\, z.E >$
*termino*2

Para esto usted debe programar una función llamada `step` que recibe el término *termino*1 y todos los argumentos de la función `infer` de la sección anterior, para devolver *termino*2. Dicha función ejecuta `infer` y compara en la ecuación resultante cual lado de la ecuación es igual a *termino*1, para así devolver como *termino*2 al otro lado de la ecuación. La función `step` debe devolver un mensaje de error, si ningún lado de la ecuación resultante de `infer` es igual a *termino*1.

Como se espera usar este programa para el curso de lógica simbólica, entonces la sintaxis para la demostración de los teoremas debe ser parecida al curso y lo menos parecido a Haskell posible, por lo tanto usted definirá una función llamada `statement` y otras funciones dummy de nombres `with`, `using`, `lambda`,

que no jugarán ningún papel en particular, sino sólo servirán de estética para nuestros usuarios. La idea es que la expresión

$$statement\ 'num'\ with\ 'sustitution'\ using\ lambda\ z\ ('E')$$

sea de tipo $Term- > IO\ Term$ y se comporte de forma que si recibe un término $'termino'$ ignore las funciones dummy `with`, `using` y `lambda`, y ejecute `step` usando el término $'termino'$, el teorema número $'num'$, la sustitución $'sustitution'$ y la regla de Leibniz con la función $\lambda\, z.'E'$, al mismo momento que imprime por consola el hint y el término resultante.

Por ejemplo si el teorema de número 3.1 es `(p <==> q) <==> r === p <==> (q <==> r)` entonces ejecutar

```
statement 3.1 with (p,q =: q,r) using lambda z (z <==> q)
```

con el argumento `((p <==> (p <==> q)) <==> q)`
imprime

```
=== <statement 3.1 with (p,q =: q,r) using lambda z.z <==> q>
((p <==> p) <==> q) <==> q
```

4.9. Ejecución de una demostración

Una demostración del teorema $t1 === t2$ se considerará como una composición de varias reglas de inferencias del tipo

$$statement\ 'num'\ with\ 'sustitution'\ using\ lambda\ z\ ('E'),$$

que dado el término $t1$ inicial, dicha composición lo transforma en el término $t2$.

El orden de deducción es importante, por lo que toda inferencia para demostrar $t1 === t2$, va del lado izquierdo de la ecuación al lado derecho.

Una demostración puede ejecutarse e ir imprimiéndose paso a paso si componemos las reglas de inferencias con el operador $>>=$. Por ejemplo

```
return true
>>=
statement 3.2 with (p =: p) using lambda z (z)
>>=
statement 3.1 with (p <==> q,p =: p,r) using lambda z (z)
>>=
statement 3.3 with (p =: p) using lambda z (z <==> p)
```

debe imprimir:

```
=== <statement 3.2 with p =: p using lambda z.z>
(p <==> q) <==> (q <==> p)
=== <statement 3.1 with (p <==> q,p =: p,r) using lambda z.z>
((p <==> q) <==> q) <==> p
=== <statement 3.3 with p =: p using lambda z.z <==> p>
p <==> p
```

En la ejecución de una regla de inferencia se debe verificar, que la regla puede aplicarse al término que recibe como argumento. Si la regla no puede aplicarse se debe dar un mensaje de error.

Por ejemplo la ejecución de

```
return true
>>=
statement 3.2 with (p =: p) using lambda z (z/\p)
```

Debe imprimir

```
=== <statement 3.2 with q =: p using lambda z.z/\p>
*** Exception: invalid inference rule
```

Adicionalmente si el número del enunciado no existe en el módulo de teoremas (ver siguiente sección), entonces debe imprimir un mensaje indicando que el enunciado no existe.

5. Módulo de teoremas

Para almacenar la colección de teoremas que se usarán como base para hacer inferencias, usted creará un módulo en Haskell denominado *Theorems.hs*, en donde definirá una única función $prop :: Float \rightarrow Equation$ que asocia un número a algún teorema.

Un ejemplo de archivo theorems.hs con sólo tres teoremas es el siguiente:

```
-------------------------------------------------------
-- Aqui van todos los import que sean necesarios    --
-------------------------------------------------------

prop :: Float -> Equation
prop num
  | num == 3.1 = (p <==> q) <==> r === p <==> (q <==> r)
  | num == 3.2 = (p <==> q) <==> (q <==> p) === true
  | num == 3.3 = (p <==> q) <==> q === p
  | otherwise  = error "The statement doesn't exists"
```

En las reglas de inferencia de tipo

$$statement\ 'num'\ with\ 'sustitution'\ using\ lambda\ z\ ('E'),$$

el valor de $'num'$ hace referencia a alguno de los números de los teoremas en este módulo, por lo que usted debe usar esta función `prop` para extraer la ecuación indexada con el número correspondiente a la hora de programar la función `infer` y la función `step`

6. Archivo de Teorema

Cada Teorema a demostrar tendrá un archivo propio y la notación será

verify = *let theorem* = *enunciado in*
proof theorem
>>=
*regla*1
>>=
*regla*2
>>=
⋮
>>=
reglaN
>>=
done theorem

Por ejemplo el contenido de un archivo de teorema *Theorem*3_4.*hs* pudiera ser:

```
-------------------------------------------------------
-- Aqui van todos los import que sean necesarios    --
-------------------------------------------------------

verify = let theorem = (true === ((p <==> p) <==> (q <==> q))) in
         proof theorem
         >>=
         statement 3.2 with (p =: p) using lambda z (z)
         >>=
         statement 3.1 with (p <==> q,p =: p,r) using lambda z (z)
         >>=
         statement 3.3 with (p =: p) using lambda z (z <==> p)
         >>=
         statement 3.3 with (p =: p) using lambda z (p <==> z)
         >>=
         statement 3.1 with (p <==> q,q =: q,r) using lambda z (z)
         >>=
         statement 3.1 with (p,q =: q,r) using lambda z (z <==> q)
         >>=
         statement 3.1 with (p <==> p,q =: p,r) using lambda z (z)
         >>=
         done theorem
```

`proof` es una función que recibe la ecuación del enunciado del teorema y devuelve el término del lado izquierdo después de imprimirlo por consola.

Después de ejecutar la última regla de inferencia, se ejecuta la función `done` que verifica si el término que recibe de la última regla es igual al lado derecho de la equivalencia en el enunciado del teorema, en caso positivo devolveremos un mensaje exitoso y en caso contrario un mensaje de fracaso.

Por ejemplo la corrida de la función `verify` en el ejemplo anterior debe imprimir lo siguiente:

```
prooving true === ((p <==> p) <==> (q <==> q))

true
===<statement 3.2 with (p =: p) using lambda z (z)>
(p <==> q) <==> (q <==> p)
===<statement 3.1 with (p <==> q,p =: p,r) using lambda z (z)>
((p <==> q) <==> q) <==> p
===<statement 3.3 with (p =: p) using lambda z (z <==> p)>
p <==> p
===<statement 3.3 with (p =: p) using lambda z (p <==> z)>
p <==> ((p <==> q) <==> q)
===<statement 3.1 with (p <==> q,q =: q,r) using lambda z (z)>
(p <==> (p <==> q)) <==> q
===<statement 3.1 with (p,q =: q,r) using lambda z (z <==> q)>
((p <==> p) <==> q) <==> q
===<statement 3.1 with (p <==> p,q =: p,r) using lambda z (z)>
(p <==> p) <==> (q <==> q)

proof successful
```

Note que es importante que usted imprima los términos con la parentización correcta y simplificando según las reglas de precedencias del lenguaje definido en la sección tres.

7. Condiciones de entrega

Si un programa no se ejecuta, el equipo tiene cero como nota del proyecto.

Se considerará para su evaluación los aspectos de modularidad y diseño del código Haskell.

El trabajo es por equipos de laboratorio. Debe entregar los códigos fuentes de sus programas, en un archivo comprimido llamado *Proyecto1-X-Y.tar.gz*, donde X y Y son los números de carnet de los integrantes del grupo. La entrega se realizará *antes* de la $1:00$ pm del $21-$ de Mayo de 2016. El no cumplimiento de algunos de los requerimientos podrá resultar en el rechazo de su entrega.

8. Referencias

[1] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant*

for Higher-Order-Logic. (2015)
http://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf.

[2] *The Coq Proof Assistant. Reference Manual* (2016)
https://coq.inria.fr/distrib/current/refman/

[3] D. Gries. *The Science of Programming.* New York , Springer, 1981.