UNIVERSIDAD DE GRANADA

Curry-Howard

Mario Román

24 de octubre de 2014

El objetivo de estos apuntes es poner de manifiesto el isomorfismo de Curry-Howard y presentar con él el asesor de demostraciones Coq. Al lector sólo se le requiere un conocimiento superficial de programación funcional.

PRELUDIO: CONJUNTOS Y PROPOSICIONES

Las proposiciones son conjuntos, y los conjuntos son proposiciones. Este isomorfismo es trivial para el que estudia matemáticas, pero es potente porque expresa que esas dos estructuras son esencialmente, la misma cosa.

$$
\begin{aligned}
x \in A & \qquad\qquad & Px \\
x \in A^c & & \neg Px \\
x \in A \cup B & & (Px) \vee (Qx) \\
x \in A \cap B & & (Px) \wedge (Qx) \\
A \subset B & & P \Rightarrow Q
\end{aligned}
$$

De hecho puede construirse el isomorfismo como:

$$
\begin{aligned}
f : \texttt{Prop} &\to \texttt{Set} \\
Px \quad &\mapsto \{\, x \mid Px \,\}
\end{aligned}
$$

$$
\begin{aligned}
f^{-1} : \texttt{Set} &\to \texttt{Prop} \\
A \quad &\mapsto (Px \Leftrightarrow x \in A)
\end{aligned}
$$

Paradoja de Russell	**Paradoja de Quine**
El conjunto de los conjuntos que no se contienen a sí mismos no se contiene a sí mismo. $$m = \{x \mid x \notin x\}$$ $$m \in m \quad (?)$$	Es falso precedido de sí mismo es falso precedido de sí mismo. $$Mx \equiv \neg xx$$ $$MM \quad (?)$$

Lo realmente interesante del isomorfismo es que preserva las propiedades profundas entre los dos objetos. Planteamos la paradoja de Russel en uno de los dos lados y obtenemos la paradoja de Quine en el otro. Esta correspondencia puede parecer trivial, pero permite intercambiar las dos formas de escribir esa misma idea.

1. El isomorfismo Curry-Howard

Los tipos son teoremas, los programas son demostraciones.

En Haskell es fácil encontrar, para tipos *arbitrarios*, funciones que tengan la forma general:

```
a -> (b -> a)
(a,b) -> a
a -> (a -> b) -> b
```

Mientras que es imposible encontrar funciones de tipos como:

```
a -> (a -> b)
a -> b
b -> (b,c)
```

En este contexto es natural preguntarse qué tipos están poblados (existen elementos de ese tipo) y cuales no. [1] La respuesta a esta pregunta la da el isomorfismo de Curry-Howard:

Existe un objeto del tipo T si y sólo si T, interpretado como proposición lógica, es cierto.

1.1. Implicatio

Comprobamos que si tomamos el operador `->` como la implicación lógica, obtenemos resultados coherentes:

```
-- Tautología
idem :: a -> a
idem x = x
```

[1] Realmente no es exactamente imposible poblar un tipo cualquiera: Haskell permite salirse un poco del isomorfismo usando funciones recursivas que no terminan o usando la constante `undefined`. Pero no consideraremos esos casos límites.

```haskell
-- Podemos aplicar el modus ponens
modusPonens :: a -> (a -> b) -> b
modusPonens x f = f x

-- Pero no podemos encontrar funciones de este tipo
nop :: a -> (a -> b)
nop x = ??
```

La belleza del isomorfismo queda patente al probar que el modus ponens no es ni más ni menos que la aplicación de funciones.

1.2. Conjunctio

Y comprobemos ahora que la conjunción lógica se corresponde con el par de tipos `(_,_)`.

```haskell
-- Introducción de conjunción
conjIntro :: a -> b -> (a,b)
conjIntro x y = (x,y)

-- Eliminación de conjunción
fst :: (a,b) -> a
snd :: (a,b) -> b
```

De hecho, el tipo `(a,b)` está poblado si y sólo si lo están `a` y `b`.

1.3. Disiunctio

Necesitamos un tipo que esté poblado si lo está `a` o lo está `b`. Es claro observar que el tipo `Either a b` cumple lo pedido.

```haskell
-- Introducción de la disyunción
disjIntro :: a -> Either a b
disjIntro x = Left x

-- Análisis por casos
either :: (a -> c) -> (b -> c) -> Either a b -> c
```

1.4. Falsum et Verum

Nos quedan por definir las proposiciones *Falso* y *Verdadero* y la negación de proposiciones. Aquí llegamos al punto en el que el Haskell no sirve exactamente para nuestro propósito. En Haskell no se puede definir la función vacía, que sería necesaria para la negación. [2]
Falso será un tipo sin constructores, y por tanto, no habitado. De él obtenemos la negación y `Verdadero` mediante implicaciones:

[2] Realemente hay una forma bastante lógica de seguir con estas definiciones. Pero no es la que luego usaremos en Coq: `http://www.haskell.org/haskellwiki/Curry-Howard-Lambek_correspondence`

```haskell
-- Falso
data Falsum

-- La función vacía permite llegar a cualquier tipo
-- pero la sintaxis de Haskell no permite definirla
exFalsoQuodlibet :: Falsum -> a

-- Negar algo es afirmar que implica falsedad
type Not a = a -> Falsum

-- Y el tipo Verdadero es, trivialmente
type Verum = Falsum -> Falsum
```

Nótese que aquí estamos usando las reglas lógicas:

$$\begin{array}{lrrr} \forall A: & (A \Rightarrow False) = & (\neg A \vee False) = & \neg A \\ \forall A: & False \Rightarrow False = & (\neg False \vee False) = & True \end{array}$$

1.5. El isomorfismo en Haskell

Los tipos ya definidos y habitados en Haskell (`Int, Bool, String`) no serían más que axiomas definidos previamente.

2. Deducción natural

En 1935, Gerhard Genzen publicó dos nuevas formulaciones de la lógica, siendo una de ellas la **deducción natural**. Junto a ellas estableció un método de simplificación de demostraciones, que servía para asegurarse que la demostración no daba vueltas innecesarias.
Para asegurarse de esto, estableció que en la demostración normalizada de una fórmula sólo podían aparecer sus subfórmulas. Por ejemplo, para demostrar $A \wedge B$, sólo podían usarse A, B y sus subexpresiones, nunca algo como $A \vee B$. Este método asegura además la consistencia. No existe forma de demostrar $\perp$, la proposición contradicción (False).

2.1. Lógica intuicionista

La lógica intuicionista se diferencia de la lógica clásica en que usa una noción constructivista de verdad. En particular, el enunciado $A \vee B$ sólo puede ser demostrado si se construye alguna prueba de A o de B. En consecuencia, la ley del tercio excluso $A \vee \neg A$, no puede demostrarse. Aun así, puede introducirse como axioma.
La lógica intuicionista es perfecta para ilustrar el isomorfismo, porque se corresponde con el cálculo lambda tipado.

El isomorfismo va más allá de las reglas formales. La evaluación de programas en el cálculo lambda se corresponde directamente con la simplificación y normalización de pruebas que usa el cálculo natural.

$$\frac{A \quad B}{A \,\&\, B}\ (\&\text{-I}) \qquad \frac{A::M \quad B::N}{A{\times}B::(M,N)}\ (\times\text{-I})$$

$$\frac{A \,\&\, B}{A}\ (\&\text{-}E_1) \qquad \frac{A{\times}B::(M,N)}{A::M}\ (\pi_1)$$

$$\frac{A \,\&\, B}{B}\ (\&\text{-}E_2) \qquad \frac{A{\times}B::(M,N)}{B::N}\ (\pi_2)$$

$$\frac{A \to B \quad A}{B}\ (\to\text{-}E) \qquad \frac{A{\to}B::\lambda M.N \quad A::M}{B::N}\ (\to\text{-}E)$$

$$\begin{array}{c}[A]^x\\ \vdots\\ \dfrac{B}{A{\to}B}\end{array}\ (\to\text{-}I^x) \qquad \begin{array}{c}[A::M]^x\\ \vdots\\ \dfrac{B::N}{A{\to}B :: \lambda M.N}\end{array}\ (\to\text{-}I^x)$$

3. Introducción a Coq

3.1. Instalación

Los paquetes a instalar para usar el asistente de demostraciones Coq y el IDE oficial que permite escribir interactivamente las demostraciones son:

```
sudo apt-get install coq coqide coq-theories
```

Una alternativa más potente y recomendada a CoqIde es ProofGeneral, que funciona como plugin para Emacs.
En ambos puede avanzarse línea a línea sobre las definiciones y demostraciones (cada línea en Coq termina en un punto), usando `C-c C-n` en Emacs y usando el comando de avance en CoqIde.

REFERENCIAS

[1] Benjamin Pierce et al., *Software Foundations.* University of Pennsylvania, 2014 (Version 3.1).

[2] Philip Walder, *Propositions as Types*. University of Edimburg,

[3] Morten Heine B. Sørensen, Paweł Urzyczyn, *Lectures on the Curry-Howard isomorphism.*