Hacia la formalización del razonamiento ecuacional sobre mónadas

Elisabet Lobo Vesga*

Mayo de 2013

Resumen

Una de las grandes ventajas de los lenguajes funcionales puros es que permiten ser razonados ecuacionalmente, de esta forma, se facilita la verificación de su corrección. Pero esta pureza impide los efectos computacionales necesarios para que este tipo de programas tengan interacción alguna, es por esto que las funciones monádicas, que se encargan de encapsular estos efectos y así conservar la pureza, representan una estructura importante dentro de los lenguajes funcionales. Sin embargo, debido a que las mónadas poseen una estructura imperativa, no se ha podido establecer un enfoque aceptado para razonar ecuacionalmente sobre éstas. Por tanto, se pretende formalizar (desde un punto de vista computacional) en Agda la propuesta realizada por Gibbons y Hinze en [7] minimizando así los errores en los pasos de cada una de las demostraciones. Así pues, se presentan los conceptos básicos que permitan comprender cómo se razona sobre los programas y se muestra mediante un ejemplo la posibilidad de formalizar este razonamiento.

Palabras claves: *Razonamiento ecuacional, mónadas, formalización, asistente de pruebas.*

1 Introducción

Los lenguajes funcionales son llamados *puros* cuando cumplen con la propiedad de transparencia referencial:

> "Una misma expresión denota siempre el mismo valor, sea cual sea el punto del programa en que aparezca" [2].

Gracias a esta pureza se permite realizar manipulaciones algebráicas sobre los programas y la sustitución de iguales por iguales. Este proceso es conocido como razonamiento ecuacional y es desarrollado tal y como se hace en matemática, de esta manera es posible verificar más fácilmente la corrección de los programas, transformar programas en otros equivalentes más eficientes y realizar su debida depuración.

No obstante, esta pureza limita la interacción con el usuario, con datos externos y con otros programas debido a que excluye los *side-effects* (efectos de cómputo ó efectos secundarios) tales como modificación de variables, modificación de algún argumento, lanzamiento de excepciones, lectura y escritura de datos desde archivos o consola, indeterminismo (posibilidad de retornar múltiples valores en paralelo), entre otros. Por consiguiente, se tendrían programas que se puede probar que responden exactamente a lo especificado (son correctos) pero no podrían ser ejecutados. Sin embargo, a principios de los 90's Moggi[4] y Walder[10], muestran cómo las *mónadas* dan solución a éste problema encapsulando los *side-effects* logrando así conservar un ambiente de pureza mientras se computan este tipo de efectos.

Sin embargo, las funciones monádicas tienen una estructura imperativa y de esta manera, el razonamiento ecuacional que anteriormente se efectuaba de manera sencilla y natural, se dificulta sobre los programas que contienen este tipo de funciones. De allí surge la necesidad de plantear un enfoque aceptado para razonar ecuacionalmente sobre programas con funciones monádicas y de esta manera "recuperar" las bondades que brinda la programación funcional.

Aunque son pocos los trabajos realizados sobre éste tema, se resalta la aproximación axiomática planteada por Gibbons y Hinze en [7], éste será el punto de partida sobre el cual se basará la presente investigación. A pesar de que se ha analizado e implementado en Haskell cada una de las pruebas y ejemplos enmarcados en dicho trabajo, tales demostraciones no han sido formalizadas (desde un punto de vista computacional) y por tanto se desconoce si existe algun error (humano) o limitación en aquel razonamiento ecuacional. Es por esto que se usará el asistente de pruebas Agda para garantizar el proceder de cada una de las demostraciones.

*Grupo en Lógica y Computación, Departamento de Ciencias Básicas, Universidad EAFIT, Medellín, Colombia, elobove@eafit.edu.co

Cabe resaltar que la formalización y verificación de programas en asistentes de prueba, tales como Agda, Coq, Isabelle/HOL, NuPRL, entre otros, tiene una muy larga tradición. Uno de los trabajos más reconocidos es la formalización que realizó T. Nipkow, en 1998, de las 100 primeras páginas del libro de Wiskel (*The Formal Semantics of Programming Languages in the theorem prover*) en Isabelle/HOL [12]. A partir del año 2000, la cantidad de formalizaciones a través de asistentes de prueba se incrementó significativamente, tanto así que en ocasiones son exigidas para dar validez a investigaciones realizadas. Así pues, J. C. Filliâtre en 2007 formaliza en Coq el clásico programa *find* [6]. Posteriormente W. Swierstra en el 2011 verifica en Agda el *Dutch national flag problem* [14] formulado por Dijkstra (1976). De esta manera, los problemas clásicos que se conocían en matemáticas, vuelven a ser estudiados y verificados descartando todo posible error humano en las pruebas realizadas varios años o siglos atrás. Finalmente, lo más reciente es la investigación de M. Stannett e I. Németi (2013) [9] donde se realiza la formalización de teorías físicas empleando el asistente de pruebas Isabelle, para esto, se fundamentan en las versiones que se han creado de la teoría de la relatividad basada únicamente en lógica de primer orden.

Así pues, en el presente artículo se realiza una introducción al tema donde se descompone cada uno de los aspectos involucrados tales como, el razonamiento ecuacional, el razonamiento sobre programas, las funciones monádicas y finalmente se empalman en la implementación en Agda del problema de las torres de Hanoi planteado en [7]. Debido a que el foco principal es el razonamiento ecuacional, la sección inicial será está, la cual dará pie para seguir con el desarrollo del tema.

2 Razonamiento ecuacional

En ciencias de la computación existen diferentes métodos formales para razonar sobre programas, uno de los más poderosos y sencillos es el razonamiento ecuacional, el cual se basa en la sustitución de términos equivalentes tal cómo se plantea en álgebra básica, es decir, dadas unas propiedades, demostrar determinada igualdad. Por ejemplo:

Propiedades:

$$\begin{aligned} xy &= yx \\ x+(y+z) &= (x+y)+z \\ x(y+z) &= xy+xz \\ (x+y)z &= xz+yz \end{aligned}$$

Demostrar:

$$(x+a)(x+b) = x^2 + (a+b)x + ab$$

Prueba:

$(x+a)(x+b)$
$=$ [[distributiva a la izquierda]]
$(x+a)x + (x+a)b$
$=$ [[distributiva a la derecha]]
$xx + ax + xb + ab$
$=$ [[elevando]]
$x^2 + ax + xb + ab$
$=$ [[conmutatividad]]
$x^2 + ax + bx + ab$
$=$ [[distributiva a la derecha]]
$x^2 + (a+b)x + ab$

Ahora bien, debido a que los programas que se implementan sobre lenguajes funcionales puros son en sí mismos funciones, es posible efectuar pruebas matemáticas sobre estos programas para su verificación formal. Esta relación fue expuesta por De Millo, Lipton y Perlis[11] en 1979, es su trabajo se presentan analogías entre los componentes de una demostración matemática y su equivalente en programación. Esta idea ha sido desarrollada por diferentes autores a través del tiempo[1], plasmando una idea concreta de las respectivas equivalencias. Una de las analogías más claras es la denominada correspondencia Curry-Howard[13] que plantea:

Matemática	Programación
Teorema	Tipo
Prueba	Programa
Verificación	Chequeo de tipos
Eliminación-cut	Computación

Es así como el razonamiento empleado anteriormente puede ser mapeado sobre programas funcionales puros. Para observar claramente la similitud que se presenta entre las demostraciones matemáticas y el razonamiento sobre programas, tomemos un programa realizado en Haskell[5]. La siguiente función toma una lista y la invierte:

$reverse :: [a] \to [a]$
$reverse\ [\,] = [\,]$
$reverse\ (x : xs) = reverse\ xs +\!\!+ [x]$

Demostrar:

$reverse\ [x] = [x]\ |$

Prueba:

$reverse\ [x]$
$=$ [[notación]]
$reverse\ (x : [\,])$

$=$ [[aplicando *reverse*]]
$reverse\ [\,] +\!\!+ [x]$
$=$ [[aplicando *reverse*]]
$[\,] +\!\!+ [x]$
$=$ [[aplicando $+\!\!+$]]
$[x]$

De igual forma es posible demostrar propiedades de los programas razonando inductivamente, por ejemplo, basados en la misma definición de *reverse*:

Demostrar:

$$reverse\ (reverse\ xs) = xs$$

Prueba:

Caso base:
$reverse\ (reverse\ [\,])$
$=$ [[aplicando *reverse* al interior]]
$reverse\ [\,]$
$=$ [[aplicando *reverse*]]
$[\,]$

Para el paso inductivo asumimos que $reverse\ (reverse\ xs) = xs$, por tanto demostraremos que $reverse\ (reverse\ (x : xs)) = x : xs$.

Paso inductivo:
$reverse\ (reverse\ (x : xs))$
$=$ [[aplicando *reverse* al interior]]
$reverse\ (reverse\ xs +\!\!+ [x])$
$=$ [[distribtividad]]
$reverse\ [x] +\!\!+ reverse\ (reverse\ xs)$
$=$ [[lista singular]]
$[x] +\!\!+ reverse\ (reverse\ xs)$
$=$ [[hipótesis inductiva]]
$[x] +\!\!+ xs$
$=$ [[aplicando $+\!\!+$]]
$x : xs$

3 Funciones monádicas en Agda

Cómo se mencionó anteriormente, las mónadas fueron introducidas por Moggi para estructurar las especificaciones de notaciones y posteriormente Walder mostró su adaptación a los lenguajes de programación, en particular, los funcionales.

Una mónada en Haskell es representada como un objeto de tipo $M\ a$ resultante de encapsular un cómputo de tipo a mediante un constructor genérico M. Las mónadas son entonces constructores de tipos que, dada una entrada, la convierten en un nuevo tipo. Además del constructor, la mónada cuenta con dos funciones principales, el (>>=) y el *return*. La primera función mejor conocida como *bind* es la encargada de modelar la composición secuencial de computaciones y la segunda función modela la identidad. Adicionalmente en Haskell, la clase Monad está compuesta de otras dos funciones por defecto que resultan útilies desde un punto de vista computacional. Luego la clase esta definida así:

```
class Monad m where
  return :: a → m a
  (>>=)  :: m a → (a → m b) → m b
  (>>)   :: m a → m b → m b
  fail   :: String → m a
    -- Mínimo a completar: return, bind
  mx >> my = mx >>= λ_ → my
  fail s   = error s
```

Sin embargo, para que el constructor de datos y las dos funciones principales (*bind* y *return*) constituyan una mónada, se deben satisfacer las siguientes leyes:

```
  -- Identidad a la izquierda
return x >>= f = f x
  -- Identidad a la derecha
mx >>= return = mx
  -- Asociatividad
(mx >>= f) >>= g = mx >>= (λx → f x >>= g)
```

Por tanto, una vez definido el constructor y las funciones *bind* y *return* que cumplan las propiedades anteriores, se ha definido una mónada. Cabe aclarar que al realizar una intancia de la clase *Monad* en Haskell, el compilador no comprueba el cumplimiento de tales propiedades, será entonces deber del programador crear correctamente las instancias deseadas.

Es allí, donde se puede incurrir en errores por parte del programador, éstos pueden minimizarse al implementar estas estructuras en asistentes de prueba, debido a que en ellas cada una de estas propiedades se debe cumplir al crear cada uno de los términos monádicos. Así pues, en Agda se plantean las mónadas como se muestra en el *Anexo 1.*

Se puede observar que, al igual que en Haskell, en Agda una mónadar equiere de un constructor genérico M y esta estructura consta de las funciones *return* y *bind*, además de (>>). Además de esto, en Agda, al crear un término del *record Monad* se exige la demostración de las tres propiedades que, junto con los dos operadores principales y M, garantizan que el término creado es realmente una Mónada.

4 Razonando sobre funciones monádicas

Ahora bien, sabemos que aunque las mónadas son funciones con una estructura imperativa y se encargan de computar y encapsular los denominados *sidde-effecs*, éstas en ningún momento deterioran la pureza de los programas que las emplean. Es por esta razón que surge la pregunta ¿será posible razonar ecuacionalmente sobre este tipo de funciones?. Se debe tener en cuenta que en computación no se ha podido determinar la posibilidad de razonar ecuacionalmente sobre programas con efectos de cómputo, por lo que el poder responder afirmativamente al cuestionamiento formulado anteriormente supondrá un avance importante en el análisis matemático sobre este tipo de programas y por ende su debida depuración y verificación de la corrección.

Con todo esto, los trabajos que se han realizado enfocados en responder a ésta pregunta son escasos, sin embargo, se resalta la aproximación axiomática realizada por Gibbons y Hize [7] ya que preserva la abstracción mónadica. Basados en dicho trabajo, se realiza el siguiente análisis e implementación que da cuenta de lo cerca que se está al razoanmiento ecuacional sobre funciones monádicas.

4.1 Monad Count: Torres de Hanoi

Dada la siguiente implementación de *Monad Count* que se describe en *Anexo 2.* se plantean dos funciones las cuales representan la repetición finita de algún cómputo y la computación de cada uno de los movimientos que se deben realizar para solucionar el problema de las torres de Hanoi para n discos de la siguiente manera:

$$
\begin{aligned}
&rep : Nat \to M\ \ T \to M\ \ T\\
&rep\ zero\ mx = skip\\
&rep\ (suc\ n)\ mx = mx \gg rep\ n\ mx
\end{aligned}
$$

$$
\begin{aligned}
&hanoi : Nat \to M\ \ T\\
&hanoi\ zero = skip\\
&hanoi\ (suc\ n) = hanoi\ n \gg tick \gg hanoi\ n
\end{aligned}
$$

Sabiendo que

$$
\begin{aligned}
&skip : M\ \ T\\
&skip = return\ tt
\end{aligned}
$$

Se desea demostrar que la cantidad movimientos que se deben realizar para solucionar una torre de Hanoi de n discos es igual a $2^n - 1$, es decir, se desea demostrar que $hanoi\ n \equiv rep\ (2^n - 1)\ tick$.

Para realizar ésta demostración se parte de dos propiedades que se especifican sobre la función *rep* las cuales son:

$$
\begin{aligned}
&rep1 : (mx : M\ \ T) \to rep\ 1\ mx \equiv mx\\
&rep1 = unity - right
\end{aligned}
$$

$$
\begin{aligned}
&repMn : (m\ n : Nat) \to (mx : M\ \ T) \to\\
&\quad rep\ (m + n)\ mx \equiv (rep\ m\ mx \gg rep\ n\ mx)\\
&repMn\ zero\ \_\ mx = sym\ (unity - left\ tt)\\
&repMn\ (suc\ m)\ n\ mx =\\
&\quad begin\\
&\qquad bind\ (\lambda\_ \to rep\ (m + n)\ mx)\ mx\\
&\qquad\quad ==<\ cong\ f\ (repMn\ m\ n\ mx) >\\
&\qquad bind\ (\lambda\_ \to bind\ (\lambda\_ \to rep\ n\ mx)\ (rep\ m\ mx))\ mx\\
&\qquad\quad ==<\ sym\ (associativity\ mx) >\\
&\qquad (rep\ (suc\ m)\ mx \gg rep\ n\ mx)\\
&\quad qed\\
&\qquad \textbf{where}\ f = \lambda x \to bind\ (\lambda\_ \to x)\ mx
\end{aligned}
$$

De esta manera, dadas tales propiedades y el teorema algebráico enunciado en *Anexo 3.*, se procede a realizar la demostración deseada, la cual se halla en el *Anexo 4*. Finalmente, se logra demostrar por inducción sobre n que $hanoi\ n \equiv rep\ (2^n - 1)\ tick$

5 Conclusiones

Este trabjo fue inspirado, como ya se ha mencionado, en el trabajo realizado por Gibbons y Hinze[7] debido a que presenta el razonamiento sobre efectos de cómputo de una manera sencilla, clara y concisa. Así pues se ha podido evidenciar la gran utilidad que representan las funciones mónadicas en los lenguajes de porgramación puros, ya que conservan la *transparencia referencial* y a su vez permiten que los programas tengan "efectos secundarios". Además, se crea de manera simplificada el enlace entre lo que es el razonamiento ecuacional matemático y el razonamiento sobre programas funcionales.

Por otra parte, a través de la implementación en Agda del problema de las torres de Hanoi se abre la posibilidad de realizar el razonamiento ecuacional que hasta entonces, solo se planteaba y verificaba manualmente dando así lugar a posibles erros por parte del humano. Al utilizar un asistente de pruebas como Agda, se garantiza que este tipo de error son mínimos y de esta manera cada uno de los pasos realizados para la demostración se asumen correctos.

Hasta este punto, se puede concluir que la igualdad definida en la librería estándar de Agda, es útil para razonar ecuacionalmente sobre funciones

monádicas, aun así, queda abierta la investigación sobre otras mónadas que modelen otro tipo de efectos de cómputo (transformación de estado, no determinismo, probabilidades, entre otras) donde se pueda presentar alguna discrepancia.

Referencias

[1] A. Asperti, H. Geuvers y R. Natarajan. Social processes, program verification and all that. *Math. Struct. in Comp. Sience*. Cambrige University. vol.2, 2009.

[2] B. C. Ruiz, F. Gutierrez, P. Guerrero y J. E. Gallardo. Razonando con Haskell. Thomson, 2004.

[3] B. O'Sullivan, J. Goerzen y D. Stewart. Real World Haskell. O'Reilly, 2008.

[4] E. Moggi. Notions of computation and monads. *Information And Computation*, 93(1), 1991.

[5] G. Hutton. Programming in Haskell. Illustrated. *Cambridge University Press*, 2007.

[6] J. Christophe Filliâtre. Formal Proof of a Program: Find. *Science of Computer Programming*. 64, 2007.

[7] J. Gibbons y R. Hinze. Just do it: Simple Monadic Equational Reasoning. ICFP, 2011.

[8] M. Vanier. Mike's world-o-programming: Yet another monad tutorial. `http://mvanier.livejournal.com/3917.html`, Octubre 2012.

[9] M. Stannett, I. Németi. Using Isabelle to verify special relativity, with application to hypercomputation theory. 2013.

[10] P. Wadler. Monads for functional programming. M. Broy, editor, *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*, 1992.

[11] R. A. De Millo, R. J. Lipton y A. J. Perlis. Social processes and proofs of theorems and programs. ACM, 22(5), 1979

[12] T. Nipkow. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. *Formal Aspects of Computing*, 1998.

[13] W. A. Howard. The formulae-as-types notion of construction. *Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press,1980.

[14] W. Swierstra. FUNCTIONAL PEARL Sorted Verifying the Problem of the Dutch National Flag in Agda. Journal of Functional Programming, *Cambridge University Press*, 21(6),2011.

Anexos

Anexo 1. Implementación de mónada en Agda

$$
\begin{array}{l}
\mathit{record}\ \mathit{Monad}\ (M : \mathit{Set} \to \mathit{Set}) : \mathit{Set1}\ \mathbf{where} \\
\quad \mathit{constructor}\ \mathit{mkMonad} \\
\quad \mathit{field} \\
\qquad \mathit{return} \quad : \{A : \mathit{Set}\} \to A \to M\ A \\
\qquad \mathit{bind} \quad : \{A\ B : \mathit{Set}\} \to (A \to M\ B) \to M\ A \to M\ B \\
\qquad \mathit{associativity} \quad : \{A\ B\ C : \mathit{Set}\}\ \{f : A \to M\ B\}\ \{g : B \to M\ C\}\ (mx : M\ A) \to \\
\qquad\qquad \mathit{bind}\ g\ (\mathit{bind}\ f\ mx) \equiv \mathit{bind}\ (\mathit{bind}\ g \circ f)\ mx \\
\qquad \mathit{unity} - \mathit{left} \quad : \{A\ B : \mathit{Set}\}\ \{f : A \to M\ B\}\ (x : A) \to \\
\qquad\qquad \mathit{bind}\ f\ (\mathit{return}\ x) \equiv f\ x \\
\qquad \mathit{unity} - \mathit{right} : \{A : \mathit{Set}\}\ (mx : M\ A) \to \mathit{bind}\ \mathit{return}\ mx \equiv mx \\
\qquad \mathbf{infixl}\ 1\ \_ \gg\!= \_ \\
\qquad \_ \gg\!= \_ : \{A\ B : \mathit{Set}\} \to M\ A \to (A \to M\ B) \to M\ B \\
\qquad mx \gg\!= f = \mathit{bind}\ f\ mx
\end{array}
$$

Anexo 2. Implementación de MonadCount en Agda

```
record MonadCount {M : Set → Set} (monad : Monad M) : Set1 where
  constructor mkMonadCount
  open Monad monad
  field
    tick : M T
```

Anexo 5. Teorema algebráico

```
thm : (Nat : n) → ((2 ↑ n) − 1) + 1 + ((2 ↑ n) − 1) ≡ 2 ↑ (n + 1) − 1
```

Anexo 4. Demostración

```
test : (n : Nat) → hanoi n ≡ rep ((2 ↑ n) − 1) tick
test zero = refl
test (suc n) =
  begin
    (hanoi n ≫ tick ≫ hanoi n)
      ==< cong f (test n) >
    (rep ((2 ↑ n) − 1) tick ≫ tick ≫ rep ((2 ↑ n) − 1) tick)
      ==< cong g (sym (rep1 tick)) >
    (rep ((2 ↑ n) − 1) tick ≫ rep 1 tick ≫ rep ((2 ↑ n) − 1) tick)
      ==< cong (λx → x ≫ r) (sym (repMn ((2 ↑ n) − 1) 1 tick)) >
    (rep (((2 ↑ n) − 1) + 1) tick ≫ rep ((2 ↑ n) − 1) tick)
      ==< sym (rep − mn (((2 ↑ n) − 1) + 1) ((2 ↑ n) − 1) tick) >
    rep (((2 ↑ n) − 1) + 1 + ((2 ↑ n) − 1)) tick
      ==< cong (λx → rep x tick) (thm n) >
    rep ((2 ↑ (n + 1)) − 1) tick
      ==< cong (λx → rep ((2 ↑ x) − 1) tick) (sym (succ n)) >
    rep ((2 ↑ (suc n)) − 1) tick
  qed
    where f = λx → x ≫ tick ≫ x
      r = rep ((2 ↑ n) − 1) tick
      g = λx → r ≫ x ≫ r
```