Teoría de categorías aplicada a la programación funcional con Agda

Juan Pedro Villa-Isaza

2011-2, Universidad EAFIT, Medellín, Colombia (e-mail: jvillais@eafit.edu.co).

Resumen: *Teoría de categorías aplicada a la programación funcional con Agda* es una implementación de algunos conceptos de la programación funcional que están basados o han sido tomados de la teoría de categorías (categorías, functores, functores aplicativos, mónadas...). En este artículo se analizan las diferencias entre la definición formal de un functor y su definición en el lenguaje de programación Haskell, y se implementa un functor en el asistente de pruebas Agda que soluciona dichas diferencias.

Keywords: Teoría de categorías, programación funcional, tipo dependiente, asistente de pruebas, functor.

1. INTRODUCCIÓN

Teoría de categorías aplicada a la programación funcional con Agda (*Abel*) es una implementación de varios conceptos de la programación funcional que están inspirados o han sido tomados de la teoría de categorías. Aunque el proyecto implementa varios conceptos, este artículo sólo analiza el caso de los functores porque el esquema es similar para todos los conceptos.

En la sección 2 se presenta un resumen del proyecto (marco teórico y planteamiento, justificación y objetivos) como se propuso al inicio del mismo. En la sección 3 se explican algunas definiciones básicas de teoría de categorías y varias referencias de los distintos temas que trata el proyecto (teoría de categorías, programación funcional con Haskell y Agda, y las aplicaciones de la teoría de categorías a la programación).

En las secciones 4 y 5 se expone un ejemplo (functores) que muestra claramente cuál fue el problema resuelto por este proyecto y algunos de los resultados que hacen parte de la librería desarrollada. Por último, en la sección 6 se plantean las conclusiones del proyecto y algunas ideas a futuro, y en el apéndice A, algunos detalles del contenido de la librería.

La librería desarrollada se puede descargar de

`https://github.com/jpvillaisaza/Abel`.

Para más documentación, en

`https://github.com/jpvillaisaza/Cacao`

se encuentran algunas presentaciones elaboradas durante el desarrollo del proyecto y este artículo.

2. PROYECTO

Se le da el nombre de *teoría de categorías* a una rama relativamente reciente de las matemáticas que ocupa una posición central en las matemáticas contemporáneas y la informática teórica. Los objetos que estudia la teoría de categorías se denominan *categorías*. Básicamente, una categoría es un agregado o conjunto de *objetos* y «mapeos» (*morfismos* o *flechas*) entre objetos. El ejemplo más conocido de categoría es **Set**, la categoría que tiene los conjuntos como objetos y las funciones como morfismos. La definición de categoría es importante para presentar algunas nociones que convierten la teoría de categorías en una herramienta común que organiza y unifica muchos aspectos de la matemática (Marquis, 2011).

Algunos lenguajes de *programación funcional* (como Haskell y Agda) presentan ciertos conceptos importantes para relacionar la teoría de categorías con la programación. En este sentido, los tipos *functor*, *functor aplicativo* y *mónada* de Haskell y Agda, etcétera, son conceptos que pueden ser conceptualizados o que están basados en ideas provenientes de la teoría de categorías. Si bien su conceptualización categórica no es necesaria para su utilización, es un complemento ideal para comprender mejor la programación funcional y sus fundamentos teóricos (Haskell, 2011).

Teniendo en cuenta que es posible conceptualizar muchas nociones de la programación funcional por medio del estudio de la teoría de categorías, el proyecto de grado del autor (*Teoría de categorías aplicada a la programación funcional* (con Haskell)) plantea realizar un estudio monográfico de dicha rama de las matemáticas que permita describir y explicar algunas de sus aplicaciones a la programación funcional.

Como complemento a dicho proyecto, este tiene como objetivo programar una librería de Agda que implemente algunos conceptos de la programación funcional que están inspirados o han sido tomados de la teoría de categorías, y las leyes matemáticas que los rigen. Aunque el alcance del proyecto sólo incluye la implementación de cuatro conceptos (categorías, functores, functores aplicativos y mónadas), la librería implementa otros conceptos (functores monoidales y monoides) y contiene varios ejemplos de todos los constructos programados.

La diferencia entre este proyecto y el proyecto de grado del autor es la implementación de los conceptos y de las leyes matemáticas que los rigen. Dichas leyes no pueden ser implementadas en Haskell y no han sido implementadas en la librería estándar de Agda (Danielsson et ál., 2011).

3. PRELIMINARES

Aunque es posible entender el problema y la solución planteados por este artículo sin conocer las definiciones formales de los conceptos categóricos involucrados, es importante conocer las bases formales en las que está basada la implementación de los ejemplos tratados. Las siguientes definiciones están basadas en las de Mac Lane (1998) (la referencia estándar de la teoría de categorías).

Definición 1. Un *grafo* $\mathcal{G}$ es un conjunto de *objetos* $\mathcal{G}_{\mathrm{O}}$, un conjunto de *morfismos* $\mathcal{G}_{\mathrm{M}}$, y dos funciones

$$\mathrm{dom} : \mathcal{G}_{\mathrm{M}} \to \mathcal{G}_{\mathrm{O}} \quad \text{y} \quad \mathrm{cod} : \mathcal{G}_{\mathrm{M}} \to \mathcal{G}_{\mathrm{O}}$$

que asignan a cada morfismo $f : a \to b \in \mathcal{G}_{\mathrm{M}}$ un *dominio* $a = \mathrm{dom}(f) \in \mathcal{G}_{\mathrm{O}}$ y un *codominio* $b = \mathrm{cod}(f) \in \mathcal{G}_{\mathrm{O}}$.

Definición 2. Una *categoría* $\mathcal{C}$ es un grafo con una *función de identidad*

$$\mathrm{id} : \mathcal{C}_{\mathrm{O}} \to \mathcal{C}_{\mathrm{M}}$$

que asigna a cada objeto $a \in \mathcal{C}_{\mathrm{O}}$ un morfismo $\mathrm{id}_a : a \to a \in \mathcal{C}_{\mathrm{M}}$, y una *función de composición*

$$\circ\colon \mathcal{C}_{\mathrm{M}} \times \mathcal{C}_{\mathrm{M}} \to \mathcal{C}_{\mathrm{M}}$$

que asigna a cada par de morfismos componibles $g : b \to c \in \mathcal{C}_{\mathrm{M}}$ y $f : a \to b \in \mathcal{C}_{\mathrm{M}}$ un morfismo $g \circ f : a \to c \in \mathcal{C}_{\mathrm{M}}$, de manera que

$$\mathrm{id}_b \circ f = f \quad \text{y} \quad g \circ \mathrm{id}_b = g,$$

y, para $h : c \to d \in \mathcal{C}_{\mathrm{M}}$, que

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Definición 3. Para dos categorías $\mathcal{C}$ y $\mathcal{D}$, un *functor* $F : \mathcal{C} \to \mathcal{D}$ consta de una *función de objetos*

$$F_{\mathrm{O}} : \mathcal{C}_{\mathrm{O}} \to \mathcal{D}_{\mathrm{O}} \tag{1}$$

que asigna a cada objeto $a \in \mathcal{C}_{\mathrm{O}}$ un objeto $F_{\mathrm{O}}(a) \in \mathcal{D}_{\mathrm{O}}$, y de una *función de morfismos*

$$F_{\mathrm{M}} : \mathcal{C}_{\mathrm{M}} \to \mathcal{D}_{\mathrm{M}} \tag{2}$$

que asigna a cada morfismo $f : a \to b \in \mathcal{C}_{\mathrm{M}}$ un morfismo $F_{\mathrm{M}}(f) : F_{\mathrm{O}}(a) \to F_{\mathrm{O}}(b) \in \mathcal{D}_{\mathrm{M}}$, de manera que

$$F_{\mathrm{M}}(\mathrm{id}_a) = \mathrm{id}_{F_{\mathrm{O}}(a)} \tag{3}$$

y, siempre que el morfismo compuesto $g \circ f$ esté definido en $\mathcal{C}$, que

$$F_{\mathrm{M}}(g \circ f) = F_{\mathrm{M}}(g) \circ F_{\mathrm{M}}(f) \tag{4}$$

Haskell es un lenguaje de programación funcional. Lipovača (2011) y O'Sullivan et ál. (2008) son, en la opinión del autor, las mejores referencias de dicho lenguaje. Por su parte, Agda es un lenguaje de programación funcional con tipos dependientes y un asistente de pruebas. La página web de Agda (Agda, 2011) contiene varios tutoriales, entre los cuales es importante recomendar Norell (2009), y Bove y Dybjer (2008).

Por último, Elkins (2009) y Yorgey (2009) describen los conceptos de Haskell que están basados o han sido tomados de la teoría de categorías. Wadler (1989), Moggi (1991), y McBride y Paterson (2008) contienen análisis y bases formales de otros de los conceptos estudiados (*polimorfismo*, mónadas, y functores aplicativos y monoidales, respectivamente).

4. PROBLEMA

En Haskell, un (endo)functor representa cosas que pueden ser «mapeadas». En este sentido, puede pensarse en un functor como en un contenedor de elementos de algún tipo (como una lista). Los functores de Haskell están definidos así:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

`f :: * -> *` es un constructor de tipos y corresponde a la función de objetos F_{O} ((1)).

```
fmap :: (a -> b) -> f a -> f b
```

corresponde a la función de morfismos F_{M} ((2)) y es equivalente a

```
fmap :: (a -> b) -> (f a -> f b),
```

es decir, es una función que transforma funciones.

Los functores de Haskell sólo están definidos para una categoría, la categoría de los tipos de Haskell y las funciones entre dichos tipos. Dicha categoría recibe el nombre de **Hask**.

El tipo `Maybe` es uno de los ejemplos más sencillos de functores. El tipo de Haskell está definido así:

```
data Maybe a = Nothing | Just a
```

Por ejemplo, un valor de `Maybe Int` puede contener un número entero (`Just 3`) o no (`Nothing`). La siguiente instancia es la definición de `Maybe` como un functor:

```
instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap f (Just x) = Just (f x)
```

En pocas palabras, cuando hay un valor (`Just ...`), `fmap` aplica la función `f` a ese valor y retorna el resultado.

El problema de esta definición es que es responsabilidad del programador que una instancia cumpla les leyes que rigen los functores. Aunque todas las instancias (como la anterior) de Haskell cumplen las leyes, el lenguaje no requiere que esto se cumpla y es posible crear un functor que no lo sea. Un ejemplo es el tipo `CMaybe` (Lipovača, 2011):

```
data CMaybe a = CNothing | CJust Int a
```

Una instancia de la clase `Functor` para este tipo es:

```
instance Functor CMaybe where
  fmap _ CNothing    = CNothing
  fmap f (CJust n x) = CJust (n + 1) (f x)
```

Esta instancia es válida en Haskell, pero no cumple con ninguna de las leyes que debe cumplir un functor. Un contraejemplo es suficiente para demostrar esto: para `CJust 0 True`,

```
fmap id (CJust 0 True) = CJust 1 True,
```

pero

```
id (CJust 0 True) = CJust 0 True,
```

es decir, no se cumple que `fmap id = id`.

5. SOLUCIÓN

La solución planteada por este artículo exige un poco más de trabajo para el programador, pero garantiza que el código sea correcto y que cumpla con las leyes matemáticas que rigen los conceptos. El siguiente tipo es equivalente a la clase de tipos `Functor` de Haskell, pero también exige que las instancias demuestren que cumplen con las leyes que rigen los functores.

```
record Functor (F : Set → Set) : Set₁ where
  constructor
    mkFunctor

  field
    fmap : ∀ {A B} → (A → B) → F A → F B

    fmap-id : ∀ {A} (fx : F A) → fmap id fx ≡ id fx
    fmap-∘  : ∀ {A B C}
              {g : B → C} {f : A → B} (fx : F A) →
              fmap (g ∘ f) fx ≡ (fmap g ∘ fmap f) fx
```

`F` corresponde al constructor de tipos `f` de Haskell. Los campos del tipo contienen la función de morfismos `fmap` y dos leyes: `fmap-id` es (3) y `fmap-∘`, (4).

La siguiente definición es el tipo `Maybe` de Agda:

```
data Maybe (A : Set) : Set where
  just    : (x : A) → Maybe A
  nothing : Maybe A
```

Para definir `Maybe` como un functor, es necesario definir la función `fmap` y demostrar las dos leyes de la definición del tipo:

```
functor : Functor Maybe
functor = mkFunctor fmap fmap-id fmap-∘
  where
    fmap : ∀ {A B} → (A → B) → Maybe A → Maybe B
    fmap f (just x) = just (f x)
    fmap _ nothing  = nothing

    fmap-id : ∀ {A} (mx : Maybe A) → fmap id mx ≡ id mx
    fmap-id (just _) = refl
    fmap-id nothing  = refl

    fmap-∘ : ∀ {A B C}
             {g : B → C} {f : A → B} (mx : Maybe A) →
             fmap (g ∘ f) mx ≡ (fmap g ∘ fmap f) mx
    fmap-∘ (just _) = refl
    fmap-∘ nothing  = refl
```

En este caso, las demostraciones son muy simples, pero garantizan que `Maybe` en verdad es un functor, lo cual garantiza que su comportamiento como functor será el esperado.

Es posible definir el tipo `CMaybe` en Agda, así:

```
data CMaybe (A : Set) : Set where
  cjust    : (n : ℕ) (x : A) → CMaybe A
  cnothing : CMaybe A
```

Sin embargo, no es posible definir `CMaybe` como un functor. A continuación se presenta la instancia de `CMaybe` que Haskell aceptaba, pero en Agda no es posible demostrar las leyes (porque dicha definición no representa un functor).

```
functor : Functor CMaybe
functor = mkFunctor fmap fmap-id fmap-∘
  where
    fmap : ∀ {A B} → (A → B) → CMaybe A → CMaybe B
    fmap f (cjust n x) = cjust (n + 1) (f x)
    fmap _ cnothing    = cnothing

    fmap-id : ∀ {A} (mx : CMaybe A) → fmap id mx ≡ id mx
    fmap-id (cjust n x) = {!!}
    fmap-id cnothing    = refl

    fmap-∘ : ∀ {A B C}
             {g : B → C} {f : A → B} (mx : CMaybe A) →
             fmap (g ∘ f) mx ≡ (fmap g ∘ fmap f) mx
    fmap-∘ (cjust n x) = {!!}
    fmap-∘ cnothing    = refl
```

6. CONCLUSIONES

Como se mencionó anteriormente, el objetivo del proyecto se cumplió y se implementaron algunos conceptos adicionales (functores monoidales y monoides) y ejemplos para toda la librería. Es importante recordar que este proyecto es un complemento del proyecto de grado del autor y que su novedad está en la implementación de los conceptos categóricos estudiados y las leyes que los rigen (que no hacen parte de las librerías de Haskell ni de Agda). En este sentido, vale la pena tener en cuenta que existen otros proyectos de Agda que son implementaciones de la teoría de categorías y no formalizaciones categóricas de conceptos de la programación funcional. Además, aunque el estudio de los fundamentos teóricos de la programación no es requisito para ser un buen programador, este proyecto permite concluir que es un complemento ideal y también puede servir para reforzar el estudio de la teoría de categorías.

Los resultados de este proyecto se incluirán en el documento final del proyecto de grado mencionado (ya sea como un capítulo independiente o como complemento de otros capítulos) y, por lo tanto, el desarrollo de la librería continuará a pesar del fin del proyecto. Algunos aspectos para mejorar incluyen: revisar algunas leyes y restricciones (algunas leyes pueden ser demostradas de manera automática, algunas restricciones no son necesarias...); hacer que la librería sea más general (la librería estándar de Agda trabaja con distintos universos y la librería debe ser modificada si se propone como complemento) y más simple, e implementar más ejemplos de instancias de los conceptos que la componen.

REFERENCIAS

Agda (2011). `http://wiki.portal.chalmers.se/agda/`.

Bove, A. y Dybjer, P. (2008). Dependent types at work. En: L. Barbosa, A. Bove, A. Pardo y J.S. Pinto (eds.), *Language engineering and rigorous software development*, volumen 5520 de *Lecture notes in computer science*, págs. 57–99. Springer.

Danielsson, N.A. et ál. (2011). The Agda standard library, versión 0.5.

Elkins, D. (2009). Calculating monads with category theory. *The Monad.Reader*, 13, págs. 73–91.

Haskell (2011). `http://www.haskell.org/`.

Lipovača, M. (2011). *Learn you a Haskell for great good!: A beginner's guide.* No Starch Press.

Mac Lane, S. (1998). *Categories for the working mathematician*, volumen 5 de *Graduate texts in mathematics*. Springer, segunda edición.

Marquis, J.P. (2011). Category theory. En: E.N. Zalta (ed.), *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, edición Spring 2011.

McBride, C. y Paterson, R. (2008). Applicative programming with effects. *Journal of Functional Programming*, 18(1), págs. 1–13.

Moggi, E. (1991). Notions of computation and monads. *Information and Computation*, 93(1), págs. 55–92.

Norell, U. (2009). Dependently typed programming in Agda. En: P. Koopman, R. Plasmeijer y S.D. Swierstra (eds.), *Advanced functional programming*, volumen 5832 de *Lecture notes in computer science*, págs. 230–266.

O'Sullivan, B., Goerzen, J. y Stewart, D. (2008). *Real world Haskell*. O'Reilly.

Wadler, P. (1989). Theorems for free! En: *Functional programming languages and computer architecture*, págs. 347–359. ACM Press.

Yorgey, B. (2009). The typeclassopedia. *The Monad.Reader*, 13, págs. 17–68.

Apéndice A. ABEL

En la librería

`https://github.com/jpvillaisaza/Abel`

hay un módulo que contiene una breve descripción de todos los módulos que hacen parte de Abel:

```
module Abel.Everything where

-- Functores aplicativos
import Abel.Category.Applicative
-- Categorías
import Abel.Category.Category
-- Functores
import Abel.Category.Functor
-- Mónadas
import Abel.Category.Monad
-- Functores monoidales
import Abel.Category.Monoidal

-- El functor List
import Abel.Data.List.Functor

-- El functor aplicativo Maybe
import Abel.Data.Maybe.Applicative
-- El functor Maybe
import Abel.Data.Maybe.Functor
-- La mónada Maybe
import Abel.Data.Maybe.Monad
-- El functor monoidal Maybe
import Abel.Data.Maybe.Monoidal

-- El functor aplicativo (_×_ A)
import Abel.Data.Product.Applicative
-- El functor (_×_ A)
import Abel.Data.Product.Functor
-- La mónada (_×_ A)
import Abel.Data.Product.Monad
-- El functor monoidal (_×_ A)
import Abel.Data.Product.Monoidal

-- El functor aplicativo (_⊎_ A)
import Abel.Data.Sum.Applicative
-- El functor (_⊎_ A)
import Abel.Data.Sum.Functor
-- La mónada (_⊎_ A)
import Abel.Data.Sum.Monad
-- El functor monoidal (_⊎_ A)
import Abel.Data.Sum.Monoidal

-- Los tipos _⇒_ y _⇐_
import Abel.Function
-- Las categorías _⇒_ y _⇐_
import Abel.Function.Category

-- El functor aplicativo (_⇒_ A)
import Abel.Function.Applicative
-- El functor (_⇒_ A)
import Abel.Function.Functor
-- La mónada (_⇒_ A)
import Abel.Function.Monad
-- El functor monoidal (_⇒_ A)
import Abel.Function.Monoidal
```