Reimplementacin de *AspectAG* basada en nuevas extensiones de Haskell

Juan García Garland

May 31, 2018

Contents

1 Intro

2 *Type Level Programming*

2.1 Antes (`MultiparamTypeClasses, FunctionalDependencies`)

2.1.1 Typeclasses y Typeclasses Multiparametro

Haskell posee un sistema de *TypeClasses* originalmente pensado para proveer polimorfismo ad-hoc [1]. Una interpretación usual es que una *Typeclass* funciona como un predicado sobre tipos. Con la existencia de la extensión de GHC, `MultiParamTypeclasses` es posible programar type classes multiparametro. En realidad la limitación original a clases monoparámetro es algo arbitraria.

2.1.2 Dependencias Funcionales

Una solución a un problema que surge (cuando queremos typeclasses con tipos relacionados) fué tomada de las bases de datos relacionales [3] Tempranamente era sabido que el lenguaje a nivel de tipos es isomorfo al lenguaje a nivel de valores, y las fundeps de forma "a implies b" makes certain things able to resolve to a unique instance.
A fines del siglo pasado [2]. blah blah
———————-

2.1.3 Ejemplo: Naturales a Nivel de tipos

Considremos por ejemplo la siguiente implementación de los naturales unarios, como tipo inductivo:

```
data Nat = Zero
         | Succ Nat
```

Notar que esta definición introduce los constructores `Zero::Nat` y `Succ::Nat -> Nat`
Podemos entonces construir términos de tipo `Nat` de la forma

```
n0 = Zero
n4 = Succ $ Succ $ Succ $ Zero
```

O definir funciones de la siguiente manera:

```
add :: Nat -> Nat -> Nat
add n Zero     = n
add n (Succ m) = Succ (add n m)
```

Consideramos la definición a nivel de tipos:

```
data Zero
data Succ n
```

Nótese que introduce constructores `Zero::*` y `Succ::*->*` Análogamente podemos implementar la suma a nivel de tipos de la siguiente manera:

```
class Add m n smn | m n -> smn where
  tAdd :: m -> n -> smn

instance Add Zero m m
  where tAdd = undefined
```

```
instance Add n m k => Add (Succ n) m (Succ k)
  where tAdd = undefined
```

Ahora el término:

```
u3 = tAdd (undefined :: Succ (Succ Zero))(undefined :: Succ Zero)
```

tiene tipo `Succ (Succ (Succ Zero))`

Prolog vs TypeClasses Éste tipo de programación se asemeja a la programación lógica.
En Prolog[REF] escribiríamos:

```
add(0,X,X) :-
    nat(X).
add(s(X),Y,s(Z)) :-
    add(X,Y,Z).
```

Sin embargo, programar relaciones funcionales con Typeclasses difiere respecto a programar en Prolog, dado que el type checker de GHC no realiza backtracking al resolver una instancia.
Cuando tenemos una sentencia de la forma:

```
class (A x, B x) => C x
```

y GHC debe probar `C a`, primero se matchea la *cabeza* `C x`, agregando las restricción `x ã`, y **luego** tratando de probar el contexto. Si se falla habrá un error de compilación se abortará.
En Prolog es válido:

```
c(X) :- a(X), b(X)
c(X) :- d(X), e(X)
```

Si se trata de probar `c(X)` y fallan `a(X)` o `b(X)`, el intérprete hace *bactracking* y busca probar la alternativa.
En particular entonces no podemos decidir la implementación de los métdos(TODO usar termino correcto) de una clase a partir de la resolución de un contexto u otro. Esto sigue siendo relevante cuando programamos con las técniicas modernas y existe una solución sistemática que ilustraremos más adelante [REF]

2.1.4 Turing Completness

Con estas técnicas se pueden realizar computaciones sofisticadas en tiempo de compilación (como en [6]), y puede demostrarse que de hecho, que el sistema de tipos con estas extensiones tiene el poder de expresividad de un lenguaje Turing Completo, lo cual queda demostrado al codificar, por ejemplo un calculo de combinadores SKI [4].

2.2 Tipado a nivel de Tipos

Nótese que los constructores `Zero` y `Succ` tienen kinds `*` y `* → *`. Nada impide entonces construir instancias patológicas de tipos como `Succ Bool`, o `Succ (Succ (Maybe Int))`.
El lenguaje a nivel de tipos es entonces escencialmente *untyped*.
Una solución al problema de las instancias inválidas es programar un predicado (una nueva typeclass) que indique cuando un tipo representa un natural a nivel de tipos, y requerirla como contexto cada vez que se quiere asegurar que solo se puedan construir instancias válidas, así:

```
class TNat a
instance TNat Zero
instance TNat n => TNat (Succ n)
```

Por ejemplo la función add entonces puede definirse como:

```
class (TNat m, TNat n, TNat smn) => Add m n smn | m n -> smn where
  tAdd :: m -> n -> smn
```

2.2.1 Aplicaciones

La mayor utilidad de estas técnicas no pasa por realizar computaciones de propósito general en nivel de tipos, sino por codificar chequeos de propiedades que nuestro programa debe cumplir, como se hace usualmente con lenguajes de tipos dependientes aunque con algunas limitaciones, pero también con algunas ventajas.
La propia biblioteca AspectAG [7] o HList [5] (sobre la cual AspectAG hace uso intensivo) son buenos ejemplos de la utilidad de éste uso.
Para ejemplificar, consideremos un clásico ejemplo de tipo de datos dependiente: Las listas indizadas por su tamaño.

[TODO] Esto requiere GADTs, GADTs se introduce en ghc 6.8.1 igual que FunctionalDependencies Ademas el conte

```
{-TNat n => -}
data Vec a n where
  VZ :: Vec a Zero
  VS :: a -> Vec a n -> Vec a (Succ n)
```

Ejemplos:

```
safeHead :: (TNat n) =>  Vec a (Succ n) -> a
safeHead (VS a _) = a
```

```
<interactive>:3:10: error:
    - Couldn't match type 'Zero' with 'Succ n0'
      Expected type: Vec a (Succ n0)
        Actual type: Vec a Zero
    - In the first argument of 'safeHead', namely 'VZ'
      In the expression: safeHead VZ
      In an equation for 'it': it = safeHead VZ
```

TODO Ejemplo con HList?? para ver predicados sobre tipos lista con todos los tipos distintos por ej

3 Limitaciones

TODO

References

[1] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, March 1996.

[2] Thomas Hallgren. Fun with functional dependencies. In *Proc. of the Joint CS/CE Winter Meeting*, 2000.

[3] Mark P. Jones. Type classes with functional dependencies. In *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2000.

[4] Oleg Kiselyov. Type-level call-by-value lambda-calculator in three lines. http://okmij.org/ftp/Computation/lambda-calc.html. Accedido: 28-5-2018.

[5] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM.

[6] Conrad Parker. Type-Level Instant Insanity. *The Monad.Reader*, Issue Eight, September 2007.

[7] Marcos Viera, S. Doaitse Swierstra, and Wouter Swierstra. Attribute grammars fly first-class: How to do aspect oriented programming in haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 245–256, New York, NY, USA, 2009. ACM.