Reimplementación de *AspectAG* basada en nuevas extensiones de Haskell

Juan García Garland

October 27, 2018

Contents

1 Intro **3**

2 Programación a nivel de tipos **3**
- 2.1 Antes (`MultiparamTypeClasses, FunctionalDependencies, FlexibleInstances`) 3
  - 2.1.1 Typeclasses y Typeclasses Multiparametro . . . 3
  - 2.1.2 Dependencias Funcionales . . . 4
  - 2.1.3 Programación a nivel de tipos . . . 5
  - 2.1.4 Ejemplo: Naturales a Nivel de tipos . . . 5
  - 2.1.5 Poder computacional . . . 6
  - 2.1.6 Tipado a nivel de Tipos . . . 7
  - 2.1.7 Aplicaciones . . . 7
- 2.2 Ahora (`TypeFamilies, DataKinds, GADTs ...`) . . . 9
  - 2.2.1 DataKinds . . . 9
  - 2.2.2 TypeFamilies . . . 9
  - 2.2.3 Azucar sintáctica . . . 10
  - 2.2.4 Polimorfismo de Kinds . . . 11
  - 2.2.5 Programando con tipos dependientes . . . 11
  - 2.2.6 Limitaciones . . . 11
  - 2.2.7 Singletons y Proxies . . . 12
- 2.3 HList : Colecciones Heterogeneas Fuertemente tipadas . . . 14
  - 2.3.1 Listas Heterogeneas . . . 14
  - 2.3.2 Programando con restricciones . . . 18
  - 2.3.3 Manejo de Errores . . . 18
  - 2.3.4 Logic vs Functional . . . 19
- 2.4 Records Heterogeneos . . . 19
  - 2.4.1 Más Restricciones . . . 20

3 AspectAG **21**
- 3.1 Gramáticas de atributos . . . 21
- 3.2 Ejemplo: `repmin` . . . 21
- 3.3 AspectAG . . . 23

4 Reimplementacin de AAG **25**
- 4.1 Estructuras de Datos . . . 25
- 4.2 Declaraciones de Reglas . . . 26
- 4.3 Aspectos . . . 27
  - 4.3.1 Definifión . . . 27
  - 4.3.2 Combinación de Aspectos . . . 28
  - 4.3.3 Funciones semánticas . . . 29
  - 4.3.4 La función `knit` . . . 30

5 Comparación **31**

1 Intro

2 Programación a nivel de tipos

2.1 Antes (`MultiparamTypeClasses, FunctionalDependencies, FlexibleInstances`)

2.1.1 Typeclasses y Typeclasses Multiparametro

Haskell posee un sistema de *TypeClasses* originalmente pensado para proveer polimorfismo ad-hoc [2]. Una interpretación usual es que una *Typeclass* es un predicado sobre tipos. Cuando las *TypeClasses* fueron introducidas fueron consideradas una característica experimental, por lo que tuvieron un diseño conservador [11].
Las *Typeclasses* son una extensión al sistema de tipos de Hindley-Milner que es originalmente decidible. Para mantener la decibilidad en la extensión, en el diseño original del sistema de clases, se restringieron las instancias que es posible definir. En particular todas las declaraciones deben ser de la forma:

```
instance <Name> (...) => T a1 a2 ... an
```

en donde `a1 ... an` son variables de tipo distintas, y lo mismo vale para los contextos. En la práctica existen muchos casos de uso interesantes en que estas restricciones no permiten construir, y que usualmente no causan que la compulación diverja. Las extensiones de GHC `FlexibleInstances` y `FlexibleContexts` que se implementan a partir de su versión `6.8.1` eliminan algunas de estas restricciones y son ampliamente utilzadas.
Por otro lado, la limitación original a clases de un solo parámetro es arbitraria, y del mismo modo en que una clase monoparámetro puede interpretarse como un conjunto de tipos, podemos interpretar una clase multiparámetro como una relación entre tipos (un subconjunto del producto cartesiano, o un predicado binario). A partir de la versión `6.8.1` de GHC, se provee la extensión `MultiParamTypeclasses`, con la cual es posible programar typeclasses multiparámetro.
Existen múltiples usos de las mismas y no pretendemos ser exhaustivos en este documento. Uno inmediato es implementar relaciones como por ejemplo, el isomorfismo:

```
classs Iso a b where
  iso :: a -> b
  osi :: b -> a
```

En donde como es usual, las instancias que implementen la interfaz además del tipado (que será chequeado por el compilador) deben cumplir otras propiedades que serán responsabilidad del programador, en este caso que `iso.osi=id` y que `osi.iso=id`. En [11] se presentan múltiples ejemplos que ilustran la utilidad de las clases multiparámetro. Un caso de uso muy usual es la *sobrecarga con parámetros restringidos*, usado por ejemplo en la implementación de colecciones:

```
class Eq e => Collection c e where
  insert :: c -> e -> c
  member :: c -> e -> Bool
  ...
```

Los tipos `c` y `e` están relacionados en el sentido de que la colección (una estructura de tipo `c`) contiene elementos de tipo `e`. Por ejemplo, con listas podemos construir una implementación de colecciones:

```
instance Eq a => Collection [a] a where
  insert = flip (:)
  member = flip elem
  ...
```

2.1.2 Dependencias Funcionales

Supongamos que la clase `Collection` tiene además un método de tipo:

```
empty  :: c -> Bool
```

Obtenemos un error de compilación:

```
error:
   . Could not deduce (Collection c e0)
     from the context: Collection c e
       bound by the type signature for:
                  empty :: forall c e. Collection c e => c -> Bool
     The type variable 'e0' is ambiguous
```

Notar que si bien el tipo `e` está unívocamente determinado por `c` en cualquier instancia razonable, el compilador no puede deducir esto, por lo que en cada ocurrencia de `empty` el tipo `e` no puede determinarse y será ambigüo. La solución a este tipo de problemas fue tomada de las bases de datos relacionales [4]. Una dependencia funcional restringe las instancias de una clase multiparámetro. En una declaración como por ejemplo:

```
class (...) => C a b c | a -> b
```

Cada par de instancias de `C` que coincidan en el tipo concreto `a` **deben** coincidir en `b`, de lo contrario el compilador reportará un error. Con la extensión habilidasa el verificador de tipos se extiende de forma tal que una vez que se resuelva la ocurrencia de `a`, podrá resolverse la de `b` según la única posibilidad.
Así, por ejemplo la siguiente implementación de colecciones es legal, (y útil):

```
class Eq e => Collection c e | c -> e where
  insert :: c -> e -> c
  member :: c -> e -> Bool
  empty  :: c -> Bool
  ...

instance Eq a => Collection [a] a where
  insert = flip (:)
  member = flip elem
  empty  = null
  ...
```

2.1.3 Programación a nivel de tipos

Tempranamente era sabido que el lenguaje a nivel de tipos es isomorfo al lenguaje a nivel de valores, en el sentido de que la definición

```
data Zero
data Succ n
```

introduce constructores a nivel de tipos con aridad cero y uno, del mismo modo que la definición

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

los introduce a nivel de valores (con la salvedad de que a nivel de tipos, los constructores solo tienen en su *kind* información de la aridad; no están fuertemente tipados). Como se argumentó anteriormente la extensión de clases multiparámetro vino a eliminar una restricción de diseño, y las dependencias funcionales a resolver un problema con ellas. Pero la comunidad es creativa y los entusiastas no tardaron en explotar la posibilidad que proveen las extensiones de expresar computaciones en tiempo de compilación, abusando del sistema de tipos [3]. Las clases multiparametro definen relaciones sobre tipos, que combinadas con las dependencias funcionales permiten esencialmente expresar **funciones** sobre los mismos, y *decidir* una resolución por una dependencia funciona es esencialmente *computar* con él.

2.1.4 Ejemplo: Naturales a Nivel de tipos

Considremos la definición de la sección anterior del tipo Nat (la usual, con la primitiva `data`). Ésta 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 por *pattern matching* de la siguiente manera:

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

Por otra parte la definición a nivel de tipos:

```
data Zero
data Succ n
```

también introduce constructores (de tipos) `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))`, que es computado en tiempo de compilación gracias a la dependencia funcional.

Programación lógica y Programación con clases 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 verificador de tipos 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 el compilador *matchea* su objetivo con la *cabeza* `C x`, agregando las restricción `x` $\sim$ `a` a su ambiente, y luego busca probarse 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 una prueba de la alternativa. En haskell la traducción del programa anterior ni siquiera es legal (GHC retorna error por *Overlapping Instances*).
En particular entonces no podemos decidir la implementación de las operaciones de una clase a partir de la satisfacción de uno u otro contexto. Esto es una sana decisión de diseño (de lo contrario, por ejemplo, al intercambiar el órden de declaraciones puede cambiarse la semántica de un programa si ambas conducen a una prueba). Tanto utilizando clases con dependencias funcionales o las técnicas más modernas de programación a nivel de tipos que más adelante se presentarán, este comportamiento de la resolución de clases es relevante y condiciona el estilo de programación que utilizaremos.

2.1.5 Poder computacional

Con estas técnicas se pueden realizar computaciones sofisticadas en tiempo de compilación [10] [9], y puede demostrarse que de hecho, que las técnicas para definir computaciones en tiempo de compilación con estas extensiones tienen el poder de expresividad de un lenguaje Turing Completo, lo cual queda demostrado al codificar, por ejemplo un calculo de combinadores SKI [5].

2.1.6 Tipado a nivel de Tipos

En el ejemplo anterior 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 no tipado. Una solución al problema de las instancias inválidas es programar un predicado (una nueva clase) que indique cuándo 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
```

2.1.7 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 (en tiempo de compilación), como se hace usualmente con lenguajes de tipos dependientes aunque con algunas limitaciones, pero también con algunas ventajas. McBride [9] discute la aplicación usual de vectores (listas indizadas por su largo). Como ejemplos más complejos podemos citar a la biblioteca HList [6] de colecciones heterogeneas fuertemente tipadas, la propia biblioteca AspectAG [13] que vamos a reimplementar, o bases de datos fuertemente tipadas [12].
A modo de ejemplo consideremos el 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

```
data Vec a n where
  VZ :: Vec a Zero
  VS :: a -> Vec a n -> Vec a (Succ n)
```

Por ejemplo, la función:

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

aplicada a un vector vacío no compilará.

```
<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
```

La implementación de funciones no triviales sobre vectores (por ejemplo `reverse`, `take`, `chop`) son complejas y requieren desarrollar algunos *hacks*. Esencialmente con estas técnicas antiguas de programación a nivel de tipos tenemos que resolver todas las limitaciones presentadas en la sección 2.2.6, y más. La biblioteca HList [6] es una recopilación exhaustiva de estas técnicas. A los efectos de este documento no nos interesa profundizar en estos detalles.

2.2 Ahora (`TypeFamilies, DataKinds, GADTs ...`)

2.2.1 DataKinds

El desarrollo del concepto de datatype promotion [15], que se introduce en GHC en su versión `7.4.1` con la extensión `DataKinds` es un importante salto de expresividad. En lugar de tener un sistema de kinds en donde solamente se logra expresar la aridad de los tipos, pueden *promoverse* ciertos tipos de datos (con ciertas limitaciones, por ejemplo inicialmente no es posible promover `GADTs`). Con la extensión cada declaración de tipos como

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

se "duplica" a nivel de kinds. Esto es, además de introducir los términos `Zero` y `Succ` de tipo `Nat`, y al propio tipo `Nat` de kind `*` la declaración introduce los *tipos* `Zero` y `Succ` de Kind `Nat` (y al propio kind Nat). El kind `*` ya no es el único unario existente, y pasa a ser uno más, aunque particular: el de los tipos habitados. Cada vez que declaramos un tipo promovible se introducen tipos no habitados del nuevo kind promovido.
En el ejemplo de la sección anterior, el tipo `Vec` tenía kind `* -> * -> *` por lo que `Vec Bool Char` era un tipo válido. Con DataKinds podemos construir: (Utilizando además, la extensión `KindSignatures` para anotar el kind de `Vec`).

```
data Vec :: Nat -> * -> * where
  VZ :: Vec Zero a
  VS :: a -> Vec n a -> Vec (Succ n) a
```

Cuando programamos con la extensión habilitada por defecto se promueven algunos tipos básicos, en particular las listas. Por ejemplo `[Bool, Char, Int]` es un tipo de kind `[*]`, o `[Succ Zero, Zero]` es un tipo de kind `[Nat]`. Cuando existe ambigëdad (y en particular en ciertos contextos es necesario) los constructores promovidos pueden escribirse precedidos por un carácter apostrofe, por ejemplo `'[ 'Succ 'Zero, 'Zero]` es el tipo de kind `[Nat]` que antes..

2.2.2 TypeFamilies

(6.8.1)
Las *Type families* o familias de tipos indizadas, son una extensión para facilitar la programación a nivel de tipos. [1]

———————————————- [REFS: https://wiki.haskell.org/GHC/Typefamilies]
ndexed type families are a new GHC extension to facilitate type-level programming. Type families are a generalisation of associated data types (Associated Types with Class, M. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. In Proceedings of "The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05", pages 1-13, ACM Press, 2005) and associated type synonyms ("Type Associated Type Synonyms". M. Chakravarty, G. Keller, and S. Peyton Jones. In Proceedings of "The Tenth ACM SIGPLAN International Conference on Functional Programming", ACM Press, pages 241-253, 2005). Type families themselves are described in the paper "Type Checking with Open Type Functions", T. Schrijvers, S. Peyton-Jones, M. Chakravarty, and M. Sulzmann, in Proceedings of "ICFP 2008: The 13th ACM SIGPLAN International Conference on Functional Programming", ACM Press, pages 51-62, 2008.

[1]

Las type families proveen tipos indizados por tipos y funciones sobre los mismos que son útiles en programación genérica o interfaces provistas con información estática de tipado, del mismo modo que con interfaces de tipos dependientes. Pueden considerarse una alternativa a las dependencias funcionales, pero proveen un estilo más *idiomático* de programación en el contexto de haskell, dado que codificar familias de tipos se asemeja a la programación funcional en términos, a diferencia del enfoque relacional o lógico de las dependencias funcionales. La siguiente definición implementa la suma a nivel de tipos:

```
type family (m :: Nat) + (n :: Nat) :: Nat
type instance Zero + n = n
type instance Succ m  + n = Succ (m :+ n)
```

Es el equivalente de la función

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

definida a nivel de valores.
Entonces, el **tipo** `(Succ Zero) + (Succ Zero)` denota, y reduce a `Succ (Succ Zero)`.
Existe una notación alternativa, cerrada:

```
type family (m :: Nat) + (n :: Nat) :: Nat where
  (+) Z     a = a
  (+) (S a) b = S (a + b)
```

En ambos casos la anotación de los kinds podría omitirse.

2.2.3 Azucar sintáctica

En la definición anterior se utilizan algunas extensiones más para lograr una sintaxis tan limpia. La extensión `TypeOperators` (implementada desde GHC `6.8.1`) habilita el uso de simbolos operadores como constructores de tipos. Por ejemplo, podemos definir

```
data a + b = Left a | Right b
```

En la sección anterior definimos la type family `(+)`. Sin contar con la extensión `TypeOperators` deberíamos definir la familia de tipos como

```
type family Add m n ...
```

La extensión `KindSignatures` por otra parte permite anotar los kinds (del mismo modo que anotamos los tipos en funciones a nivel de valores), tanto para documentar como para desambiguar en ciertos casos.
Podemos ir más allá con la notación, el módulo `GHC.TypeLits` (`base-4.12.0.0`) declara Naturales y Caracteres a nivel de tipos con una sintaxis como la usual.
Importando el módulo, es código legal, por ejemplo:

```
data Vec :: Nat -> * -> * where
  E    :: Vec 0 a
  (:<) :: a -> Vec n a -> Vec (n+1) a
```

Y `'e':<'4':<E` una expresión de tipo `Vec 2 Char`

2.2.4 Polimorfismo de Kinds

Yorgey et al. [REF] introducen el polimorfismo a nivel de kinds, que en GHC corresponde a la extensión `PolyKinds` implementada a partir de la versión `7.4.1` del compilador. Con la extensión habilidada es posible implementar funciones a nivel de tipos polimórficas. Por ejemplo:

```
type family Length (list :: '[a]) :: Nat where
  Length '[]       = 'Zero
  Length (x ': xs) = 'Succ (Length xs)
```

2.2.5 Programando con tipos dependientes

[explicar mejor aca] Utilizando esta representación, podemos programar funciones seguras, de forma análoga a como las escribiríamos en lenguajes de tipos dependientes como por ejemplo:

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

Intentar compilar la expresión `vHead VZ` retorna un error de compilación:

```
error:
    . Couldn't match type  ''Zero' with ''Succ n0'
      Expected type: Vec ('Succ n0) a
        Actual type: Vec 'Zero a
    . In the first argument of 'vHead', namely 'VZ'
      In the expression: vHead VZ
```

Otros ejemplos, pueden ser:

```
vTail :: Vec (Succ n) a -> Vec n a
vTail (VS _ as) = as
```

```
vZipWith :: (a -> b -> c) -> Vec n a -> Vec n b -> Vec n c
vZipWith _ VZ VZ = VZ
vZipWith f (VS x xs) (VS y ys)
  = VS (f x y)(vZipWith f xs ys)
```

O incluso:

```
vAppend :: Vec n a -> Vec m a -> Vec (n :+ m) a
vAppend (VZ) bs      = bs
vAppend (VS a as) bs = VS a (vAppend as bs)
```

2.2.6 Limitaciones

A diferencia de lo que ocurre en implementaciones de lenguajes con tipos dependientes, los lenguajes de términos y de tipos en Haskell continúan habitando mundos separados. La correspondencia entre nuestra definición de vectores y las familias inductivas en los lenguajes de tipos dependientes no es tal.
Las ocurrencias de `n` los tipos de las funciones anteriores son estáticas, y borradas en tiempo de ejecución, mientras que en un lenguaje de tipos dependientes estos parámetros son esencialmente

dinámicos [8]. En las teorías de tipos intensionales una definición como la familia en 2.2.2 extiende el algoritmo de normalización, de forma tal que el compilador decidirá la igualdad de tipos según las formas normales. Si dos tipos tienen la misma forma normal entonces los mismos términos les habitarán. Por ejemplo, los tipos `Vec (S (S Z) :+ n) a` y `Vec (S (S n)) a` tendrán los mismos habitantes. Esto no va a ser cierto para tipos como `Vec (n :+ S (S Z)) a` y `Vec (S (S n)) a`, aunque que los tipos coincidan para todas las instancias concretas de `n`. Para expresar propiedades como la conmutatividad se utilizan evidencias de las ecuaciones utilizando *tipos de igualdad proposicional*[2]. [8]. [buscar mejor referencia]
En el sistema de tipos de Haskell sin embargo, la igualdad de tipos es puramente sintáctica. `Vec (n :+ S (S Z)) a` y `Vec (S (S n)) a` **no** son el mismo tipo, y no tienen los mismos habitantes. La definición como 2.2.2 axiomatiza `:+` para la igualdad proposicional de Haskell. Cada ocurrencia de `:+` debe estar soportada con evidencia explícita derivada de estos axiomas. Cuando GHC traduce desde el lenguaje externo al lenguaje del kernel, busca generar evidencia mediante heurísticas de resolución de restricciones.
La evidencia sugiere que el *constraint solver* computa agresivamente, y esta es la razón por la cual la función `vAppend` definida anteriormente compila y funciona correctamente.
Sin embargo, dada la función:

```
vchop :: Vec (m + n) x -> (Vec m x, Vec n x)
```

resulta imposible definirla si no tenemos la información de `m` o `n` en tiempo de ejecución (intuitivamente, ocurre que "no sabemos donde partir el vector").
Por otra parte la función:

```
vtake :: Vec (m + n) x -> Vec m x
```

tiene un problema más sutil. Incluso asuminedo que tuvieramos forma de obtener `m` en tiempo de ejecución, no es posible para el verificador de tipos aceptarla. No hay forma de deducir `n` a partir del tipo del tipo `m + n` sin la información de que `(+)` es una función inyectiva, lo cual el verificador de tipos es incapaz de deducir.

2.2.7 Singletons y Proxies

Existen dos *hacks*, para resolver los problemas planteados en la sección anterior.

Singletons Si pretendemos implementar `vChop` cuyo tipo podemos escribir más explícitamente como

```
vChop :: forall (m n :: Nat). Vec (m + n) x -> (Vec m x, Vec n x)
```

necesitamos hacer referencia explícita a `m` para decidir donde cortar. Como en Haskell el cuantificador $\forall$ dependiente solo habla de objetos estáticos (los lenguajes de tipos y términos están separados), esto no es posible directamente.
Un tipo *singleton*, en el contexto de Haskell, es un `GADT` que replica datos estáticos a nivel de términos.

```
data SNat :: Nat -> * where
  SZ :: SNat Zero
  SS :: SNat n -> SNat (Succ n)
```

[2]Propositional Types

Existe por cada tipo `n` de kind `Nat`, **un** [3] valor de tipo `SNat n`.
Podemos implementar `vChop`:

```
vChop :: SNat m -> Vec (m :+ n) x -> (Vec m x, Vec n x)
vChop SZ xs            = (VZ, xs)
vChop (SS m) (VS x xs) = let (ys, zs) = vChop m xs
                         in (VS x ys, zs)
```

Proxies Análogamente para definir `vTake` necesitamos `m` en tiempo de ejecución. Consideramos la implementación:

```
vTake :: SNat m -> Vec (m :+ n) x -> Vec m x
vTake SZ xs     = SZ
vTake (SS x xs) = -- (no es necesario para activar el error)
```

Obtenemos un error de tipos:

```
. Couldn't match type 'm :+ n' with 'm :+ n0'
  Expected type: SNat m -> Vec (m :+ n) x -> Vec m x
  Actual type: SNat m -> Vec (m :+ n0) x -> Vec m x
  NB: ':+' is a type function, and may not be injective
  The type variable 'n0' is ambiguous
. In the ambiguity check for 'vtake'
```

Notar que esta vez no es necesaria una representación de `n` en tiempo de ejecución. `n` es estático pero necesitamos que sea explícito.
Consideramos la definición:

```
data Proxy :: k -> * where
  Proxy :: Proxy a
```

Proxy es un tipo que no contiene datos, pero contiene un parámetro *phantom* de tipo arbitrario (de hecho, de kind arbitrario).
Historicamente, el constructor de tipos no polimórfico `Proxy :: * -> *` funcionaba como una alternativa segura a `undefined :: a`, como las usadas en la sección [REF] (usando expresiones como `Proxy :: Proxy a`).
Gracias al polimorfismo de kinds podemos construir proxys aplicando el constructor a habitantes de cualquier kind, en particular `Nat`. El uso de un proxy va a resolver el problema de `vTake`, indicando simplemente que la ocurrencia del proxy tiene la información del tipo `n` en el vector.
La siguiente implementación de vTake es funcional:

```
vTake :: SNat m -> Proxy n -> Vec (m :+ n) x -> Vec m x
vTake SZ _ xs           = VZ
vTake (SS m) n (VS x xs) = VS x (vTake m n xs)
```

[3] Formalmente esto no es cierto, si consideramos el valor $\perp$

2.3 HList : Colecciones Heterogeneas Fuertemente tipadas

2.3.1 Listas Heterogeneas

En [6] se presenta un buen ejemplo de aplicación de las técnicas de programación a nivel de tipos, usando las técnicas antiguas. La implementación original de AspectAG hace uso intensivo de estas versiones de la biblioteca.
HList sigue desarrollandose a medida de que nuevas características se añaden al lenguaje.
En lugar de reimplementar AspectAG dependiendo de nuevas versiones de HList, decidimos reescribir desde cero todas las funcionalidades necesarias, por distintos motivos:

- HList es una biblioteca experimental, en constante cambio, que no pretende ser una fuente estable de dependencias, y que constantemente cambia su interfaz sin ser compatible hacia atrás. Implementar hoy dependiendo de HList implica depender posiblemente de una versión antigua y distinta de la versión final en poco tiempo.

- Cuando programamos a nivel de tipos el lenguaje no provee fuertes mecanismos de modularización. Es común que se fugue implementación con los mensajes de error. Y la implementación basada en HList va a filtrar errores de HList, que no utilizan los mismo términos que la jerga de nuestro DSL. Si bien proveemos una solución al manejo de errores [TODO: REF], no es necesariamente exhaustiva. La biblioteca AspectAG utiliza múltiples estructuras isomorfas a Records, y dentro del propio desarrollo de la misma resultó más cómodo trabajar con estructuras con sus nombres mnemónicos.

- HList no es necesariamente adecuada si queremos tipar todo lo fuertemente posible. Por una parte es restrictiva. Por ejemplo, en la implementación vamos a utilizar una estructura que es esencialmente un Record de Records. Usando tipos de datos a medida podemos programar una solución elegante donde esto queda expresado correctamente en el kind. Implementando el Record externo como un Record de HList ofusca al interno, tratandolo como un tipo "plano". Por otra parte es muy general (se implementan varias veces las mismas funcionalidades para comparar)

- Por interés académico. Reescribir HList (varias veces, un subconjunto mayor al necesario para AspectAG) fué la forma de dominar las técnicas.

  "¿Cómo se aprende a programar? Con leer mucho código y escribir mucho código."

  — Richard Stallman

  Esto no es una razón en si para efectivamente depender de una nueva implementación en lugar de la implementación moderna de HList, pero a los argumentos anteriores se le suma que no requerimos de mayor costo.

Definición Una lista (o ás en general una colección) heterogenea es tal si contiene valores de distintos tipos. En haskell el tipo `[a]` va a ser un contenedor de valores de tipo `a`.
Por ejemplo

```
hlist = "foo" : True : []
```

conduce a un error de tipos. Existen varios enfoques para construir colecciones heterogeneas en Haskell, REF: https://wiki.haskell.org/Heterogenous_collections
A nosotros nos interesan las que son fuertemente tipadas, se conoce estáticamente el tipo de cada miembro (análogamente al largo en la implementación de vectores).
Existen variantes para definir HList (incluso considerando las técnicas modernas) REF: https://hackage.haskell.org/packa
0.5.0.0/docs/Data-HList-HList.html
Las versiones más antiguas (y sobre estas se implementó originalmente AAG) utilizan la siguiente representación:

```
data HCons a b = HCons a b
data HNil = HNil
```

Así por ejemplo

```
HCons "4" (HCons True HNil) :: HCons [Char] (HCons Bool HNil)
```

Es una lista heterogenea bien tipada. El inconveniente de esta representación (además de la verborragia) es que podemos construir tipos sin sentido como `HCons Bool Char`. Notar que esta implementación es análoga a los naturales de 2.1.3 [TODO: PONER BIEN REF], y podemos resolver el problema y "clausurar" las listas con una typeclass análoga a `TNat`. ... O sacar partido de las nuevas extensiones.
En versiones posteriores HList utilizó un GADT, y en las últimas versiones se utiliza una data Family.
En https://hackage.haskell.org/package/HList-0.5.0.0/docs/Data-HList-HList.html
se explicita cual es la ventaja de cada representación. Dado que el GADT y la data Family son prácticamente equivalentes (de hecho en nuestra implementación se pueden cambiar una por la otra), preferimos el GADT por ser la solucoón más clara y elegante.

```
data HList (l :: [Type]) :: Type  where
  HNil  :: HList '[]
  HCons :: x -> HList xs -> HList (x ': xs)
```

Por ejemplo

```
boolStrChar = HCons True $ HCons "foo" $ HCons 't' HNil
```

es un término válido de tipo `HList '[Bool, [Char], Integer]`
Una posible definición de la instancia `Show`

```
instance Show (HList '[]) where
  show _ = "[]"

instance (Show x, Show (HList xs)) => Show (HList (x ': xs))  where
  show (HCons e l) = let tsl = tail (show l)
                     in "[" ++show e ++ if tsl == "]"
                                        then "]"
                                        else  "," ++ tsl
```

Podemos definir Head o Tail seguras:

```
hHead :: HList (x ': xs) -> x
hHead (HCons x _) = x

hTail :: HList (x ': xs) -> HList xs
hTail (HCons _ xs) = xs
```

Supongamos que queremos concatenar dos listas:

```
hAppend :: HList xs -> HList ys -> HList ?? -- oops
```

Definimos la concatenación a nivel de tipos:

```
type family (xs :: [Type]) :++ ( ys :: [Type]) :: [Type]
type instance '[]       :++ ys = ys
type instance (x ': xs) :++ ys = x ': (xs :++ ys)
```

A nivel de términos:

```
hAppend :: HList xs -> HList ys -> HList (xs :++ ys)
hAppend HNil ys = ys
hAppend (HCons x xs) ys = HCons x (hAppend xs ys)
```

Una alternativa:

```
class HAppend xs ys where
  type HAppendR xs ys :: [Type]
  chAppend :: HList xs -> HList ys -> HList (HAppendR xs ys)

instance HAppend '[] ys where
  type HAppendR '[] ys = ys
  chAppend HNil ys = ys

instance (HAppend xs ys) => HAppend (x ': xs) ys where
  type HAppendR (x ': xs) ys = x ': (HAppendR xs ys)
  chAppend (HCons x xs) ys = HCons x (chAppend xs ys)
```

Consideramos por ejemplo la función reverse:

```
type family Reverse (l::[Type]) :: [Type]
type instance Reverse '[]       = '[]
type instance Reverse (x ': xs) = Reverse xs :++ '[x]
```

```
*Main> :kind! Reverse ('[Bool, [Char], Integer])
Reverse ('[Bool, [Char], Integer]) :: [*]
= '[Integer, [Char], Bool]
```

```
hReverse :: HList xs -> HList (Reverse xs)
hReverse HNil = HNil
hReverse (HCons x xs) = hAppend (hReverse xs) (HCons x HNil)
```

Intentemos programar una función que actualiza la n-ésima entrada en una lista heterogenea, incluso eventualmente cambiando el tipo. Intuitivamente debería tener el tipo:

```
updateAtNat :: Nat -> x -> HList xs -> HList xs'
```

Donde `xs'` depende de x y del **valor** del natural del tipo `Nat`. Evidentemente esto no es posible, y necesitamos el natural a nivel de tipos.
La solución es usar Proxys, o singletons (en este caso singleton es más adecuado, vamos a hacer pattern Matching sobre el natural así que lo necesitamos en runtime) Una segunda firma candidata es entonces:

```
updateAtNat :: SNat n -> x -> HList xs -> HList xs'
```

Que por supuesto aún no funciona porque no hay relación entre n, x xs y xs'.
Programemos el update a nivel de tipos:

```
type family UpdateAtNat (n :: Nat)(x :: Type)(xs :: [Type]) :: [Type]
type instance UpdateAtNat Zero     x (y ': ys) = x ': ys
type instance UpdateAtNat (Succ n) x (y ': ys) = y ': UpdateAtNat n x ys
```

```
*Main> :kind! UpdateAtNat Zero (Maybe Int) '[Bool, [Char], Integer]
UpdateAtNat Zero (Maybe Int) '[Bool, [Char], Integer] :: [*]
= '[Maybe Int, [Char], Integer]
*Main> :kind! UpdateAtNat (Succ Zero) (Maybe Int) '[Bool, [Char], Integer]
UpdateAtNat (Succ Zero) (Maybe Int) '[Bool, [Char], Integer] :: [*]
= '[Bool, Maybe Int, Integer]
*Main> :kind! UpdateAtNat (Succ (Succ (Succ Zero))) (Maybe Int) '[Bool, [Char], Integer]
UpdateAtNat (Succ (Succ (Succ Zero))) (Maybe Int) '[Bool, [Char], Integer] :: [*]
= Bool : [Char] : Integer : UpdateAtNat 'Zero (Maybe Int) '[]
```

Ahora la versón final:

```
updateAtNat :: SNat n -> x -> HList xs -> HList (UpdateAtNat n x xs)
updateAtNat SZ y (HCons _ xs) = HCons y xs
updateAtNat (SS n) y (HCons x xs) = HCons x (updateAtNat n y xs)
```

Que pasa si intentamos actualizar un índice que no existe? por ejemplo al evaluar

```
updateAtNat (SS (SS(SS SZ))) '5' mylist
```

El término de hecho está bien tipado,

```
updateAtNat (SS (SS(SS SZ))) '5' mylist
  :: HList (Bool : [Char] : Integer : UpdateAtNat 'Zero Char '[])
```

Podrámos trabajar con él, no podemos por ejemplo imprimirle:

```
<interactive>:36:1: error:
    . No instance for (Show (HList (UpdateAtNat 'Zero Char '[])))
        arising from a use of 'print'
    . In a stmt of an interactive GHCi command: print it
```

Si nuestro objetivo es que nuestros programas sean confiables, rechazar la compilación de programas incorrectos siempre que sea posible, esas expresiones deberían estar **mal tipadas**.

2.3.2 Programando con restricciones

Para resolver el problema con `updateAtNat` deberíamos **limitar** las instancias monomórficas válidas de:

```
updateAtNat2 :: forall (n::Nat) (x :: Type) (xs :: [Type]).
                SNat n -> x -> HList xs -> HList (UpdateAtNat n x xs)
```

Esto es justamente lo que podemos hacer con una typeclass (Predicar sobre los tipos).
Consideramos la siguiente definición alternativa:

```
class UpdateAtNat (n :: Nat) (y :: Type) (xs :: [Type]) where
  type UpdateAtNatR n y xs :: [Type]
  updateAtNat :: SNat n -> y -> HList xs -> HList (UpdateAtNatR n y xs)

instance UpdateAtNat Zero y (x ': xs) where
  type UpdateAtNatR Zero y (x ': xs) = (y ': xs)
  updateAtNat SZ y (HCons _ xs) = HCons y xs

instance UpdateAtNat n y xs
  => UpdateAtNat (Succ n) y (x ': xs) where
  type UpdateAtNatR (Succ n) y (x ': xs) =  x ': UpdateAtNatR n y xs
  updateAtNat (SS n) y (HCons x xs) = HCons x (updateAtNat n y xs)
```

Funciona correctamente para los índices válidos, por ejemplo `updateAtNat (SS (SS SZ)) True boolStrChar` reduce a `[True,"foo",True]`, pero si evaluamos:

```
updateAtNat (SS (SS (SS SZ))) True boolStrChar
```

Obtenemos un error de compilación, como queríamos:

```
<interactive>:32:1: error:
    . No instance for (UpdateAtNat 'Zero Bool '[])
        arising from a use of 'updateAtNat'
    . In the expression: updateAtNat (SS (SS (SS SZ))) True boolStrChar
      In an equation for 'it':
          it = updateAtNat (SS (SS (SS SZ))) True boolStrChar
```

En la reimplementación de AspectAG este estilo de programación será ampliamente usado.
[TODO: tambien se puede presentar aca la version con dependencia funcional, para explicar por que es util y que no necesariamente significa un paso atras talvez hay que usar un ejemplo mejor..]

2.3.3 Manejo de Errores

El error de compilación tal y como lo aprecia el programador en la sección anterior es bastante engañoso. Mucho más aún si consideramos código complicado. "`No instance for (UpdateAtNat 'Zero Bool '[])` " no nos dice nada sobre la lista original.
Kyselyov et al [6] proponen una solución (en el año 2004). [TODO Mover esto arriba?]
En lugar de la definición antigua:

```
class Fail e
data PositionOutOfBound

instance Fail (PositionOutOfBound) => UpdateAtNat n x '[] where
  type UpdateAtNatR n x '[] = '[]
  updateAtNat = undefined
```

Que conduce al error "`No instance for (Fail PositionOutOfBound)`"
Podemos recurrir al módulo `GHC.TypeLits`. [4]
EXPLICAR MAS

```
instance TypeError (Text "Type Error ----" :$$:
                    Text "From the use of 'UpdateAtNat' :" :$$:
                    Text "Position Out of Bound" :$$:
                    Text "Perhaps your list is too short?")
  => UpdateAtNat n x '[] where
  type UpdateAtNatR n x '[] = '[] -- unreachable
  updateAtNat = undefined
```

La compilación produce un error mucho más legible:

```
<interactive>:10:1: error:
    . Type Error ----
      From the use of 'UpdateAtNat' :
      Position Out of Bound
      Perhaps your list is too short?
    . In the expression: updateAtNat (SS (SS (SS SZ))) True boolStrChar
```

2.3.4 Logic vs Functional

Supongamos que queremos codificar:

```
getByType :: x -> HList xs
```

2.4 Records Heterogeneos

AspectAG requiere de *Records* heterogeneos, esto es, colecciones etiqueta-valor, heterogeneas, donde además las claves estén dadas por tipos, y no por valores.
El enfoque de HList para implementarles era utilizar una lista Heterogenea, donde cada entrada era del tipo `Tagged l v`, definido como

```
Tagged l v = Tagged v
```

Esto no es satisfactorio con las herramientas modernas, no se está utilizando la posibilidad de tipar que nos provee la promoción de datos. HList implementa predicados como typeclasses para asegurar que todos los miembros son de tipo `Tagged` cuando podría expresarse en el kind. Además

[4] `TypeError` también es una typefamily (polykinded) y funciona como una versión promovida de `error :: [Char] -> a`, Podríamos también operar sobre la TypeFamily (la primer implementación) y evitar el uso de Typeclasses en este caso.

las etiquetas son de kind `Type`, cuando en realidad nunca requieren estar habitadas a nivel de valores. AspectAG genera etiquetas utilizando metaprogramación con Template Haskell [REF], mientras podría usarse simplemente data promotion.
En su lugar se propone la siguiente implementación:

```
data Record :: forall k . [(k,Type)] -> Type where
  EmptyR :: Record '[]
  ConsR  :: LabelSet ( '(l, v) ': xs) =>
   Tagged l v -> Record xs -> Record ( '(l,v) ': xs)
```

Un record es una lista con más estructura. Notar que el record vacío es análogo a la lista vacía. Para agregar un campo, requerimos un valor de tipo `Tagged l v` definido como:

```
data Tagged (l :: k) (v :: Type) where
  Tagged :: v -> Tagged l v
```

Tegged es polimorfico en el kind de las etiquetas. Notar que para construir un valor hay que anotar el tipo de `l` o utilizar un `constructor inteligente` que use un proxy (o en la jerga de AspectAG, una etiqueta).

```
tag :: Label l -> v -> Tagged l v
```

La restricción Labelset garantiza que las etiquetas no se repitan, y se explica en la sección siguiente.

2.4.1 Más Restricciones

Las typeclasses, entendidas como predicados sobre tipos

```
class LabelSet (l :: [(k,k2)])
instance LabelSet '[]         -- empty set
instance LabelSet '[ '(x,v)]  -- singleton set

instance ( HEqK l1 l2 leq
         , LabelSet' '(l1,v1) '(l2,v2) leq r)
        => LabelSet ( '(l1,v1) ': '(l2,v2) ': r)

class LabelSet' l1v1 l2v2 (leq::Bool) r
instance ( LabelSet ( '(l2,v2) ': r)
         , LabelSet ( '(l1,v1) ': r)
         ) => LabelSet' '(l1,v1) '(l2,v2) False r

instance TypeError (Text "LabelSet Error:" :$$:
                    Text "Duplicated Label on Record" :$$:
                   Text "On fields:" :$$: ShowType l1 :$$:
                   Text " and " :$$: ShowType l1 )
          => LabelSet' l1 l2 True r
```

3 AspectAG

3.1 Gramáticas de atributos

Las gramáticas de atributos [7] son un formalismo para describir computaciones recursivas sobre tipos de datos. Dada una gramática libre de contexto, se le asocia una semántica considerando atributos en cada producción, que toman valores que son calculados mediante reglas a partir de los valores de los atributos de los padres y de los hijos del arbol de sintaxis abstracta.
Los atributos se dividen clásicamente en dos tipos: heredados y sintetizados. Los atributos heredados son "pasados" como un contexto desde los padres a los hijos. Los atributos sintetizados son calculados según las reglas semánticas, en función de los atributos de los hijos (y eventualmente de los padres).
Un *Aspecto* es una colección de (uno o más) aspectos, y sus reglas de cómputo.
Las gramáticas de atributos son especialmente interesantes para la implementación de compiladores [REF UHC, Aho, etc etc], traduciendo el arbol sintáctico directamente en algún lenguaje de destino o representación intermedia. También es posible validar chequeos semánticos de reglas que no están presentes sintácticamente (por ejemplo compilando lenguajes con sintaxis no libre de contexto, parseados previamente según una gramática libre de contexto como la mayoría de los languajes de programación modernos), o para implementar chequeadores de tipos.
Además, las gramáticas de atributos son útiles en sí mismas como un paradigma de programación, y significan una solución a un conocido tópico de discusión en la comunidad llamado "El problema de la expresión" ("The expression problem", término acuñado por P. Wadler [14]). Cuando el software se construye de manera incremental es deseable que sea sencillo introducir nuevos tipos de datos o enriquecer los existentes, y también que sea simple implementar nuevas operaciones. Normalmente diseñar un lenguaje pensando en una de las utilidades va en desmedro de la otra, siendo la programación orientada objetos el ejemplo paradigmático de técnica orientada a los datos, y la programación funcional, por el contrario el ejemplo donde es simple agregar funciones, siendo costoso en cada paradigma hacer lo dual (pensar en cuan complicado (y cuantos módulos hay que modificar) es agregar un método en una estructura de clases amplia, o cuantas funciones hay que modificar en los lenguajes funcionales si en un tipo algebráico se agrega una construcción).
Las *Programación orientada a aspectos*, mediante gramáticas de atributos son una propuesta de solución a este problema, debería ser simple agregar nuevas producciones (definiendo *localmente* las reglas de computación de los atributos existentes sobre el nuevo caso, así como agregar nuevas funcionalidades (definiendo *localmente* nuevos atributos con sus reglas, o bien combinando los ya existentes).
Por sus características, donde las computaciones se expresan de forma local en cada producción combinando cómo la información fluye de arriba a abajo y de abajo a arriba, una aplicación útil de las AGs es la de definir computaciones circulares. En la próxima sección introducimos un ejemplo.

3.2 Ejemplo: `repmin`

Como ejemplo consideramos la clásica función `repmin` [1], que dado un árbol de enteros (por ejemplo con la información en las hojas), retorna un árbol con la misma topología, conteniendo el menor valor del árbol original en cada hoja. Consideramos la siguiente estructura en haskell para representar el árbol:

```
data Root = Root Tree deriving Show
```

```
data Tree = Node Tree Tree
          | Leaf Int
          deriving Show
```

Notar que utilizaremos la raíz "marcada" con el tipo algebráico `Root` en lugar de definir los árboles como es usual, donde la raíz es un nodo más. Lo hacemos de esta manera para tener información de donde exactamente dejar de calcular el mínimo local, que será a partir de ese punto global y comenzar a propagarlo a los hijos.
La función `repmin` puede definirse como sigue:

```
repmin = sem_root

sem_root :: Root -> Tree
sem_root (Root tree)
  = let (smin,sres) = (sem_Tree tree) smin
    in sres

sem_Tree :: Tree -> Int -> (Int, Tree)
sem_Tree (Node l r)
  = \ival -> let (lmin,lres) = (sem_Tree l) ival
                 (rmin,rres) = (sem_Tree r) ival
             in (lmin `min` rmin, Node lres rres )

sem_Tree (Leaf i)
  = \ival -> (i, Leaf ival)
```

Por otra parte, una definición por gramáticas de atributos viene dada de la siguinte manera:

```
DATA Root | Root tree
DATA Tree | Node l, r : Tree
          | Leaf  i : {Int}
SYN Tree  [smin : Int]
SEM Tree
      | Leaf  lhs .smin = @i
      | Node lhs .smin = @l.smin `min` @r.smin
INH  Tree [ival : Int]
SEM Root
      | Root  tree.ival  = @tree.smin
SEM Tree
      | Node l    .ival  = @lhs.ival
             r    .ival  = @lhs.ival
SYN Root Tree [sres : Tree]
SEM Root
      | Root  lhs .sres  = @tree.sres
SEM Tree
      | Leaf  lhs .sres  = Leaf @lhs.ival
      | Node lhs .sres  = Node @l.sres @r.sres
```

Nuevamente tenemos un árbol con raíz explícita. La razón para tomar ésta decisión es una vez más tener un símbolo de inicio explícito de la gramática, que a nivel operacional nos va a permitir saber cuando encadenar atributos sintetizados con heredados, aunque ahora la decisión es más natural; la semántica (i.e. cómo se computan los atributos) difiere en un nodo ordinario y en la raíz.
La palabra clave `SYN` introduce un atributo sintetizado. `smin` y `sres` son atributos sintetizdos de tipo `Int` y `Tree` respectivamente. Las semánticas de cada uno se definen luego de la sentencia `SEM`.
`smin` representa en cada producción el mínimo valor de una hoja en el subárbol correspondiente, calculandose en las hojas como el valor que contiene (que, formalmente puede considerarse un nuevo atributo, implícito), y en los nodos como una función (el mínimo) del valor del mismo atributo `smin` en los subárboles.
`sres` en cada producción vale un árbol con la misma forma que el subárbol original, con el mínimo global en cada hoja. En la raíz se copia el subárbol, en cada nodo se construye un nodo con los subárboles que contiene el atributo `sres` en los subárboles. En las hojas se calcula en función del atributo heredado `ival`.
Los atributos heredados se definen con la sentencia `INH`. En el ejemplo `ival` es el único atributo heredado, que representa el valor mínimo global en el árbol.
En la raíz, `ival` se computa como una copia del valor `smin`. Se aprecia por qué necesitabamos marcar la raíz del árbol: para saber cuando copiar. En los nodos, a cada subárbol se le copia el valor de ival actual.

3.3 AspectAG

La implementación sigue a grandes razgos la siguiente idea:
En cada producción, llamamos *atribución* (Attribution) al registro de todos los atributos. Una atribución será un mapeo de nombres de atributos a sus valores. Los nombres de atributos se manejan en tiempos de compilación, por lo que una estructura como la presentada en la sección 2.4 es adecuada. La definición de la estructura en tiempo de compilación permitirá realizar chequeos estáticos de propiedades deseables de la gramática.
En cada producción, la información fluye de los atributos heredados del padre y los sintetizados de los hijos, que se llaman en la literatura "familia de entrada" (*input family*), a los sintetizados del padre y heredados de los hijos, la *output family*.
En cada producción, una regla semántica consiste en un mapeo de una input family a una output family.
Presentamos una solución al problema `repmin` en la reimplementación del EDSL, para que el lector tome contacto con el estilo de programación en el EDSL. Luego se presentará mayor detalle la implementación.
Hay que definir múltiples *Etiquetas*. Hay etiquetas para los no terminales, para los atributos, y para nombrar a los hijos en cada producción. Por ejemplo, para los atributos:

```
data Att_smin; smin = Label :: Label Att_smin
data Att_ival; ival = Label :: Label Att_ival
data Att_sres; sres = Label :: Label Att_sres
```

Las etiquetas existen solo a nivel de tipos([REF phantom types]), `Label` es una implementación especializada de `Proxy`. En nuestra implementación todos los Records extensibles son polimórficos en el kind de los índices, por lo cual es posible definir tipos de datos para cada tipo de etiqueta y utilizar el kind promovido.

Defínanse las reglas para el atributo `smin`. Notar en la especificación de la gramática de atributos que tiene reglas de computación en el árbol, por lo que hay dos producciones (Node y Leaf). En AspectAG:

```
node_smin (Fam chi par)
  = syndef smin $ (chi # ch_l # smin) `min` (chi # ch_r # smin)

leaf_smin (Fam chi par)
  = syndef smin $ chi # ch_i # leafVal
```

Informalmente, para el nodo se define un atributo `smin`, que se calcula como el mínimo entre el valor de `smin` del hijo `ch_l` (nombre del hijo izquierdo), y el valor de `smin` del hijo `ch_r`. En el caso de la hoja, se toma el valor de leafVal (que es un nombre para el valor guardado), para el (unico) hijo en la producción `ch_i`. Si bien todos los terminales van a tener un único hijo con su valor, implementar fuertemente tipado nos obliga a respetar esta estructura (o complicar mucho la implementación).
Notar que lo que definimos son en realidad funciones: un mapeo de la *input family* (atributos heredados del padre y sintetizados de los hijos) a la *output family* (sintetizados del padre, heredados a los hijos). Los valores de arriba tienen tipo *Rule* [RFRENCIA MAS ADELANTE].
Análogamente se define el atributo heredado `sres`:

```
root_sres (Fam chi par)
  = syndef sres $ chi # ch_tree # sres

node_sres (Fam chi par)
  = syndef sres $ Node (chi # ch_l # sres)(chi # ch_r # sres)

leaf_sres (Fam chi par)
  = syndef sres $ Leaf (par # ival)
```

Notar que está definido para la raíz, y que en la hoja usamos un atributo sintetizado para computar.
Por último presentamos el atributo sintetizado:

```
root_ival (Fam chi par) =
  inhdef ival [nt_Tree] ( ch_tree .=.(chi # ch_tree # smin)
                        .*. emptyRecord)

node_ival (Fam chi par) =
  inhdef ival [nt_Tree] (  ch_l .=. par # ival
                        .*. ch_r .=. par # ival
                        .*. emptyRecord)
```

Informalmente, declaramos que se define un atributo heredado llamado `ival`, y se declara un registro donde se especifica para cada hijo cómo se computará `ival`. El parámetro extra `[nt_Tree]` es una lista de no terminales, por ahora no le damos importancia.
Los *aspectos* se definen como un registro con las reglas para cada producción:

```
asp_ival =  p_Root .=. root_ival
        .*. p_Node .=. node_ival
        .*. emptyRecord
asp_sres =  p_Root .=. root_sres
        .*. p_Node .=. node_sres
        .*. p_Leaf .=. leaf_sres
        .*. emptyRecord
asp_smin =  p_Leaf .=. leaf_smin
        .*. p_Node .=. node_smin
        .*. emptyRecord
```

Que pueden combinarse mediante el operador `.+.`:

```
asp_repmin = asp_smin .+. asp_sres .+. asp_ival
```

Finalmente `repmin :  Tree -> Tree` viene dado por:

```
repmin t = sem_Root asp_repmin (Root t) emptyAtt # sres
```

En donde `sem_Root` es una función definida una sola vez.

4 Reimplementacin de AAG

4.1 Estructuras de Datos

Como se definió antes, una atribución (*attribution*) es un mapeo de nombres de atributos (que serán represenentados puramente a nivel de tipos como etiquetas) a sus valores. La estructura de Registro extensible (fuertemente tipado) presentada anteriormente es ideal para representarles.
Para obtener mensajes de error precisos y evitar que se filtre implementación en los mismos, decidimos tener estructuras especializadas.
Un atributo (etiquetado) viene entonces dado por:

```
newtype Attribute label value = Attribute value
```

que es el componente principal para construir atribuciones:

```
data Attribution :: forall k . [(k,Type)] -> Type where
  EmptyAtt :: Attribution '[]
  ConsAtt  :: LabelSet ( '(att, val) ': atts) =>
   Attribute att val -> Attribution atts -> Attribution ( '(att,val) ': atts)
```

Notar que ya estamos utilizando todo el poder de las extensiones modernas. Se utilizan kinds promovidos en las listas (`-XDataKinds`), polimorfismo en kinds en las etiquetas (`-XPolyKinds`) la estructura es un GADT (`-XGADTs`), LabelSet está predicando sobre un kind polimórfico (por lo que usamos kind equality)(`ConstraintKinds`), y el kind `Type` fué introducido en `-XTypeInType`.
Una `familia` consiste en la atribución del padre y una colección de atribuciones para los hijos (etiquetadas por sus nombres).
Representamos ésta última estructura como

```
data ChAttsRec :: forall k k' . [(k , [(k',Type)])] -> Type where
  EmptyCh :: ChAttsRec '[]
  ConsCh  :: LabelSet ( '(l, v) ': xs) =>
   TaggedChAttr l v -> ChAttsRec xs -> ChAttsRec ( '(l,v) ': xs)
```

Notar la analogía con la anterior, es de nuevo una implementación de Registro heterogeneo, especializada. Notar también que las etiquetas no tienen por qué tener el mismo kind. Esto se decidió así para soportar posiblemente a futuro la generación de etiquetas por parte del programador a nivel de valores y usar los kinds promovidospara las mismas.
Dado que una atribución una vez bajo el wrapper `Attrribution` tiene kind `Type`, podríamos haber implementado a los hijos como un registro agnóstico respecto al contenido. Se prefiere una implementación fuertemente tipada sobre reutilizar el código existente.
En cada nodo de la gramática, una *Familia* contiene la atribución del padre y la colección de atribuciones de los hijos.

```
data Fam (c::[(k,[(k,Type)])]) (p :: [(k,Type)]) :: Type where
  Fam :: ChAttsRec c  -> Attribution p -> Fam c p
```

Una regla es una función de la familia de entrada a la de salida, el tipo de las reglas se implementa con una aridad extra para hacerlas componibles, como en [REF al paper de Moor et al]

```
type Rule sc ip ic sp ic' sp'
  = Fam sc ip -> (Fam ic sp -> Fam ic' sp')
(f `ext` g) input = f input . g input
```

Para ser más precisos, el tipo de rule:

```
type Rule (sc  :: [(k', [(k, Type)])])
          (ip  :: [(k,       Type)])
          (ic  :: [(k', [(k, Type)])])
          (sp  :: [(k,       Type)])
          (ic' :: [(k', [(k, Type)])])
          (sp' :: [(k,       Type)])
  = Fam sc ip -> Fam ic sp -> Fam ic' sp'
```

4.2 Declaraciones de Reglas

Se proveen distintas construcciones para luego declarar reglas. En el ejemplo se utilizaron `syndef` e `inhdef`, que son las mínimas adecuadas para tener un sistema interesante.
En la implementación se proveen otras construcciones, y parte del trabajo futuro pasa por codificar otras nuevas.
Por ejemplo, la función `syndef` provee la definición de un nuevo atributo sintetizado. Dada una etiqueta no definida previamente, que represente el nombre del atributo a definir, y un valor para el mismo, construye una función que actualiza la familia construida hasta el momento.

```
syndef  :: LabelSet ( '(att,val) ': sp) =>
    Label att -> val -> (Fam ic sp -> Fam ic ( '(att,val) ': sp))
syndef latt val (Fam ic sp) = Fam ic (latt =. val *. sp)
```

Como ejemplo de una primitiva alternativa,

```
synmod  ::  UpdateAtLabelAtt att val sp sp'
  =>  Label att -> val -> Fam ic sp -> Fam ic sp'
synmod att v (Fam ic sp) = Fam ic (updateAtLabelAtt att v sp)
```

La función `inhdef` introduce un atributo heredado de nombre `att` para una colección de no terminales `nts`. `vals` es un registro con claves consistentes en los nombres de los hijos, conteniendo valores que describen como computar el atributo que está siendo definido para cada uno de ellos. En contraste con `syndef`, es bastante más compleja de implementar.
Primero, es necesaria una función auxiliar insertar una definición en la atribución de un hijo:

```
class SingleDef (mch::Bool)(mnts::Bool) att pv (ic ::[(k,[(k,Type)])])
                 (ic' ::[(k,[(k,Type)])]) | mch mnts att pv ic -> ic' where
  singledef :: Proxy mch -> Proxy mnts -> Label att -> pv -> ChAttsRec ic
               -> ChAttsRec ic'

instance ( HasChild lch ic och
         , UpdateAtChild lch ( '(att,vch) ': och) ic ic'
         , LabelSet ( '(att, vch) ': och))
  => SingleDef True True att (Tagged lch vch) ic ic' where
  singledef _ _ att pch ic =
    updateAtChild (Label :: Label lch) ( att =. vch *. och) ic
    where lch = labelTChAtt pch
          vch = unTaggedChAtt pch
          och = lookupByChild lch ic
```

[MOSTRAR MAS COSAS]

4.3 Aspectos

4.3.1 Definifión

Un aspecto (*aspect*) se implementa simplemente como un registro heterogeneo, (que contendrá reglas, etiquetadas por nombres de producciones). En este caso no consideramos necesaria une implementación especializada. En contraste a las atribuciones y registros de atribuciones de los hijos, en donde la estructura es compleja e induce a errores de programación (por lo que preferimos tipar lo más fuertemente posible), aquí se prefiere reutilizar código.
Notar que de cualquier manera, para el usuario de la biblioteca la construcción de malas instancias puede ser prohibida por constructores inteligentes. En particular al utilizar el combinador de aspectos se utiliza la función `comSingle` que solo compila si las reglas están bien formadas, como vemos más adelante.
Tenemos disponible el poder de los tipos dependientes (casi, al menos una simulación de los mismos) y podríamos chequear otras propiedades.

```
type Aspect = Record                 -- tipo de los Aspectos
type Prd prd rule = Tagged prd rule  -- tipo de las producciones
```

4.3.2 Combinación de Aspectos

La combinación de aspectos viene dada por la función `.+.` definida a nivel de tipos como la clase `Com`.

```
class Com (r :: [(k,Type)]) (r' :: [(k, Type)]) (r'' :: [(k,Type)])
  | r r' -> r'' where
  (.+.) :: Aspect r ->  Aspect r' ->  Aspect r''
```

La función inserta en el resultado intactas las producciones que aparecen en un solo aspecto parámetro. Por otro lado las producciones que aparezcan en ambos aspectos deberán incluirse con las reglas combinadas (según la función `ext` definida previamente).
La función de combinación viene definida por recursión en la segunda componente. Si el segundo registro es vacío, en la operación es neutro.

```
instance Com r '[] r where
  r .+. _ = r
```

Si la segunda componente consiste en al menos una producción con su regla, la combinamos al primer aspecto mediante la función `comSingle`, y llamamos recursivamente a la combinación del nuevo registro creado con la cola del segundo parámetro.

```
instance ( Com r''' r' r''
         , HasLabelRec prd r
         , ComSingle (HasLabelRecRes prd r) prd rule r r''')
  => Com r ( '(prd, rule) ': r') r'' where
     r .+. (pr `ConsR` r') = let
                                 b   = hasLabelRec (labelPrd pr) r
                                 r'''= comSingle b pr r
                                 r'' = r''' .+. r'
                             in  r''
```

La función `comSingle` es una función cuyo comportamiento es dependiente de los tipos de la producción y el Aspecto parámetro. Si ya existe una producción con ese nombre de deben combinar las reglas en el campo correspondiente del aspecto, sino, el Aspecto debe extenderse. Implementamos `ComSingle` con un parámetro booleano extra que indica la pertenencia o no de la etiqueta `prd` al registro `r` La firma viene dada por:

```
class ComSingle (b::Bool) (prd :: k) (rule :: Type) (r :: [(k,Type)])
                (r :: [(k,Type)]) | b prd rule r -> r where
  comSingle :: Proxy b -> Prd prd rule -> Aspect r -> Aspect r
```

Como se detalló en la sección 2.2.7, en Haskell requerimos testigos en tiempo de ejecución en computaciones de tipos dependientes, por lo que debemos incluir el parámetro `Proxy b`.
Como se aprecia en la definición del caso recursivo de la clase `Com`, el booleano, tanto a nivel de tipos como de valores es una función predicado de pertenencia para los registros (HasLabelRec/`hasLabelRec`).

```
class HasLabelRec (e :: k)(r ::[(k,Type)]) where
  type HasLabelRecRes (e::k)(r ::[(k,Type)]) :: Bool
  hasLabelRec :: Label e -> Record r -> Proxy (HasLabelRecRes e r)
```

```
instance HasLabelRec e '[] where
  type HasLabelRecRes e '[] = 'False
  hasLabelRec _ _ = Proxy

instance HasLabelRec  k ( '(k' ,v) ': ls) where
  type HasLabelRecRes k ( '(k' ,v) ': ls)
      = Or (k == k') (HasLabelRecRes k ls)
  hasLabelRec _ _ = Proxy
```

en este caso usamos un tipo indizado en lugar de dependencias funcionales. [esto no es muy justificable, en realidad habría que converger a las type families, y después lo haré pero mientras están estos baches que no se bien como justificar]
`Or` es una función definida puramente a nivel de tipos, y la implementamos como una *Type Family*

```
type family Or (l :: Bool)(r :: Bool) :: Bool where
  Or False b = b
  Or True b  = 'True
```

Luego podemos definir las instancias posibles de `ComSingle`, y además chequeamos ciertas propiedades. En particular cuando combinamos reglas chequeamos que efectivamente estamos combinando reglas.

```
instance (LabelSet ('(prd, rule) : r))
   => ComSingle 'False prd rule r ( '(prd,rule) ': r) where
  comSingle _ prd asp = prd `ConsR` asp

instance ( HasFieldRec prd r,
           LookupByLabelRec prd r ~ (Rule sc ip ic' sp' ic'' sp'')
         , UpdateAtLabelRec prd (Rule sc ip ic  sp  ic'' sp'') r r'
         )
  => ComSingle 'True prd        (Rule sc ip ic  sp  ic'  sp') r r' where
  comSingle _ f r = updateAtLabelRec l (oldR `ext` newR) r :: Aspect r'
    where l    = labelPrd f                                 :: Label prd
          oldR = lookupByLabelRec l r
          newR = rulePrd f
```

4.3.3 Funciones semánticas

En cada producción, llamamos *función semántica* al mapeo de los atributos heredados a los atributos sintetizados. Obsérvese que una computación consiste en exactamente computar las funciones semánticas.
En el ejemplo [REF], la función `sem_Tree` construye, dados un aspecto y un árbol una función semántica. El tipo de `sem_Tree`, si ignoramos los parámetros implícitos de las restricciones de typeclasses, viene dado por:

```
sem_Tree :: Aspect r -> Tree -> Attribution ip -> Attribution sp
```

Observemos la definición en uno de los casos:

```
sem_Tree asp (Node l r) =  knit (asp .#. p_Node) $
                           ch_l .=. sem_Tree asp l
                      .*. ch_r .=. sem_Tree asp r
                      .*. emptyRecord
```

Es la función `knit` la que se encarga de construir la función semántica a partir de las funciones semánticas de los hijos (notar que cada llamada recursiva a `sem_Tree`, parcialmente aplicada a dos parámetros es exactamente una función semántica).
El tipo completo de `sem_Tree` viene dado por:

```
sem_Tree
  :: (HasFieldRec P_Node r, HasFieldRec P_Leaf r,
      LookupByLabelRec P_Node r
      ~ (Rule '[ '((Ch_l, Tree),  sp), '((Ch_r, Tree),  sp)]  ip
              '[ '((Ch_l, Tree), '[]), '((Ch_r, Tree), '[])] '[]
              '[ '((Ch_l, Tree),  ip), '((Ch_r, Tree),  ip)]  sp),
      LookupByLabelRec P_Leaf r
      ~ (Rule  '[ '((Ch_i, Int), '[ '((Val, Int), Int)])]     ip
               '[ '((Ch_i, Int), '[])]                       '[]
               '[ '((Ch_i, Int),  p)]                         sp)) =>
     Aspect r -> Tree -> Attribution ip -> Attribution sp
```

Se aprecian múltiples predicados que deben chequearse para que las llamadas a `sem_Tree` **compilen**. Una llamada donde el Aspecto `r` no contenga definiciones para los nodos o para las hojas no compilará. Además en el valor imagen de cada una de éstas etiquetas debe haber una regla que cumpla ciertas restricciones de forma. Por supuesto, como un aspecto es un registro, por lo que no van a permitirse instancias donde se dupliquen etiquetas de producciones. No existe sin embargo ninguna restricción sobre el largo de `r` o las etiquetas adicionales que contiene, lo cual tiene sentido porque eventualmente la gramática podría extenderse con nuevas producciones.

4.3.4 La función `knit`

La función `knit`[REF] realiza la verdadera computación. Toma las reglas combinadas para una producción, y las funciones semánticas de los hijos, y construye la función semántica del padre.

```
knit :: ( Empties fc , EmptiesR fc ~ ec
        , Kn fc ic sc )
  => Rule sc ip ec '[] ic sp -> Record fc -> Attribution ip -> Attribution sp
knit rule fc ip
  = let ec          = empties fc
        (Fam ic sp) = rule (Fam sc ip) (Fam ec EmptyAtt)
        sc          = kn fc ic
    in  sp
```

Primero se construye una familia de salida vacía, mediante la función `empties`. Ésta contiene atribuciones vacías tanto para el padre como para todos los hijos. A partir de la familia de entrada y nuestra familia "dummy" construimos la familia de salida. La familia de entrada consta de los atributos heredados del padre `ip` que tenemos disponibles como parámetro, y de los sintetizados de

los hijos `sc`. Tenemos disponibles los atributos heredados de los hijos y las funciones semánticas, por lo que para computar `sc` debemos ejecutar `knit` en cada uno de los hijos, trabajo realizado por la función `kn`, que es una función `map` especializada.

```
class Empties (fc :: [(k,Type)]) where
  type EmptiesR fc :: [(k, [(k, Type)])]
  empties :: Record fc -> ChAttsRec (EmptiesR fc)

instance Empties '[] where
  type EmptiesR '[] = '[]
  empties EmptyR = EmptyCh

instance (Empties fcr,
          LabelSet ( '(lch, '[]) ': EmptiesR fcr)) =>
  Empties ( '(lch, fch) ': fcr) where
  type EmptiesR ( '(lch, fch) ': fcr) = '(lch, '[]) ': EmptiesR fcr
  empties (ConsR pch fcr)
    = let lch = labelTChAtt pch -- TODO: name
      in  ConsCh (TaggedChAttr lch EmptyAtt) (empties fcr)

class Kn (fcr :: [(k, Type)])
         (icr :: [(k, [(k, Type)])])
         (scr :: [(k, [(k, Type)])]) | fcr -> scr icr where
  kn :: Record fcr -> ChAttsRec icr -> ChAttsRec scr

instance Kn '[] '[] '[]
  kn _ _ = EmptyCh

instance ( Kn fc ic sc
         , LabelSet ('(lch, sch) : sc)
         , LabelSet ('(lch, ich) : ic))
  =>  Kn ( '(lch , Attribution ich -> Attribution sch) ': fc)
         ( '(lch , ich) ': ic)
         ( '(lch , sch) ': sc) where
  kn (ConsR pfch fcr) (ConsCh pich icr)
   = let scr = kn fcr icr
         lch = labelTChAtt pfch    :: Label lch
         fch = unTagged pfch       :: Attribution ich -> Attribution sch
         ich = unTaggedChAttr pich :: Attribution ich
     in ConsCh (TaggedChAttr lch (fch ich)) scr
```

5 Comparación

ambiguous types al final
monomorphism restriction
hack con las funciones semanticas

References

[1] Richard Bird. Using circular programs to eliminate multiple traversals of data. 21:239–250, 10 1984.

[2] 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.

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

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

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

[6] 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.

[7] Donald E. Knuth. Semantics of context-free languages. In *In Mathematical Systems Theory*, pages 127–145, 1968.

[8] Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. *SIGPLAN Not.*, 48(12):81–92, September 2013.

[9] Conor McBride. Faking it: Simulating dependent types in haskell. *J. Funct. Program.*, 12:375–392, 2002.

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

[11] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, January 1997.

[12] Alexandra Silva and Joost Visser. Strong types for relational databases. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 25–36, New York, NY, USA, 2006. ACM.

[13] 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.

[14] P. Wadler. The expression problem, mailing list discussion. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt. Accedido: 28-8-2018.

[15] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM.