Reimplementación de *AspectAG* basada en nuevas extensiones de Haskell

Juan García Garland

November 13, 2018

Contents

1 Introducción

AspectAG [13] es un EDSL (lenguaje de dominio específico, embebido) desarrollado en Haskell que permite la construccin modular de Gramáticas de Atributos. En AspectAG, los fragmentos de una Gramática de Atributos son definidos en forma independiente y luego combinados a través del uso de operadores de composición que el propio EDSL provee. AspectAG se basa fuertemente en el uso de registros extensibles, los cuales son implementados en términos de HList [8], una biblioteca de Haskell para la manipulación de listas heterogéneas de forma fuertemente tipada. HList está implementada utilizando técnicas de programación a nivel de tipos (los tipos son usados para representar valores a nivel de tipos y las clases de tipos son usadas para representar tipos y funciones en la manipulacin a nivel de tipos).
Desde el momento de la implementación original de AspectAG hasta la actualidad la programacin a nivel de tipos en Haskell ha tenido una evolución importante, habiéndose incorporado nuevas extensiones como *data promotion* o polimorfismo de kinds, entre otras, las cuales constituyen elementos fundamentales debido a que permiten programar de forma "fuertemente tipado" a nivel de tipos de la misma forma que cuando se programa a nivel de valores (algo que originalmente era imposible o muy difcil de lograr). El uso de estas extensiones da como resultado una programación a nivel de tipos más robusta y segura. Sobre la base de estas extensiones, implementamos un subconjunto de la biblioteca original.
Estructura del documento:
En la sección 2 [ref] se presenta un estudio de las técnicas de programación a nivel de tipos y las extensiones de haskell que provee el compilador GHC que las hacen posibles. [TODO: redactar bien de aca en adelante]. Se presentan las estructuras de Listas Heterogeneas y Registros Heterogeneos que implementa originalmente `HList`, de las que depende AspectAG.
En la seccióin [ref] se presentan las gramáticas de atributos y en particular la implementación (nueva) de AspectAG mediante un ejemplo que introduce las primitivas importantes de la biblioteca.
En la sección [ref] se presentan los detalles de la implementación, que se basan en las técnicas de programación a nivel de tipos modernas.
Finalmente [ref] se presenta una breve comparación que ilustra las ventajas del uso de las técnicas modernas.
El código fuente de la biblioteca y la documentación se encuentra disponible en
http://https://gitlab.fing.edu.uy/jpgarcia/AspectAG/.
En la sección te tests se implementan ejemplos de utilización de la biblioteca.

2 Programación a nivel de tipos

2.1 Técnicas de programación

La biblioteca AspectAG presentada originalmente en 2009, además de implementar un sistema de gramáticas de atributos como un EDSL, provee un buen ejemplo de como usar las programación a nivel de tipos en Haskell. La implementación utiliza fuertemente los registros extensibles implementados por la biblioteca HList. Ambas bibliotecas se basan en la combinación de las extensiónes `MultiParamTypeClasses` (que proveen la implementación de relaciones a nivel de tipos) con `FunctionalDependencies` [7] (que permite expresar en particular relaciones funcionales).[1]
Durante la década siguiente [2] se han implementado múltiples extensiones en el compilador GHC que proveen herramientas para hacer la programación a nivel de tipos más expresiva. Las familias de tipos implementadas en la extensión `TypeFamilies` [5] [4] [12] nos permiten definir funciones a nivel de tipos de una forma más idiomática que el estilo lógico de la programación orientada a relaciones por medio de clases y dependencias funcionales. La extensión `DataKinds` [16] implementa la *data promotion* que provee la posibilidad de definir tipos de datos -tipados- a nivel de tipos, introduciendo nuevos kinds. Bajo el mismo trabajo Yorgey et al implementan la extensión `PolyKinds` proveyendo polimorfismo a nivel de kinds. Además la extensión `KindSignatures` permite anotar kinds a los términos en el nivel de tipos. Con toda esta maquinaria se tiene un lenguaje a nivel de tipos casi tan expresivo como a nivel de términos[3]. La extensión `GADTs` combinada con

[1] Además de otras relaciones de uso extendido como `FlexibleContexts`, `FlexibleInstances`, etc
[2] Algunas extensiones como `GADTS` o incluso `TypeFamilies` ya existían en la época de la publicación original de AspectAG, pero eran experimentales, y de uso poco extendido.
[3] no es posible, por ejemplo la aplicación parcial

las anteriores nos permite escribir familias indizadas, como en los lenguajes dependientes. `TypeOperators` habilita el uso de simbolos operadores como constructores de tipos. El módulo `Data.Kind` exporta la notación `Type` para el kind `*`. Esto fué implementado originalmente con la extensión `TypeInType`, que en las últimas versiones del compilador es equivalente a `PolyKinds + DataKinds + KindSignatures`
Con todas extensiones combinadas, una declaración como

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

se "duplica" a nivel de kinds (`DataKinds`). 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`). Luego es posible declarar, por ejemplo (`GADTs`, `KindSignatures`)

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

Y funciones seguras como

```
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)
```

Es posible definir funciones puramene a nivel de tipos mediante familias (`TypeFamilies`, `TypeOperators`, `DataKinds`, `KindSignatures`) como la suma:

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

o mediante la notación alternativa, cerrada

```
type family (m :: Nat) + (n :: Nat) :: Nat where
  (+) Zero     a = a
  (+) (Succ a) b = Succ (a + b)
```

2.2 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* [10]. En las teorías de tipos intensionales una definición como la suma declarada anteriormente 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* (*Propositional Types*) [10].
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 de una familia de tipos 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, funciones como:

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

resultan imposibles de definirl 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 es incapaz de deducir.

2.3 Singletons y Proxies

Existen dos *hacks*, para resolver los problemas planteados anteriormente.

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 el vector. 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)
```

Existe por cada tipo `n` de kind `Nat`, un único [4] término de tipo `SNat n`.
Estamos en condiciones de 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 para saber cuantos elementos extraer, pero una función de tipo

```
vTake :: SNat m -> Vec (m :+ n) x -> Vec m x
```

no será implementable. Necesitamos información también de `n`, pero de hecho no es necesaria una representación de `n` en tiempo de ejecución. `n` es estático pero necesitamos que sea explícito.
Consideramos la definición:

[4] Formalmente esto no es cierto, si consideramos las posibles ocurrecias de $\bot$, la unicidad es cierta para términos totalmente definidos

```
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). 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 compila y funciona correctamente:

```
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)
```

durante la implementación de AspectAG y sus dependencias haremos uso intensivo de estas técnicas.

2.4 HList : Colecciones Heterogeneas Fuertemente tipadas

2.4.1 Listas Heterogeneas

En la biblioteca HList [8] 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, que no pretende ser utilizada como dependencia de desarrollos de producción por lo 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 corriente en poco tiempo.
- Cuando programamos a nivel de tipos el lenguaje no provee fuertes mecanismos de modularización, dado que no fue diseñado para este propósito. Es común que por ejemplo, se fugue implementación con los mensajes de error. La implementación basada en HList filtraría errores de HList, que no utilizan el mismo vocabulario que nuestro EDSL. La imposibilidad de modularizar nos obliga a que si pretendemos tener estructuras distinguibles por sus nombres mnemónicos tenemos que reimplementarlas. Buscar mejores soluciones a esta complicación es parte de la investigación en el área, e idea de trabajo futuro.
- HList no es necesariamente adecuada si queremos tipar todo lo fuertemente posible. Por ejemplo, en la implementación una de las estructuras a utilizar es esencialmente un registro que contiene registros. Usando tipos de datos a medida podemos programar una solución elegante donde esto queda expresado correctamente a nivel de kinds. La alternativa de implementar el registro externo como un registro de HList trata al interno como un tipo plano.
- Por interés académico. Reescribir funcionalidades de HList (de hecho, varias veces, un subconjunto mayor al necesario para AspectAG) fué la forma de dominar las técnicas de programación a nivel de tipos. 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 el hecho de que reimplementar no significa un costo: ya lo hicimos.

Una lista (o más en general una colección) heterogenea es tal si contiene valores de distintos tipos. Existen varios enfoques para construir colecciones heterogeneas en Haskell [15] Nos interesan en particular las que son fuertemente tipadas, donde se conoce estáticamente el tipo de cada miembro.
Existen variantes para definir HList Las versiones más antiguas (y sobre estas se implementó originalmente AspectAG) utilizan la siguiente representación, isomorfa a pares anidados:

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

El inconveniente de esta representación es que podemos construir tipos sin sentido como `HCons Bool Char`, lo cual puede solucionarse mediante el uso de clases, como es usual en el enfoque antigüo de la programación a nivel de tipos.
En versiones posteriores HList utilizó un GADT, y en las últimas versiones se utiliza una data Family. En la documentación de la biblioteca 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 solución más clara.

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

Es intuitivo definir funciones en este contexto, por ejemplo

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

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

Para, por ejemplo concatenar dos listas, primero 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)
```

Y luego 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 es usar la familia cerrada:

```
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)
```

Si intentemos, por ejemplo programar una función que actualiza la n-ésima entrada en una lista heterogenea (eventualmente cambiando el tipo del dato en esa posición), estamos claramente ante una función de tipos dependientes (el tipo de salida depende de n). Éste es el escenario donde serán necesarios `Proxies` y`øSingletons`.

```
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

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)
```

2.5 Registros Heterogeneos

AspectAG requiere de registros heterogeneos, esto es, colecciones etiqueta-valor, heterogeneas, donde además las claves estén dadas por tipos. El enfoque original de HList para implementarles es utilizar una lista Heterogenea, donde cada entrada era del tipo `Tagged l v`, definido como

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

Consideramos que no es satisfactorio si pretendemos utilizar todo el poder de las técnicas 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 directamente en el kind. Además las etiquetas de HList son de kind `Type`, cuando en realidad nunca requieren estar habitadas a nivel de valores.
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 registro es una lista con más estructura, a nivel de tipos es una lista de pares, usamos la promoción de listas y de pares a nivel de tipos. 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
```

Tagged es polimorfico en el kind de las etiquetas. La restricción Labelset garantiza que las mismas no se repitan, su implementación se explica a contunuación.

2.5.1 Predicados sobre tipos

Si bien las extensiones modernas nos permiten adoptar el estilo funcional en la programación a nivel de tipos, el estilo lógico sigue siendo adecuado para codificar predicados. Una lista promovida de pares satisface el predicado `LabelSet` si las primeras componentes son únicas. Así, la lista de pares representa un mapeo, o registro indizado por las primeras componentes. El predicado se implementa a la prolog, aunque usamos el poder de las extensiones modernas para tipar fuertemente los functores.

```
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
```

donde `HEqK l1 l2 False` es instancia solo si `l1` y `l2` son probables distintos. Para ello se utiliza la igualdad sobre kinds que implementa el módulo `Data.Type.Equality`.
Notar que podríamos también codificar el predicado como una función booleana a nivel de tipos, luego se propaga como una restricción con el valor que queremos. En general, la programación mediante clases es siempre sustituible por familias, pero no parece natural en este caso.

3 AspectAG

3.1 Gramáticas de atributos

Las gramáticas de atributos [9] son un formalismo para describir computaciones recursivas sobre tipos de datos[5]. Dada una gramática libre de contexto se le asocia una semántica considerando *atributos* en cada producción, los cuales toman valores calculados mediante reglas a partir de los valores de los atributos de los padres y de los hijos en el arbol de sintaxis abstracta. Los atributos se dividen clásicamente en dos tipos: heredados y sintetizados. Los atributos heredados son pasados como contexto desde los padres a los hijos. Los atributos sintetizados, por el contrario fluyen "hacia arriba" en la gramática, propagándose desde los hijos de una producción. Un *Aspecto* es una colección de (uno o más) aributos y sus reglas de cómputo.
Las gramáticas de atributos son especialmente interesantes en la implementación de compiladores [6] [1] traduciendo el arbol de sintaxis abstracta 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 ocurre en la mayoría de los languajes de programación modernos) o implementar verificadores de tipos.
Además, las gramáticas de atributos son útiles en si mismas como un paradigma de programación. Buena parte de la programación funcional consiste en componer computaciones sobre árboles (expresadas mediante álgebras [3], aplicadas a un catamorfismo). Un álgebra en definitiva especifica una semántica para una estructura sintáctica. Cuando las estructuras de datos son complejas (por ejemplo, en la representación del árbol de sintaxis abstracta de un lenguaje de programación complejo), surgen ciertas dificultades [6]. Por ejemplo ante un cambio en la estructura es necesario el retrabajo dado que cambia el catamorfismo y todas las álgebras. Las gramáticas de atributos buscan proveer una solución más escalable.
Más en general la programación mediante gramáticas de atributos significa 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 forma incremental es deseable que sea sencillo introducir nuevos tipos de datos o enriquecer los existentes con nuevos constructores, y también que sea simple implementar nuevas funciones. Normalmente en el diseño de un lenguaje las decisiones que facilitan una de las utilidades van 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 claro ejemplo donde es simple agregar funciones, siendo costoso en cada paradigma hacer lo dual. Pensar en cuan complicado (y cuantos módulos hay que modificar, y por tanto recompilar) 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 un constructor (y nuevamente, cuántos módulos se deben recompilar).
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 la nueva producción, así como agregar nuevas funcionalidades (definiendo localmente nuevos atributos con sus reglas, o bien combinando los ya existentes).
Por su característica, 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, como veremos en el ejemplo de la próxima sección.

[5] Tipos de datos algebráicos o gramáticas son formalismos equivalentes

3.2 Ejemplo: `repmin`

Como ejemplo consideramos la clásica función **`repmin`** [2] que dado un árbol contenedor de enteros (por ejemplo binario y 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 computarán 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 el mínimo local es global y computar `ival`. En los nodos, a cada subárbol se le copia el valor de `ival` actual, que fluirá "hacia" abajo.

3.3 AspectAG

AspectAG es un lenguaje de dominio específico embebido en haskell que implementa gramáticas de atributos y explota el sistema de tipos para chequear estáticamente ciertas propiedades de buena formación.
La implementación sigue a grandes razgos la siguiente idea: Dada una estructura de datos sobre la que vamos a definir los atributos, 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 2.5 es adecuada. 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*. Una regla semántica consiste en un mapeo de una input family a una output family. Se le proveen al programador primitivas para definir atributos y sus reglas semánticas, que agrupará en aspectos. Se provee también una primitiva para combinar aspectos. Una función (el catamorfismo) se encarga de combinar las reglas semánticas para efectivamente producir una recorrida sobre la estructura de datos.
Notar que una especificación de gramática de atributos podría estar mal formada (por ejemplo, al intentar usar atributos que no están definidos para cierta producción). Como las atribuciones se conocen estáticamente, los ejemplos mal formados serán rechazados en tiempo de compilación.
Presentamos una solución al problema `repmin` en la reimplementación del EDSL, para que el lector tome contacto con el estilo de programación en el mismo. Luego se presentará mayor detalle de la implementación.
Hay que definir múltiples *Etiquetas*[6]. Hay etiquetas para los símbolos no terminales, para los atributos, y para nombrar a los hijos de cada producción. Por ejemplo, para los atributos definimos:

```
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 tienen información solo a nivel de tipos, `Label` es una implementación especializada de `Proxy`.

```
data Label l = Label
```

Bajo la extensión `PolyKinds` la declaración anterior tiene kind `k -> *`. En nuestra implementación todos los registros extensibles son polimórficos en el kind de los índices, por lo cual también sería posible definir tipos de datos para cada tipo de etiqueta y utilizar el kind promovido.
Presentamos las reglas para el atributo `smin`. En la especificación de la gramática de atributos, puede observarse que tiene el atributo tiene reglas de computación en el árbol, por lo que hay dos producciones donde hace falta definirlas (`Node` y `Leaf`). En AspectAG:

[6] La biblioteca provee funciones de templateHaskell para ahorrarnos el trabajo

```
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 definió un atributo sintetizado `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` (nombre del hijo derecho). En el caso de la hoja, se toma el valor de `leafVal` (que es un nombre para el valor guardado) para el (único) hijo en la producción `ch_i`. Si bien generalmente los terminales van a tener un único hijo con su valor -parecen innecesarias dos etiquetas- 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). Las expresiones (funciones) definidas tienen tipo `Rule`(4.1).
Análogamente se define el atributo sintetizado `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)
```

En este caso el atributo estaba definido para la raíz, y en la hoja usamos un atributo sintetizado para computarle (el mínimo global).
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
```

Declaramos la definición de un atributo heredado llamado `ival`. La función requiere un registro donde se especifica para cada hijo cómo se computará `ival`: desde la raíz se propaga el valor del atributo sintetizado `smin`, en los nodos del árbol se propaga el valor de `ival` tomado del padre, incambiado. El parámetro extra `[nt_Tree]` es una lista de no terminales utilizada para hacer ciertos chequeos estáticos, 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
```

El operador `(.*.)` concatena regitros heterogeneos, `emptyRecord` es el registro vacío. El operador `(.=.)` construye campos del registro dados un valor con el tipo de la etiqueta y su correspondiente regla.
Los aspectos pueden combinarse para formar uno nuevo 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 el catamorfismo, una función definida una sola vez[7].

[7] El catamorfismo es derivable a partir del functor de la estructura de datos. Al momento de la escritura de este documento el programador debe proveer el catamorfismo, pero es uno de los objetivos inmediatos de trabajo futuro automatizar la generación del mismo.

4 Reimplementación de AspectAG

4.1 Estructuras de Datos

Como se definió antes, una atribución (*attribution*) es un mapeo de nombres de atributos (represenentados puramente a nivel de tipos) a sus valores. La estructura de registro heterogeneo 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 (`DataKinds`), polimorfismo en kinds en las etiquetas (`PolyKinds`) la estructura es un GADT (`GADTs`), LabelSet está predicando sobre un kind polimórfico (por lo que usamos igualdad de kinds)(`ConstraintKinds`), y el kind `Type` fué introducido en `TypeInType`.
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)
```

La estructura es análoga a la anterior: es de nuevo una implementación de Registro heterogeneo especializada. Dado que una atribución una vez bajo el wrapper `Attribution` tiene kind `Type` (como todo tipo habitado) podríamos haber implementado a los hijos como un registro agnóstico respecto al contenido. Se prefiere una implementación fuertemente tipada (a nivel de kinds, queda ahora explícito que tenemos un registro de registros) 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 [11].

```
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 primitivas para declarar reglas. En el ejemplo se utilizaron `syndef` e `inhdef`, que son las mínimas adecuadas para tener un sistema utilizable. En la implementación se proveen otras construcciones, y parte del trabajo futuro pasa por codificar nuevas.
La primitiva `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)
```

Los operadores `(=.)` y `(*.)` son azucar sintáctica para El cunstructor de atributos `Attribute` y para `ConsAtt`, respectivamente.
Como ejemplo de una primitiva alternativa, podemos citar `synmod`, que redefine un atributo sintetizado existente.

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

Por otro lado, la función `inhdef` introduce un atributo heredado de nombre `att` para una colección de no terminales `nts`.

```
inhdef :: Defs att nts vals ic ic'
  => Label att -> HList nts -> Record vals -> (Fam ic sp -> Fam ic' sp)
inhdef att nts vals (Fam ic sp) = Fam (defs att nts vals ic) sp
```

`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, y para ello utilizmos la función auxiliar `defs`, que inserta las definiciones en los hijos a los que corresponda.
Primero, es necesario proveer la inserción 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
```

Los parámetros `mch` y `mnts` son booleanos a nivel de tipos que proveen evidencia de la existencia de las etiquetas del hijo o los terminales. Como se detalló en la sección 2.3, en Haskell requerimos testigos en tiempo de ejecución en computaciones de tipos dependientes, por lo que debemos incluir los parámetros `Proxy`.
Finalmente estamos en condiciones de definir la función `defs`. Se hace recursión sobre el registro `vals` que contiene las nuevas definiciones. Utilizanando `singledef` se insertan las definiciones.

```
class Defs att (nts :: [Type]) (vals :: [(k,Type)])
           (ic :: [(k,[(k,Type)])]) (ic' :: [(k,[(k,Type)])])
          | att nts vals ic -> ic' where
  defs :: Label att -> HList nts -> Record vals -> ChAttsRec ic
       -> ChAttsRec ic'
```

Si el registro está vacío no se inserta ninguna definición:

```
instance Defs att nts '[] ic ic where
  defs _ _ _ ic = ic
```

En el caso recursivo, combinamos la primer componente del registro con la llamada recursiva.

```
instance ( Defs att nts vs ic ic'
         , HasLabelChildAttsRes (lch,t) ic' ~ mch
         , HasLabelChildAtts (lch,t) ic'
         , HMemberRes t nts ~ mnts
         , HMember t nts
         , SingleDef mch mnts att (Tagged (lch,t) vch) ic' ic'')
    => Defs att nts ( '((lch,t), vch) ': vs) ic ic'' where
  defs att nts (ConsR pch vs) ic
    = let ic'  = defs att nts vs ic         -- :: ChAttsRec ic'
          lch  = labelLVPair pch            -- :: Label (lch,t)
          mch  = hasLabelChildAtts lch ic'  -- :: Proxy mch
          mnts = hMember (sndLabel lch) nts -- :: Proxy mnts
      in singledef mch mnts att pch ic'
```

En todos los casos las funciones auxiliares utilizadas en la cláusula `let` son funciones de acceso o predicados relativamente simples de definir. Nuevamente se retornan valores necesarios para tomar decisiones a nivel de tipos, porque la función es de tipos dependientes. `lch`, `mch`, `mnts` no tienen información a nivel de valores. Por ejemplo `hasLabelChildAtts` predica la existencia de una etiqueta de hijo a nivel de tipos. Se define como familia de tipos y a nivel de valores y se retorna un proxy.

```
class HasLabelChildAtts (e :: k)(r :: [(k,[(k,Type)])]) where
  type HasLabelChildAttsRes (e::k)(r :: [(k,[(k,Type)])]) :: Bool
  hasLabelChildAtts
   :: Label e -> ChAttsRec r -> Proxy (HasLabelChildAttsRes e r)

instance HasLabelChildAtts e '[] where
  type HasLabelChildAttsRes e '[] = 'False
  hasLabelChildAtts _ _ = Proxy

instance HasLabelChildAtts k ( '(k' ,v) ': ls) where
  type HasLabelChildAttsRes k ( '(k' ,v) ': ls)
      = Or (k == k') (HasLabelChildAttsRes k ls)
  hasLabelChildAtts _ _ = Proxy
```

Notar que el contenido computacional de la definición existe solo a nivel de tipos, a nivel de valores la función tan solo propaga evidencia.
`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
```

4.3 Aspectos

Un aspecto se implementa simplemente como un registro heterogeneo (que contendrá reglas etiquetadas por nombres de producciones). En este caso no consideramos necesaria une implementación especializada y preferimos reutilizar código, en contraste a la implementación de las atribuciones y registros de atribuciones de los hijos, en donde la estructura es compleja e induce a errores de programación (por lo que preferíamos tipar lo más fuertemente posible). 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.

```
type Aspect = Record
type Prd prd rule = Tagged prd rule
```

4.4 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
```

Nuevamente hacemos uso del proxy para propagar pruebas. `hasLabelRec` se define análogamente a `hasLabelChildAtts`. 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
```

4.5 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 `repmin`, 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.

4.6 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 Conclusión

Reimplemetamos un subconjunto de la biblioteca AspectAG original utilizando las técnicas modernas de programación a nivel de tipos. Con ello hicimos fuertemente tipadas las dos capas de programación, a diferencia de la biblioteca original, que a nivel de tipos era esencialmente no tipada.
A modo de ejemplo, el constructor de tipos `Fam` en las versiones previas de *AspectAG* tiene *kind*

```
Fam :: * -> * -> *
```

Notar que `Fam Bool Char` es un tipo válido Peor aún, el constructor (de valores) `Fam` tiene tipo

```
Fam :: c -> p -> Fam c p
```

por lo que el tipo patológico Fam Bool Char está habitado.
En la nueva implementación `Fam` tiene *kind*

```
Fam :: [(k', [(k, Type)])] -> [(k, Type)] -> Type
```

```
Fam :: forall k' k (c :: [(k', [(k, Type)])]) (p :: [(k, Type)]).
       ChAttsRec c -> Attribution p -> Fam c p
```

Lo cual es mucho más expresivo. Notar que de todas formas el tipo `Fam` a priori no expresa la especificación completa del tipo de datos (lo que podríamos lograr en un lenguaje de tipos verdaderamente dependientes): en ninguna parte se declara que las listas de pares son en realidad mapeos donde las primeras componentes (las etiquetas) no se repiten. Sin embargo esto está garantizado porque los constructores de `Attribution` y `ChAttsRec` tienen restricciones.
Notar por ejemplo el constructor `ConsAtt`:

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

Nuestro sistema asegura que todos los valores de tipo `Fam` van a estar bien formados.[8]
La implementación de estructuras especializadas provee nombres mnemónicos para las estructuras, comparemos el tipo de `asp_sres` en la implementación antigüa, y en la nueva:

```
asp_sres
  :: (HExtend (Att (Proxy Att_sres) val) sp1 sp'1,
      HExtend (Att (Proxy Att_sres) Tree) sp2 sp'2,
      HExtend (Att (Proxy Att_sres) Tree) sp3 sp'3,
      HasField (Proxy (Ch_tree, Tree)) r1 r2,
      HasField (Proxy (Ch_l, Tree)) r3 r4,
      HasField (Proxy (Ch_r, Tree)) r3 r5,
      HasField (Proxy Att_sres) r2 val,
      HasField (Proxy Att_sres) r4 Tree,
      HasField (Proxy Att_sres) r5 Tree,
      HasField (Proxy Att_ival) r6 Int) =>
  Record
  (HCons(LVPair (Proxy P_Root) (Fam r1 p1 -> Fam ic1 sp1 -> Fam ic1 sp'1))
  (HConS(LVPair (Proxy P_Node) (Fam r3 p2 -> Fam ic2 sp2 -> Fam ic2 sp'2))
  (HCons(LVPair (Proxy P_Leaf) (Fam c r6  -> Fam ic3 sp3 -> Fam ic3 sp'3))
   HNil)))
```

```
asp_sres
  :: (HasChild (Ch_tree, Tree) r1 v1, HasChild (Ch_l, Tree) r2 v2,
      HasChild (Ch_r, Tree) r2 v3,
      LabelSet ( '(Att_sres, val)  : sp1),
      LabelSet ( '(Att_sres, Tree) : sp2),
      LabelSet ( '(Att_sres, Tree) : sp3),
      HasFieldAtt Att_sres v1 val,
      HasFieldAtt Att_sres v2 Tree,
      HasFieldAtt Att_sres v3 Tree,
      HasFieldAtt Att_ival r3 Int) =>
  Record
  '['(P_Root, Fam r1 p1 -> Fam ic1 sp1 -> Fam ic1 ('(Att_sres, val) : sp1)),
    '(P_Node, Fam r2 p2 -> Fam ic2 sp2 -> Fam ic2 ('(Att_sres, Tree) : sp2)),
    '(P_Leaf, Fam c r3 -> Fam ic3 sp3 -> Fam ic3 ('(Att_sres, Tree) : sp3))]
```

[8] O casi: aún es posible construir valores como `undefined` o `Fam undefined undefined` de cualquier tipo construido con `Fam`. Estos valores además de ser patológicos (a nivel de términos), como no usan los constructores nos permiten saltearnos la restricción de la typeclass y repetir etiquetas, y por tanto construir instancias patológicas a nivel de tipos. Esto último podría evitarse agregando las restricciones de instancias de `LabelSet` en el constructor `Fam`, no lo consideramos necesario.

Por otra parte, los mensajes de error han sido mejorados. Por ejemplo, consideramos en el ejemplo `repmin` [REF] que omitimos en la definición de `asp_smin` la regla `p_Node .=. node_smin`. La gramática estará mal formada y la función `repmin`, o `getmin` no compilarán.
En las versiones antigüas obtenemos el críptico error:

```
    . No instance for
  (HasField (Proxy P_Node) HNil
   (Fam (Record (HCons (Chi (Proxy (Ch_l, Tree))
                            (Record (HCons (LVPair (Proxy Att_smin) Int) HNil)))
                       (HCons (Chi (Proxy (Ch_r, Tree))
                                   (Record (HCons (LVPair (Proxy Att_smin) Int) HNil)))
                        HNil)))
     (Record HNil)
   -> Fam (Record (HCons (Chi (Proxy (Ch_l, Tree)) (Record HNil))
                         (HCons (Chi (Proxy (Ch_r, Tree)) (Record HNil)) HNil)))
      (Record HNil)
   -> Fam (Record (HCons (Chi (Proxy (Ch_l, Tree)) (Record HNil))
                         (HCons (Chi (Proxy (Ch_r, Tree)) (Record HNil)) HNil)))
     (Record (HCons (LVPair (Proxy Att_smin) Int) HNil))))
    arising from a use of 'sem_Tree'
       ....
```

En la reimplementación:

```
    . Type Error : No Field found on Aspect.
      A Field of type P_Node was expected
     (Possibly there are productions where the attribute is undefined)
       ....
```

Además de tipar fuertemente, sustituimos parte de la programación a nivel de tipos al estilo lógico utilizado con las técnicas antigüas por un estilo funcional gracias a las type families, más idiomático para una biblioteca implementada en Haskell. Por ejemplo, en las distintas versiones de la biblioteca original la función `empties` viene dada por una relación entre tipos que se declara funcional con una dependencia:

```
class Empties fc ec | fc -> ec where
  empties :: fc -> ec
```

En contraste, en la nueva implementación `EmptiesR` es explícitamente una función a nivel de tipos de kind

```
EmptiesR :: [(k, *)] -> [(k, [(k, *)])]
```

Y `empties` una función a nivel de valores, de tipo

```
empties :: Record fc -> ChAttsRec (EmptiesR fc)
```

Ésta cuestión era planteada como trabajo futuro en la publicación original [13] de la biblioteca. Aún puede ser mejorado en ciertas porciones de código.
Además de las características que tiene el sistema, se provee una demostración del uso de la programación a nivel de tipos en Haskell enriquecido por el conjunto de extensiones que implementa el compilador GHC.

6 Trabajo Futuro

En este proyecto, se reimplementó un subconjunto de la biblioteca original; es natural fijarnos como objetivo a futuro, tener una versión completa de la biblioteca implementada con las técnicas modernas para publicar. Continua abierta la cuestión planteada en [13] sobre la implementación de un sistema análogo en un lenguaje de tipos dependientes.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

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

[3] Richard Bird and Oege de Moor. The algebra of programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design*, pages 167–203, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[4] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. *SIGPLAN Not.*, 40(9):241–253, September 2005.

[5] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. *SIGPLAN Not.*, 40(1):1–13, January 2005.

[6] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The architecture of the utrecht haskell compiler. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 93–104, New York, NY, USA, 2009. ACM.

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

[8] 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.

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

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

[11] Oege De Moor. First-class attribute grammars. *Informatica*, 24:2000, 1999.

[12] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. 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] Haskell Wiki. Heterogeneus collections. https://wiki.haskell.org/Heterogenous_collections. Accedido: 31-10-2018.

[16] 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.