Reimplementación de *AspectAG* basada en nuevas extensiones de Haskell

Juan García Garland

January 17, 2019

Contents

1 Introducción

AspectAG [16] es un EDSL (lenguaje de dominio específico, embebido) desarrollado en Haskell que permite la construcción 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 [11], 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 manipulación a nivel de tipos).
Desde el momento de la implementación original de AspectAG hasta la actualidad la programación 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 difícil 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 se presenta una breve reseña de las técnicas de programación a nivel de tipos y las extensiones a Haskell que provee el compilador GHC que las hacen posibles. Se presentan las estructuras de listas heterogeneas (2.5) y registros heterogeneos (2.6) que normalmente no serían implementables en un lenguaje fuertemente tipado sin tipos dependientes.
En la sección 3 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 4 se presentan los detalles de la implementación, que se basan en las técnicas de programación a nivel de tipos 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 de tests se implementan ejemplos de utilización de la biblioteca.

2 Programación a nivel de tipos en GHC Haskell

2.1 Extensiones utilizadas

2.1.1 Técnicas antiguas

La biblioteca AspectAG presentada originalmente en 2009, además de implementar un sistema de gramáticas de atributos como un EDSL provee un buen ejemplo del uso de la programación a nivel de tipos en Haskell. La implementación utiliza fuertemente los registros extensibles que provee la biblioteca HList. Ambas bibliotecas se basan en la combinación de las extensiones `MultiParamTypeClasses` (hace posible la implementación de relaciones a nivel de tipos) con `FunctionalDependencies` [10] (que permite expresar en particular relaciones funcionales).[1]

2.1.2 Técnicas modernas

Durante la década pasada [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] [15] 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` [19] 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 Haskell cuenta con un lenguaje a nivel de tipos casi tan expresivo como a nivel de términos[3]. La extensión `GADTs` combinada con las anteriores nos permite escribir familias indizadas, como en los lenguajes dependientes. `TypeOperators` habilita el uso de 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`

2.2 Programando con tipos dependientes en Haskell

Con todas estas extensiones combinadas, una declaración como

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

se "duplica" a nivel de kinds (`DataKinds`). Esto es, que 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

[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

```
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)
```

2.3 Limitaciones

En contraste a lo que ocurre en los sistemas de 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` en 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* [13]. 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*) [13].
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 poseen 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 el compilador 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 definir 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
```

tendría un problema más sutil. Incluso asuminedo que tuvieramos forma de obtener `m` en tiempo de ejecución, no es posible para el verificador de tipos aceptar la definición. 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.

2.4 Singletons y Proxies

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

2.4.1 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 ∀ solo habla de objetos estáticos (los lenguajes de tipos y términos están separados), esto no es posible directamente. Un tipo *singleton* [8] 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)
```

Existe una biblioteca que implementa la generación automática de instancias y otras utilidades [9].

2.4.2 Proxies

Análogamente para definir `vTake` es necesario el valor de `m` en tiempo de ejecución para conocer 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 estamos obligados a que sea un testigo explícito para asistir al verificador de tipos que es incapaz de deducir la inyectividad de la suma.
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). 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.

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

2.5 HList : Colecciones Heterogeneas Fuertemente tipadas

La biblioteca HList [11] provee operaciones para crear y manipular colecciones heterogeneas fuertemente tipadas (donde el largo y el tipo de los elementos se conocen en tiempo de compilación). Éstas son útiles para implementar registros heterogeneos (extensibles), variantes, productos y coproductos indizados por tipos, entre otras estructuras. HList es un buen ejemplo de aplicación de la 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 desarrollándose a medida de que nuevas extensiones se añaden al lenguaje Haskell. En lugar de reimplementar AspectAG dependiendo de nuevas versiones de HList decidimos reescribir desde cero todas las funcionalidades necesarias, por los siguientes 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 desactualizada (e incompatible con la liberación 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 información en los mensajes de error sobre las estructuras de datos utilizadas. 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 [18]. Nos interesan en particular las implementaciones que son fuertemente tipadas, donde se conoce estáticamente el tipo de cada miembro.
Existen variantes para definir HList. Las versiones más antiguas (sobre las que 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 implementación es que es posible 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 fundamenta 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.
El tipo de datos `HList` tiene la siguiente definición:

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

La extensión `DataKinds` promueve las listas con una notación conveniente similar a la utilizada a nivel de valores. En la definición anterior se utiliza la versión promovida de listas como índice del tipo de datos. `HNil` es un valor de tipo `HList '[]`, mientras que `HCons` construye un valor de tipo `HList (x ': xs)` a partir de un valor de tipo `x` y una lista de tipo `HList xs`.
A modo de ejemplo, un habitante posible del kind `[Type]` es `'[Bool, Char]`. Luego `HList [Bool, Char]` es un tipo (de kind `Type`) habitado por ejemplo por `HCons True (HCons 'c' HNil)`.
Es intuitivo definir, por ejemplo las versiones seguras de `head` o `tail`:

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

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

No es posible compilar expresiones como `hHead HNil`, dado que el verificador de tipos de GHC inferirá que es imposible satisfacer la restricción `(x ': xs)` $\sim$ `'[]`.
Para 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 definir la familia como un tipo indizado: [5]

```
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 intentamos a modo de 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 del resultado depende de n). Éste es el escenario donde serán necesarios `Proxies` y/o `Singletons`.
Una definición posible: [6]

```
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)
```

[5] Ésta es una tercer forma de definir familias de tipos, además de la notación abierta o cerrada.
[6] asumamos que el n es menor al largo de la lista, lo cual podríamos también forzar estáticamente

2.6 Registros Heterogeneos Extensibles

AspectAG requiere de registros heterogeneos extensibles, esto es, colecciones etiqueta-valor, heterogeneas, donde además las etiquetas estén dadas por tipos. Además de implementarles mediante el uso de HList, existen otros trabajos sobre los mismo en Haskell, como Vynil [REF], CTRex [REF], etc [TODO].
El enfoque original de HList para implementar registros es utilizar una lista heterogenea, donde cada entrada es del tipo `Tagged l v`, definido como

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

Esta definición no es satisfactoria si pretendemos utilizar todo el poder de las nuevas extensiones de Haskell. Por ejemplo no se está utilizando la posibilidad de tipar (en el lenguaje de tipos) que nos provee la promoción de datos. HList implementa predicados como typeclasses para asegurar que todos los miembros son de tipo `Tagged` cuando esto 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, utilizaremos 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 polimórfico en el kind de las etiquetas. La restricción `Labelset` garantiza que las etiquetas no se repitan, su implementación se explica a continuación.

2.6.1 Predicados sobre tipos

Si bien las extensiones modernas nos permiten adoptar el estilo funcional en la programación a nivel de tipos, el estilo de programación lógica sigue siendo adecuado para codificar predicados. Una lista de pares promovida 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. También podría utilizarse una familia de tipos para construir la constraint `LabelSet` (haciendo uso de la extensión `ConstraintKinds`).
En general, la programación mediante clases es siempre sustituible por familias de tipos, pero no parece natural en este caso.

3 Gramáticas de atributos y AspectAG

3.1 Gramáticas de atributos

Las gramáticas de atributos (AGs) [12] son un formalismo para describir computaciones recursivas sobre tipos de datos[7]. 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 árbol 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 [1, 7] 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 [7]. Por ejemplo ante un cambio en la estructura es necesario hacer cambios en la implementación del catamorfismo y de todas las álgebras.
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 [17]). 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 algebraico se agrega un constructor (y nuevamente, cuántos módulos se deben recompilar).
La *Programación orientada a aspectos* mediante gramáticas de atributos es una propuesta de solución a este problema, donde es 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.

[7] 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.
Una definición por gramáticas de atributos viene dada de la siguiente 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*

Utilizamos la sintaxis de Utrecht[TODO:REF] Las palabras clave `DATA` introducen tipos de datos, el ejemplo traducido a haskell corresponde a:

```
data Root = Root Tree

data Tree = Node { l, r :: Tree}
          | Leaf {    i :: Int}
```

En lugar de utilizar el tipo `Tree` tanto para nodos internos como para la raíz, como es usual, utilizamos el tipo de datos `Root` para ‘marcar‘ la raíz del árbol Ésto es útil para tener un símbolo de inicio explícito de la gramática, que a nivel operacional nos va a permitir saber cuando encadenar atributos sintetizados con heredados. La decisión es natural dado que la semántica (i.e. la forma en la que 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 sintetizados de tipo `Int` y `Tree` respectivamente. La semántica de cada uno se define mediante de la sentencia `SEM`. `smin` representa en el mínimo valor de las hojas contenidas en el subárbol correspondiente, computándose en las hojas como el valor que ellas contienen (que, formalmente puede considerarse un nuevo atributo, implícito), y en los nodos el mínimo del valor del mismo atributo `smin` en los subárboles.
El atributo `sres` vale un árbol con la misma forma que el subárbol original pero con el mínimo global en cada hoja. En la raíz se copia el subárbol, y en cada nodo iterno 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 permite especificar gramáticas de atributos y utiliza programación a nivel de tipos para que las gramáticas resultantes verifiquen estáticamente ciertas propiedades de buena formación.
La implementación sigue a grandes rasgos la siguiente idea: Dada una estructura de datos sobre la que vamos a definir los atributos, en cada producción de la gramática llamamos *atribución* al registro de todos los atributos. Una atribución es un mapeo de nombres de atributos a sus valores. Los nombres de atributos se manejan en tiempo de compilación, por lo que una estructura como la presentada en la sección 2.6 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 se agruparán en *aspectos*. Se provee también una primitiva para combinar aspectos. Una función (la función semántica) se encarga de computar la AG, es decir, calcular el valor de los atributos definidos en las producciones.
La especificación de una AG 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 términos de nuestra reimplementación de AspectAG, para que el lector tome contacto con el estilo de programación en el EDSL. Luego se presentará mayor detalle de la implementación.
Es necesario definir múltiples tipos de *Etiqueta* [8]. Hay etiquetas para los símbolos no terminales, para los atributos, y para nombrar a los hijos en 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 :: k) = Label
```

Nótese que en el ejemplo los tipos definidos para cada etiqueta son vacíos (no tienen habitantes), su única función es ser usados como parámetro en `Label`.
Veamos ahora las reglas para el atributo `smin`. En la especificación de la gramática de atributos, puede observarse que este atributo tiene reglas de computación en el árbol, por lo que hay dos producciones donde hace falta definirlas (`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
```

Para comprender la definición anterior recordemos que las reglas son 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). El parámetro sobre el que se hace pattern matching (`Fam chi par`) es entonces una familia, el campo `chi` contiene el registro de los hijos, con sus atribuciones, y el campo `par` la atribución del padre. Las funciones definidas tienen tipo `Rule`. Los tipos `Fam` y `Rule` se detallan en la sección 4.1.
La definición anterior expresa que para el nodo se define una regla para el 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 regla para 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 de forma

[8] La biblioteca provee funciones en TemplateHaskell para ahorrarnos el trabajo

fuertemente tipada nos obliga a respetar esta estructura (o complicar mucho la implementación). Todos los nombres (`ch_l`, `ch_r`, `leafVal`, `ch_i`) son etiquetas definidas de forma análoga a las etiquetas para atributos. Notar que, además de los nombres, nada indica (aún) que las reglas estén relacionadas a sus producciones. Análogamente se definen las reglas para 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 heredado `ival` para computarle (el mínimo global).
Por último presentamos el atributo heredado:

```
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
```

La función `inhdef` 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[9].
Los aspectos se definen como un registro con las reglas para cada producción (aquí es donde efectivamente asociamos a qué producción se asocia cada regla).

```
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 `(.*.)` permite construir 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 la función semántica, una función definida una sola vez[10].

[9] éste parámetro no es relevante para ninguna de las funcionalidades que se implementaron en este proyecto, se incluye para mantener la interfaz dado que será útil cuando éstas sí se implementen en el futuro

[10] La función semántica es derivable a partir del functor de la estructura de datos. Al momento de la escritura de este documento el programador debe proveerla, pero es uno de los objetivos inmediatos de trabajo futuro automatizar la generación de la misma.

4 Reimplementación de AspectAG

A continuación presentamos algunos de los aspectos más importantes de la implementación de la biblioteca.

4.1 Estructuras de Datos

Como se definió antes, una atribución (*attribution*) es un mapeo de nombres de atributos a sus valores, que representamos como un registro heterogeneo. Para obtener mensajes de error precisos y evitar que se filtre implementación en los mismos decidimos tener estructuras especializadas.
Un atributo (etiquetado por su nombre) viene 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 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, donde usamos igualdad de kinds (`ConstraintKinds`).
Una familia consiste en la atribución del padre y una colección de atribuciones para los hijos (etiquetadas por sus nombres).

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

La colección de atribuciones para los hijos son representadas de la siguiente manera:

```
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 pero especializada.
Una regla es una función de la familia de entrada a la de salida. Se implementa de la siguiente manera:

```
type Rule sc ip ic sp ic' sp'
  = Fam sc ip -> (Fam ic sp -> Fam ic' sp')
```

El tipo contiene una aridad extra para hacer las reglas componibles [14]. Dada la familia de entrada (compuesta por atributos sintetizados de los hijos (`sc`) y los heredados del padre (`ip`)) se contruye una función que toma como entrada la familia de salida construida hasta el momento (formada por los atributos heredados de los hijos (`ic`) y los sintetizados por el padre (`sp`)), y la extiende (donde los nuevos atributos heredados de los hijos son `ic'` y los sintetizados por el padre `sp'`).
El kind de `Rule` viene dado entonces por:

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

Dos reglas se construidas de esta forma se pueden componer:

```
(f `ext` g) input = f input . g input
```

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 nuevo atributo a definir y un valor para el mismo, `syndef` construye una función que actualiza la familia que se tenía 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 los constructores `Attribute` y `ConsAtt`, respectivamente.
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 heterogeneo donde las etiquetas representan 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 utilizamos la función auxiliar `defs`, que inserta las definiciones en los hijos a los que corresponda.
Para definir la función `defs` se hace recursión sobre el registro `vals` que contiene las nuevas definiciones. Se utiliza la función auiliar `singledef` que inserta una única definición.

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

Si el registro está vacío no se inserta ninguna definición:

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

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

```
instance ( Defs att nts vs ic
         , ic' ~ DefsR att nts vs ic
         , HMember t nts
         , HMemberRes t nts ~ mnts
         , HasLabelChildAttsRes (lch,t) ic' ~ mch
         , HasLabelChildAtts (lch,t) ic'
         , SingleDef mch mnts att (Tagged (lch,t) vch) ic') =>
  Defs att nts ( '((lch,t), vch) ': vs) ic where
  type DefsR att nts ( '((lch,t), vch) ': vs) ic
    = SingleDefR (HasLabelChildAttsRes (lch,t) (DefsR att nts vs ic))
                 (HMemberRes t nts)
                 att
                 (Tagged (lch,t) vch)
                 (DefsR att nts vs ic)
  defs att nts (ConsR pch vs) ic = singledef mch mnts att pch ic'
```

```
      where ic'  = defs att nts vs ic
            lch  = labelLVPair pch
            mch  = hasLabelChildAtts lch ic'
            mnts = hMember (sndLabel lch) nts
```

`singledef` implementa la inserción en la atribución de un hijo: La definición de la clase utilizada para la función viene dada por:

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

Los primeros dos parámetros son Proxys de booleanos, que proveen evidencia de la existencia de las etiquetas del hijo y los terminales respectivamente. El caso válido, en donde el código debe compilar es cuando estos valen `True`, y la implementación es la siguiente:

```
instance ( HasChildF lch ic
         , och ~ LookupByChildFR lch ic
         , UpdateAtChildF lch ( '(att,vch) ': och) ic
         , LabelSet ( '(att, vch) ': och)) =>
  SingleDef 'True 'True att (Tagged lch vch) ic where
  type SingleDefR 'True 'True att (Tagged lch vch) ic
    = UpdateAtChildFR lch ( '(att,vch) ': (LookupByChildFR lch ic)) ic
  singledef _ _ att pch ic
    = updateAtChildF (Label :: Label lch) ( att =. vch *. och) ic
    where lch = labelTChAtt pch
          vch = unTaggedChAtt pch
          och = lookupByChildF lch ic
```

En todos los casos las funciones auxiliares utilizadas en las cláusulas `where` son funciones de acceso o predicados relativamente simples de definir. Algunos de los valores que se retornan contienen información a nivel de tipos usada en tiempo de compilación. `lch`, `mch`, `mnts` son ejemplos de expresiones que 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 todo el contenido computacional se da a nivel de los tipos, a nivel de valores 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)
```

La implementación es inmediata, buscamos la ocurrencia de la etiqueta en las primeras componentes de una lista de pares:

```
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
```

Por otra parte, `Or` es una función definida puramente a nivel de tipos, y la implementamos como familia.

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

Como ejemplo de una primitiva no usada en `repmin` podemos citar `synmod`, que redefine un atributo sintetizado existente (Tanto el valor, como el tipo).

```
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)
```

4.3 Aspectos

Un aspecto se implementa simplemente como un registro heterogeneo (que contendrá reglas etiquetadas por nombres de producciones).

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

En este caso no consideramos necesaria una 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.

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)]) (s :: [(k, Type)]) where
  type (.++.) r s :: [(k,Type)]
  (.+.) :: Aspect r -> Aspect s -> Aspect (r .++. s)
```

La función se define por recursión en la segunda componente. Si una producción aparece en un solo aspecto parámetro aparecerá en el resultado intacta. Por otro lado las producciones que aparezcan en ambos aspectos deberán incluirse en el resultado con las reglas combinadas (según la función `ext` definida previamente). Para el caso base, donde el segundo aspecto es vacío la función retorna el primer aspecto.

```
instance Com r '[] where
  type r .++. '[] = r
  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 (ComSingleR (HasLabelRecRes prd r) prd rule r)  r'
         , HasLabelRecRes prd r ~ b
         , HasLabelRec prd r
         , ComSingle b prd rule r)
  => Com r ( '(prd, rule) ': r') where
     type r .++. ( '(prd, rule) ': r')
       = (ComSingleR (HasLabelRecRes prd r) prd rule r) .++. r'
     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)]) where
  type ComSingleR b prd rule r :: [(k, Type)]
  comSingle :: Proxy b -> Prd prd rule -> Aspect r
            -> Aspect (ComSingleR b prd rule r)
```

Nuevamente hacemos uso del proxy para propagar pruebas. La función `hasLabelRec` se define análogamente a `hasLabelChildAtts`. Luego podemos definir las instancias posibles de `ComSingle`, y además chequeamos ciertas propiedades de buena furmación. En particular cuando combinamos reglas chequeamos que efectivamente estamos combinando reglas.
En el caso donde la etiqueta de la producción a insertar no está presente en el aspecto parámetro simplemente extendemos el aspecto[11]

```
instance ( LabelSet ('(prd, rule) ': r)) =>
  ComSingle 'False prd rule r where
  type ComSingleR 'False prd rule r = '(prd, rule) ': r
  comSingle _ prd asp = prd .*. asp
```

Si la producción ya existía en el aspecto parámetro, el algoritmo (informalmente) consiste en obtener la regla correspondiente, combinarla con la nueva (`ext`), e insertar la regla resultante en el aspecto.

```
instance ( UpdateAtLabelRecF prd (Rule sc ip ic  sp  ic'' sp'') r
         , HasFieldRec prd r
         , LookupByLabelRec prd r ~ (Rule sc ip ic' sp' ic'' sp'')
         , ic'' ~ (Syn3 (LookupByLabelRec prd r))
         , sp'' ~ (Inh3  (LookupByLabelRec prd r))
         ) =>
  ComSingle 'True prd (Rule sc ip ic  sp  ic'  sp') r where
  type ComSingleR 'True prd (Rule sc ip ic  sp  ic'  sp') r
    = UpdateAtLabelRecFR prd (Rule sc ip ic sp (Inh3 (LookupByLabelRec prd r))
                                               (Syn3 (LookupByLabelRec prd r))) r
  comSingle _ f r = updateAtLabelRecF l (oldR `ext` newR) r
    where l    = labelPrd f
          oldR = lookupByLabelRec l r
          newR = rulePrd f
```

Donde `Syn3` e `Inh3` son funciones de proyección definidas puramente a nivel de tipos.

```
type family Syn3 (rule :: Type) :: [(k', [(k, Type)])] where
  Syn3 (Rule sc ip ic  sp  ic'' sp'') = sp''
type family Inh3 (rule :: Type) :: [(k, Type)] where
  Inh3 (Rule sc ip ic  sp  ic'' sp'') = ic''
```

4.5 Funciones semánticas

En cada producción, llamamos *función semántica* al mapeo de los atributos heredados a los atributos sintetizados. Una computación sobre la gramática 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:

[11] Lo colocamos delante, pero podría ser en cualquier parte, como en todos los casos de uso de registros heterogeneos, el órden en el que aparecen los elementos no es relevante.

```
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.
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` [6] 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.
La función `empties` se define por recursión, sin mayores complicaciones:

```
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
      in  ConsCh (TaggedChAttr lch EmptyAtt) (empties fcr)
```

La función `kn` es una porción de código bastante técnica de programación a nivel de tipos. Dadas las funciones semánticas de los hijos y sus entradas (atributos heredados de los hijos) se construyen los atributos sintetizados para los hijos. La funcición debería tener un tipo similar a:

```
kn :: Record fcr -> ChAttsRec ich -> ChAttsRec sch
```

Pero observemos que `ich` y `sch` están determinados por el contenido de las funciones semánticas. La clase entonces se implementa con la firma:

```
class Kn (fcr :: [(k, Type)]) where
  type ICh fcr :: [(k, [(k, Type)])]
  type SCh fcr :: [(k, [(k, Type)])]
  kn :: Record fcr -> ChAttsRec (ICh fcr) -> ChAttsRec (SCh fcr)
```

Y luego la función `kn` se implementa por recursión en la forma de la función semántica. Se asemeja a `zipWith $`:

```
instance Kn '[] where
  type ICh '[] = '[]
  type SCh '[] = '[]
  kn _ _ = EmptyCh

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

5 Conclusión

En este documento mostramos las principales característica de la reimplementación de un subconjunto de la biblioteca AspectAG utilizando técnicas modernas de programación a nivel de tipos. Como resultado de la reimplementación 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
```

Y el tipo del constructor `Fam` está dado por:

```
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 (la constraint `LabelSet`).
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.[12]
La implementación de estructuras especializadas provee nombres mnemónicos para las estructuras, comparemos el tipo de `asp_sres` (definido en la sección 3.3) en la implementación antigua, 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)))
```

[12] 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.

```
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))]
```

Por otra parte, los mensajes de error han sido mejorados. Por ejemplo, supongamos que en el ejemplo `repmin` (sección 3.2) qomitimos en la definición de `asp_smin` la regla `p_Node .=. node_smin`. La gramática estará mal formada y la función `repmin`, no compilará.
En las versiones antigüas obtenemos el siguiente error críptico:

```
    . 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 [16] 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 [16] 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] Oege de Moor, Simon L. Peyton Jones, and Eric Van Wyk. Aspect-oriented compilers. In *Generative and Component-Based Software Engineering, First International Symposium, GCSE'99, Erfurt, Germany, September 28-30, 1999, Revised Papers*, pages 121–133, 1999.

[7] 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.

[8] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. *SIGPLAN Not.*, 47(12):117–130, September 2012.

[9] R. Eisenberg et al. singletons Library. http://hackage.haskell.org/package/singletons.

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

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

[18] Haskell Wiki. Heterogeneus collections. https://wiki.haskell.org/Heterogenous_collections. Accedido: 31-10-2018.

[19] 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.