Memorización selectiva de atributos en Attribute Grammars

Juan Saavedra

Tutores : Marcos Viera, Alberto Pardo

Facultad de Ingeniería

Carrera Ingeniería en computaciòn

28/09/2020

Contents

Chapter 1

Introducción

El mundo de la programación está en constante crecimiento y cada día aparecen nuevos lenguajes de programación que juegan un rol específico en la industria. Estos lenguajes son creados en su mayoría en base a lenguajes preexistentes, y se consideran de más alto nivel que su lenguaje anfitrión.

Los lenguajes de dominio específico (o DSL), son lenguajes creados y utilizados en contextos específicos como puede ser una tabla de Excel para un contador, MATLAB para matemáticos o Verilog para quienes programan chips electrónicos.

Cualquiera de estos actores podría utilizar un lenguaje de propósito general para resolver sus necesidades en vez de un DSL, como puede ser Java o C. MATLAB por ejemplo, implementa funciones matemáticas como *annurate*, que calcula una tasa de interés periódica, o *average*, que calcula el promedio de una lista de números. Si un matemático decidiera utilizar un lenguaje de propósito general como Java en vez de MATLAB, debería implementar y mantener todas estas funciones, lo cual implicaría un gran esfuerzo.

Es por esto que se utilizan los DSLs, nos permiten escribir instrucciones (o código) en un dominio específico para realizar determinadas acciones, como puede ser una fórmula de Excel o un comando SQL. Este código debe ser escrito en el formato sintáctico correcto que el DSL propone, así como también debe acatar ciertas reglas semánticas. Un compilador se encarga de realizar estas validaciones, y el ciclo de compilación se compone de tres fases:

1. Análisis léxico
2. Análisis sintáctico
3. Análisis semántico

Existen distintos formalismos que nos ayudan a resolver las distintas fases de un compilador, uno de ellos son las llamadas Attribute Grammars. Las Attribute Grammars (o AGs), definidas por Knuth [11] en 1968, son un formalismo que nos permite por ejemplo expresar algoritmos complejos de análisis y manipulación de lenguajes de programación. En el caso de un compilador, se utilizan las AGs para resolver la fase 3, el análisis semántico.

El análisis semántico parte de una gramática libre de contexto ya construida en las fases anteriores y realiza validaciones semánticas sobre la misma. Una gramática libre de contexto por sí sola no es capaz de representar aspectos semánticos como podría ser la restricción "toda variable debe haberse definido antes de ser usada" o "el nombre de una variable no puede repetirse". Para resolver este tipo de restricciones en un compilador se utilizan las AGs, que se pueden ver informalmente como una gramática libre de contexto decorada con atributos, condiciones y evaluación de reglas, que en conjunto proveen una extensión semántica a dicha gramática. Mostraremos un ejemplo en la Sección 4.2.

Las AGs se han utilizado no solo para especificar lenguajes de programación reales, por ejemplo Haskell [2], sino también para especificar algoritmos de pretty-printing [17], técnicas de deforestación [3], sistemas de tipos potentes [14], editores de sintaxis [9], entornos de programación [9] o lenguajes visuales [10].

Es de interés entonces investigar sobre técnicas que mejoren el desempeño de las AGs y también que faciliten su desarrollo. El objetivo de este proyecto es estudiar y demostrar la utilidad de una técnica llamada memorización selectiva de atributos. La técnica de memorización como forma de evitar el recálculo de funciones existe desde el siglo pasado y ha sido utilizada en áreas como programación dinámica y computación incremental. Existen varios papers en torno al tema, por ejemplo [1] donde se estudia la memorización de funciones en un lenguaje llamado *MLF*. Estas funciones reciben determinados inputs y sus resultados son almacenados en una memo table de tipo *Hash* utilizando como *key* estos inputs. En esta clase de memorización la semántica e implementación de la función *lookup* sobre la memo table resulta clave para la performance.

En [4] se muestra un mecanismo de memorización en el contexto de Attribute Grammars, donde se memoriza la ejecución de ciertas funciones llamadas atributos, los cuales se aplican sobre los nodos de la gramática. Tomando como base lo desarrollado en [4], el objetivo de este proyecto es ahondar en esta técnica específica. Utilizaremos el DSL que el paper propone para representar AGs así como el código necesario para memorizar sus atributos. Extenderemos esta última parte para lograr una memorización selectiva automática, y así comparar distintas estrategias de memorización en tiempo de ejecución.

El presente documento está estructurado de la siguiente forma:

- En el Capítulo 2 presentaremos un conjunto de definiciones necesarias para entender lo que se hizo en el proyecto. Luego estudiaremos el caso de *repmin*, mediante el cual presentaremos la técnica de memorización de atributos; *repmin* es un ejemplo de AG bien conocido en la bibliografía que también fue utilizado en [4]. A partir de este ejemplo expondremos la problemática del re-cálculo innecesario de atributos y una posible solución, que diseñaremos manualmente mediante el análisis del código de la AG. Por último hablaremos sobre las diferencias en tiempos de ejecución y de consumo de memoria que pueden causar las distintas estrategias para una misma AG.

- En el Capítulo 3 presentaremos un módulo escrito en Haskell llamado Test module, con el cual probaremos distintas AGs en busca de la mejor estrategia de memorización. Esta vez la evaluación será de forma empírica, ya que para algunos casos nos va a resultar difícil darnos cuenta de cuál es la mejor estrategia a partir del análisis del código. Además explicaremos qué parámetros recibe y los cambios necesarios en el código de la AG para poder utilizarlo.

- Ya finalizando, en el Capítulo 4 expondremos los resultados de testear 3 AGs distintas utilizando el módulo antes descrito. Estas pruebas nos ayudarán a sacar algunas conclusiones y observaciones sobre la técnica de memorización selectiva en AGs.

- Por último, en el Capítulo 5 expondremos algunos resultados observados así como posibles trabajos futuros en torno al tema.

Chapter 2

Antecedentes

Este proyecto se desprende directamente del estudio realizado en el paper [4] donde se presenta una forma de aplicar memorización en Attribute Grammars basadas en zipper genéricos. En el mismo se plantea la problemática del recálculo innecesario de atributos mediante una AG llamada *repmin*, la cual explicaremos en este capítulo y será también una de nuestras AG de estudio.

2.1 Attribute Grammars

Una AG está compuesta por atributos que realizan cálculos sobre los nodos de un árbol que satisface una cierta gramática. Estos atributos pueden ser de dos tipos [11]:

- Sintetizados: Un atributo sintetizado se calcula a partir de sus nodos hijos. Como los valores de los nodos hijos deben ser calculados primero, es un ejemplo de propagación de abajo hacia arriba.
- Heredados: Un atributo heredado se calcula a partir de su padre y/o hermanos. Es un ejemplo de propagación hacia abajo.

El resultado de un atributo entonces, depende de sus nodos hijos en el caso de los sintetizados, y de su padre/hermanos en el caso de los heredados. Un ejemplo de una gramática decorada con atributos es *repmin*. La misma recibe como parámetro de entrada un árbol binario de números enteros y retorna otro árbol con estructura idéntica pero con todas sus hojas sustituidas por el mínimo global. Esta AG consta de dos atributos sintetizados: *locmin*, *replace* y un atributo heredado: *globmin*.
En la Figura 2.1 se pueden observar las reglas de producción de la gramática así como el cálculo de

```
Root -> Tree
Root.globmin = Tree.locmin
Root.replace = Tree.replace

Tree -> "Fork" Tree1 Tree2
Tree1.globmin = Tree.globmin
Tree2.globmin = Tree.globmin
Tree.replace = "Fork" Tree1.replace Tree2.replace
Tree.locmin = min Tree1.locmin Tree2.locmin

Tree -> "Leaf" Int
Tree.replace = "Leaf" Tree.globmin
Tree.locmin = Int.value
```

Figure 2.1: Gramática repmin

los atributos en base a estas. Como se puede observar, los atributos sintetizados (*replace* y *locmin*) están definidos utilizando la parte derecha de las reglas de producción de la gramática, o sea, el valor del atributo del nodo padre se computa a partir del valor del atributo en los hijos. Mientras que *globmin* que es heredado, se define a partir de la parte izquierda, y el valor del atributo en los hijos se hereda del nodo padre.
Para poder codificar una AG de este estilo, necesitamos definir una forma de navegar el árbol de entrada de forma tal que podamos acceder a los distintos tipos de nodos hijos o padres partiendo de un nodo cualquiera. Para esto vamos a utilizar una estructura llamada *zipper*.

2.2 Zipper-based Attribute Grammars

2.2.1 Introducción

Consideremos la definición del siguiente árbol binario en Haskell:

```
data Tree = Leaf Int | Fork Tree Tree
```

Si quisiéramos por ejemplo calcular el mínimo de un árbol, podríamos utilizar una función del estilo:

```
locmin :: Tree -> Int
locmin (Fork t1 t2) = min (locmin t1) (locmin t2)
locmin (Leaf v) = v
```

Es decir, podemos navegar la estructura de arriba hacia abajo sin inconvenientes. Ahora bien, supongamos que desde un nodo hijo necesitamos calcular un valor que viene de su padre, por ejemplo desde un nodo *Leaf* consultar el mínimo global del árbol. Necesitamos alguna forma de "ir hacia arriba", hasta llegar al *Root* del árbol, y desde ahí calcular el mínimo global. Esto no es posible utilizando directamente la estructura de datos *Tree* ya que en Haskell no se manejan ni punteros ni estados como en otros lenguajes. Debemos entonces emplear algún otro mecanismo. Es aquí donde entran en juego los *zippers*.

Los *zippers* [13] encapsulan una estructura de datos de forma tal que nos permiten navegar utilizando operaciones predefinidas. Definen un contexto con el cual podemos reconstruir el nodo en el que estábamos luego de navegar en cualquier dirección, por ejemplo hacia un nodo hijo. Consideremos el siguiente código:

```
type Zipper = (Tree,Cxt)
data Cxt = Root | Left Cxt Tree | Right Tree Cxt
```

En *Cxt* almacenamos el contexto necesario para reconstruir el nodo anterior en el que estábamos. Ahora en vez de trabajar directo con el tipo *Tree*, utilizamos el tipo *Zipper*. Y definimos las operaciones que nos permiten movernos al subárbol izquierdo o derecho:

```
left :: Zipper -> Zipper
left (Fork l r,cxt) = (l,Left cxt r)
right :: Zipper -> Zipper
right (Fork l r,cxt) = (r,Right l cxt)
```

También definimos la función *up* que navega hacia el nodo padre:

```
up :: Zipper -> Zipper
up (t, Top) = (t,Root)
up (t, Left cxt r) = (Fork t r,cxt)
up (t, Right l cxt) = (Fork l t,cxt)
```

Observar que en todo momento tenemos un árbol en la componente izquierda de la tupla, es decir tenemos el foco en determinado nodo. Un ejemplo de navegación utilizando *zippers* se observa en la Figura 2.2, donde contamos con un árbol de 4 hojas. El contexto en la raiz del árbol contiene el valor *Root*. Luego de un movimiento, por ejemplo *left*, se guarda en el nuevo contexto el hijo

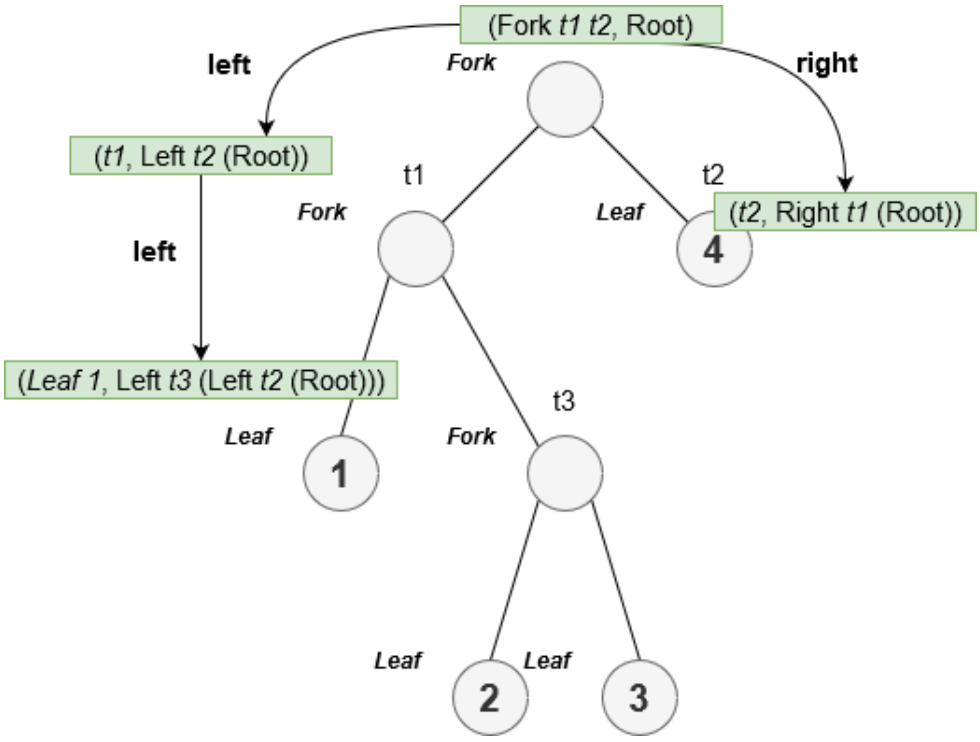


Figure 2.2: Navegación y contexto utilizando zippers

derecho *t2* y el contexto anterior, es decir *Root*, y se retorna en la primer componente de la tupla el nodo solicitado *t1*. Se puede observar que en el contexto se guarda toda la información necesaria para reconstruir el nodo padre.
Definiendo algunas operaciones más, por ejemplo para consultar y actualizar el valor del árbol dentro del *zipper*, podemos decir que contamos con lo necesario para navegar la estructura y modificarla. Si bien en [4] se utiliza un *zipper* particular como el recién descrito, en este proyecto vamos a cambiar esa parte y utilizaremos directamente la biblioteca *Generics.Zipper* [13] (o GZ), la cual implementa *zippers* de forma genérica y nos permite asociarlos a estructuras de datos arbitrarias del usuario.

2.2.2 Data.Generics.Zipper

Partimos de nuestra estructura inicial *Tree* y la embebemos en una estructura de tipo *Zipper Tree* utilizando la operación:

```
toZipper :: Data a => a -> Zipper a -- Definida en Generics.Zipper
```

Una vez creado el *zipper*, podemos utilizar las siguientes funciones definidas en la biblioteca:

```
GZ.left    :: Zipper a -> Maybe (Zipper a) -- Mueve el foco hacia el hermano izquierdo
GZ.right   :: Zipper a -> Maybe (Zipper a) -- Mueve el foco hacia el hermano derecho
GZ.down'   :: Zipper a -> Maybe (Zipper a) -- Mueve el foco hacia el hijo de m s a la
GZ.up      :: Zipper a -> Maybe (Zipper a) -- Mueve el foco hacia arriba
GZ.getHole :: Typeable b => Zipper a -> Maybe b --  Retorna la estructura original dent
                                                --  Si el tipo no coincide, retorna Not
GZ.setHole :: Typeable a => a -> Zipper b -> Zipper b -- Actualiza el valor dentro del
```

Cuando el valor retornado por *getHole* es *Nothing*, significa que el tipo de datos que esperábamos no coincide con el del foco. En este caso nuestro AG tiene un error en el flujo y debemos corregirlo.
Vamos a definir ahora algunas funciones en base a las de la biblioteca:

```
left :: Zipper a -> Zipper a -- Hijo izquierdo
left t = t.$1
right :: Zipper a -> Zipper a -- Hijo derecho
right t = t.$2
up :: Zipper a -> Zipper a -- Padre
up t = parent t
parent = fromJust $ GZ.up
```

```
(.$) :: Zipper a -> Int -> Zipper a -- Retorna el hijo n mero n
z .$ 1 = fromJust (GZ.down' z)
z .$ n = fromJust (GZ.right ( z.$(n-1) ))
(.|) :: Zipper a -> Int -> Bool -- Chequea si z es el hermano n
z .| 1 =  case (GZ.left z) of
            Nothing -> True
            _ -> False
z .| n =  let l = (GZ.left z)
          in  case l of
                Nothing -> False
                Just x ->  fromJust (l) .| (n-1)
```

Estas funciones nos van a ayudar a escribir nuestras AGs de manera elegante. Veremos a continuación cómo representar una AG.

2.2.3 Representación de AGs

Definimos una AG de la siguiente manera:

```
type AGTree a = Zipper a -> a
```

Y la construimos partiendo de un *Tree*:

```
mkAG :: Tree -> Zipper Tree
mkAG t = toZipper t
```

Luego definimos el tipo de datos *Cons* y la operación *constructor*, que nos va a permitir determinar en qué tipo de nodo está el foco en un momento dado.

```
data Cons = CRoot | CFork | CLeaf Int
constructor :: Zipper Tree -> Cons
constructor a =  case ( getHole a :: Maybe Tree ) of
                   Just (Fork _ _ _) ->
                     case (up a) of
                       Nothing -> CRoot
                       otherwise -> CFork
                   Just (Leaf _ l) -> CLeaf l
```

Observar que la función *up* nos devuelve el valor *Nothing* cuando estamos en el nodo raiz del arbol. Vamos a ver a continuación el caso de *repmin*, donde utilizaremos todos los conceptos vistos hasta el momento para escribir su código.

2.3 Repmin

2.3.1 Introducción

Como se explicó anteriormente, *repmin* recibe un árbol de enteros y sustituye todas sus hojas con el mínimo global. El código principal de la AG se muestra en la Figura 2.3.1 y se observa que el algoritmo consta de tres atributos:

- *replace :: AGTree Tree*. Atributo sintetizado que reemplaza cada hoja por el valor de *globmin*.
- *globmin :: AGTree Int*. Atributo heredado que retorna el mínimo global. Es decir que por cada invocación, recorre el árbol en busca de su mínimo.
- *locmin :: AGTree Int*. Atributo sintetizado que busca el mínimo local. Es utilizado por *globmin* para calcular el mínimo global.

La dinámica del algoritmo es ir recorriendo el árbol de izquierda a derecha y en cada hoja reemplazar su valor actual por el mínimo global. Para acceder al mínimo global desde las hojas, se llama al atributo heredado *globmin*. Este atributo sube hasta la raíz del árbol y luego calcula el mínimo global a partir del sintetizado *locmin*.
Cada llamada a *globmin* implica que se recorra el árbol por completo en busca del mínimo global. Si observamos detenidamente, este comportamiento se repite por cada invocación a *globmin* en cada hoja y no es para nada eficiente.
La idea es evitar este recálculo de atributos almacenando el resultado de la primera ejecución en algun lado. En [4] se plantea como solución almacenar estos valores en lo que llamaremos *Memo-Table* o tabla de memorización, la cual presentaremos en la siguiente sección.
El flujo de ejecución de *repmin* puede visualizarse más en detalle en la Figura 2.4. La misma muestra las llamadas a *globmin* y *locmin* implicadas en el cálculo de *replace* en un árbol de cuatro hojas. En la primera imagen se muestra el flujo de ejecución en la hoja de valor 1. En la siguiente imagen se muestra el flujo del mismo atributo pero en la segunda hoja. Se pueden observar subrayados los re-cálculos tanto de *globmin* como de *locmin*.

```
-- Inherited
globmin :: AGTree Int
globmin t =  case constructor t of
               CRoot   -> locmin t
               CLeaf l -> globmin (up t)
               CFork   -> globmin (up t)

-- Synthesized
locmin :: AGTree Int
locmin t =  case constructor t of
              CLeaf l -> l
              CFork   -> min (locmin (left t)) (locmin (right t))

-- Synthesized
replace :: AGTree Tree
replace t =  case constructor t of
               CLeaf l -> Leaf (globmin t)
               _       -> Fork (replace (left t)) (replace (right t))

repmin :: Tree -> Tree
repmin t = replace (mkAG t)
```

Figure 2.3: repmin

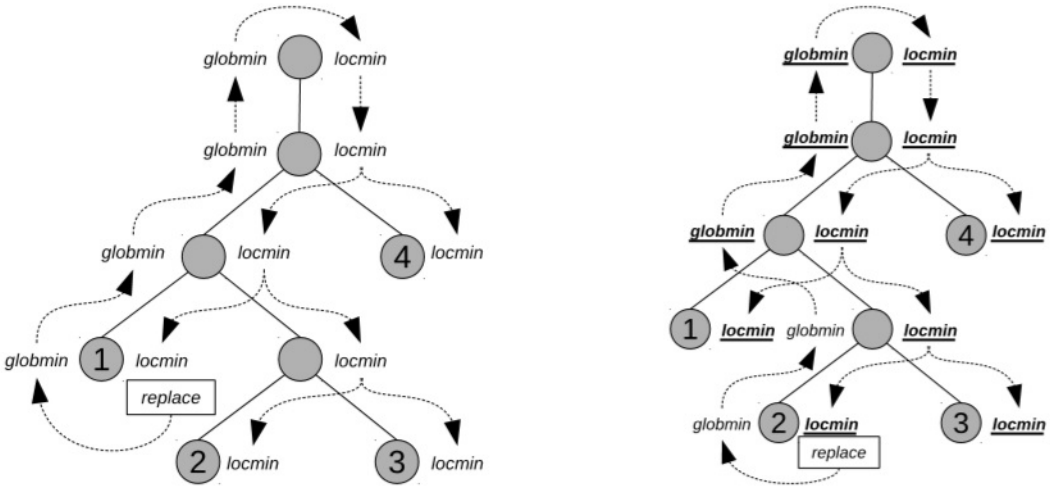


Figure 2.4: Flujo repmin

2.3.2 Memorización

Vamos a presentar ahora una solución al problema de re-cálculo de atributos, que fue inicialmente presentada en [4].

Primero definamos el tipo de datos de *MemoTable*, el cual depende de los tipos de datos de los atributos de la AG (ya que son los valores que queremos memorizar).

```
type MemoTable = (Maybe Int ,Maybe Int ,Maybe Tree)
```

Donde la primera componente corresponde al atributo *globmin*, la segunda a *locmin* y la tercera a *replace*. Se utiliza *Maybe* [5] debido a que un atributo puede haber sido calculado o no en determinado momento.

Tenemos ahora que agregar la tabla en nuestra estructura de datos para poder trabajar con ella. Además va a ser necesario modificar el código de la AG.

El tipo *Tree* modificado queda:

```
data Treem m = Leafm m Int | Forkm m (Treem m) (Treem m)
```

Observar que *Treem* es un tipo algebraico polimórfico que al definirlo de esta forma nos permite implementar la estructura de la *Memotable* de diferentes maneras.

Nos interesa ahora definir funciones para guardar un valor o consultar la tabla para obtener uno previamente calculado. Para esto definimos la clase *memo*:

```
data Globmin = Globmin
data Locmin = Locmin
data Replace = Replace
class memo att m a where
  mlookup :: att -> m -> Maybe a
  massign :: att -> a -> m -> m
```

La clase *memo* es instanciada utilizando un atributo *att* (*Globmin, Locmin* o *Replace*), el tipo *MemoTable* y el tipo que retorna el atributo, como se muestra a continuación:

```
instance Memo Globmin MemoTable Int where
  mlookup _ (g,_,_) = g
  massign _ v (g,l,r) = (Just v,l,r)
instance Memo Locmin MemoTable Int where
  mlookup _ (_,l,_) = l
  massign _ v (g,j,r) = (g,Just v,r)
instance Memo Replace MemoTable Tree where
  mlookup _ (_,_,r) = r
  massign _ v (g,l,r) = (g,l,Just v)
```

Se observa que para cada atributo se actualiza o retorna el valor correspondiente en la tupla de la *MemoTable*. Esta implementación puede ser modificada según el tipo de dato que se utilice como implementación de la *MemoTable*. De hecho fueron probadas otras implementaciones como por ejemplo listas heterogéneas HList [7] y Data.Map [6], pero no se llegó a ninguna mejora en el código que justificara cambiar la implementación de tuplas.
Volviendo al código, vamos a definir algunas funciones más para terminar de incorporar lo que necesitamos. Para consultar y actualizar la tabla dentro del *Zipper* definimos:

```
-- Consulta la memo table de un tree
getMemoTable :: Zipper ( Treem m ) -> m
getMemoTable t = fromJust( getHole (t.$1) )
-- Modifica la memo table de un Tree
updMemoTable :: (m -> m) -> Treem m  -> Treem m
updMemoTable f (Leafm mt i) = Leafm (f mt) i
updMemoTable f (Forkm mt l r) = Forkm (f mt) l r
-- Modifica un tree dentro de un zipper
modifym :: Zipper ( Treem m ) -> (Treem m -> Treem m) -> Zipper ( Treem m )
```

```
modifym a f =  case ( getHole a :: Maybe ( Treem m ) ) of
                  Just t -> setHole (f t) a
```

También tenemos que re-definir nuestras funciones *right* y *left*, ya que ahora agregamos la tabla de memorización y el tipo *Fork* pasa a tener 3 hijos.

```
leftm :: Zipper ( Treem m ) -> Zipper ( Treem m )
leftm t = t.$2
rightm :: Zipper ( Treem m ) -> Zipper ( Treem m )
rightm t = t.$3
```

Para construir un *Treem* a partir de un *Tree* definimos la función *buildm*. Esta función generalmente se utiliza para construir la estructura de datos inicial con la tabla vacía.

```
buildm :: Tree -> m -> Treem m
buildm (Leaf n) mt = Leafm mt n
buildm (Fork l r) mt = Forkm mt (buildm l mt) (buildm r mt)
```

Luego, para aplicar memorización, vamos reescribir el tipo *AGtree* y también vamos a definir dos funciones que nos van a ser de utilidad para navegar correctamente la estructura de datos. El tipo *AGtree* pasa a ser:

```
type AGTreem m a = Zipper ( Treem m ) -> (a, Zipper( Treem m ))
```

donde el valor retornado *(a, Zipper(Treem m))* contiene en la primer componente el valor *a* que es el resultado del atributo que invocamos y en la segunda componente el árbol modificado *Zipper(Treem m)*. Este último es imprescindible ya que en él van a estar memorizados los atributos que se hayan calculado en esa llamada así como en sus invocaciones recursivas. Por lo tanto cada vez que invocamos un atributo en un nodo y recibimos una tupla como la recién descrita, debemos continuar nuestra ejecución con el árbol retornado ya que en él se encuentran los resultados memorizados.
Cada invocación a un atributo en un nodo hijo o padre provoca que se pierda el foco. Por ejemplo si aplicamos *leftm* o *upm* antes de invocar el atributo estamos moviendo el foco. Para devolver el foco al nodo en el que estábamos definimos las siguientes funciones auxiliares:

```
(.@.) :: AGTreem m a -> AGTreem m a -- Eval a en hijo y retorna foco al padre
eval .@. z =  let (v,z0) = eval z
              in  (v,upm z0)
```

```
atParent eval z = (v,(back z) z0) -- Eval a en padre y retorna el foco al hijo
                  where
                    (v,z0) = eval (upm z)

back t = if (t.|2) then
           leftm
         else
           rightm
```

Observar que *(.@@.)* se define invocaciones en nodos hijos mientras que *atParent* se define para el nodo padre.
Por último vamos a presentar la función *memo*, que es donde se realiza efectivamente la tarea de memorización. La misma chequea si el valor de un atributo en un nodo no fue previamente calculado. En caso de haber sido calculado, lo retorna, en caso contrario lo calcula y luego actualiza la tabla.

```
memo :: (Memo attr m a, Show attr, Show a) => attr -> AGTreem m a -> AGTreem m a
memo attr eval z = case mlookup attr (getMemoTable z) of
                     Just v -> (v,z)
                     Nothing -> let (v,z0) = eval z
                                in (v, modifym z0 (updMemoTable (massign attr v)))
```

2.3.3 Repmin memorizado

En la Figura 2.3.3 se muestra el código de *repmin* modificado para aplicar memorización. Podemos ver la utilización de las funciones anteriormente definidas como *.@@.*, *atParent* y *memo*. Se puede observar también que el código principal no sufrió grandes modificaciones en su sintaxis y conserva una codificación elegante.

También estamos seguros de que esta nueva Attribute Grammar tiene exactamente el mismo comportamiento que la original pero incluye los siguientes cambios:

- Cada nodo del árbol contiene una *MemoTable* inicialmente vacía que servirá para almacenar los resultados de las invocaciones de los atributos.
- Al invocar un atributo cualquiera sobre un nodo, primero se chequea que no exista un resultado previamente almacenado en su *MemoTable*. Si existe un resultado, es retornado de

```
    -- Inherited
    globminm :: AGTreem m Int
    globminm = memo Globmin $ \z -> case constructorm z of
                                      CRootm   -> locminm z
                                      CLeafm _ -> globminm `atParent` z
                                      CForkm   -> globminm `atParent` z

    -- Synthesized
    locminm :: AGTreem m Int
    locminm = memo Locmin $ \z -> case constructorm z of
                                    CLeafm v -> (v,z)
                                    CForkm -> let (left, z0) = locminm .@. leftm z
                                                  (right, z00) = locminm .@. rightm z
                                              in (min left right,z00)
    -- Synthesized
    replacem :: AGTreem m Tree
    replacem = memo Replace $ \z -> case constructorm z of
                                    CLeafm _ ->
let (mini,z0) = globminm z
                                                in (Leaf mini,z0)
                                    _ -> let (l, z0) = replacem .@. leftm z
                                             (r,z00) = replacem .@. rightm z0
                                         in (Fork l r,z00)
```

Figure 2.5: Código repmin con memorización

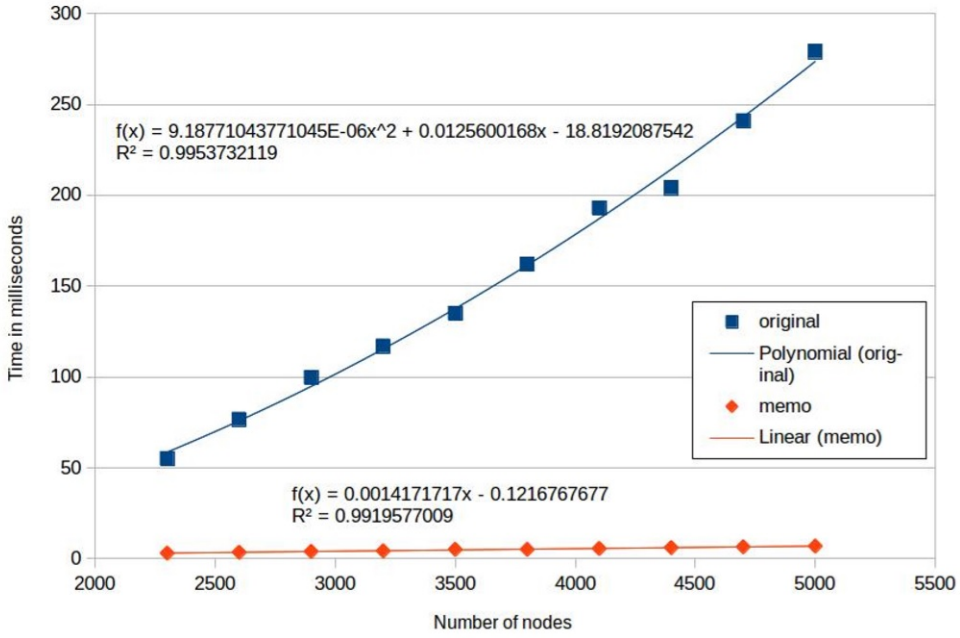


Figure 2.6: Resultado repmin full memoization

inmediato, de lo contrario primero se calcula el atributo, luego se almacena el resultado en la tabla y por último se retorna el resultado.

2.3.4 Resultados en términos de tiempo de ejecución

El resultado en términos de tiempo de ejecución luego de comparar la versión sin memorización con la versión que aplica memorización completa se muestra en la Figura 2.6. Este resultado fue el obtenido en [4]. Se observa una notoria mejora al aplicar memorización, lo cual era de esperar. En esta primera prueba se utiliza memorazión completa, es decir de todos sus atributos. Pero intuitivamente y por lo que se detallo antes, podemos darnos cuenta que la mejora se da al memorizar únicamente el atributo *globmin*. Si éste atributo es calculado una única vez, entonces *locmin* también es invocado una única vez por nodo y no es necesario memorizarlo. Por otro lado, *replace* es invocado una vez por nodo siempre.

Esto nos da la pauta de que podríamos memorizar algunos atributos y no todos para optimizar aún más la ejecución y evitar así el overhead de almacenar y chequear atributos que no nos interesan. La Figura 2.7 expone el resultado luego de aplicar memorización selectiva, es decir, memorizando únicamente el atributo *globmin* y dejando fuera los otros dos.

Se observa una ganancia respecto a la memorización completa, lo cual está alineado con el análisis

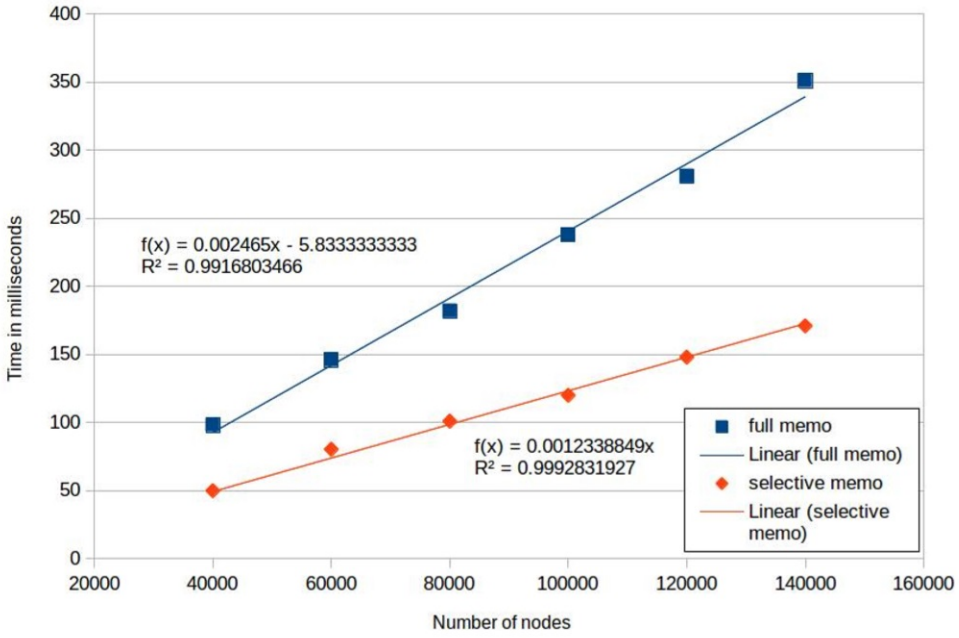


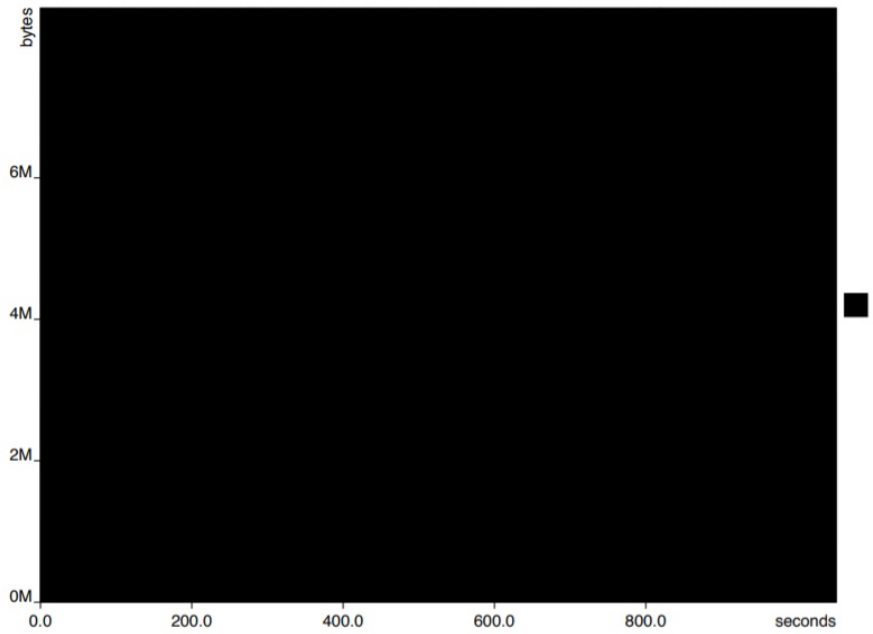
Figure 2.7: Resultado selective memoization

anterior. A continuación estudiaremos el consumo de memoria provocado por los tres casos vistos.

2.4 Consumo de memoria

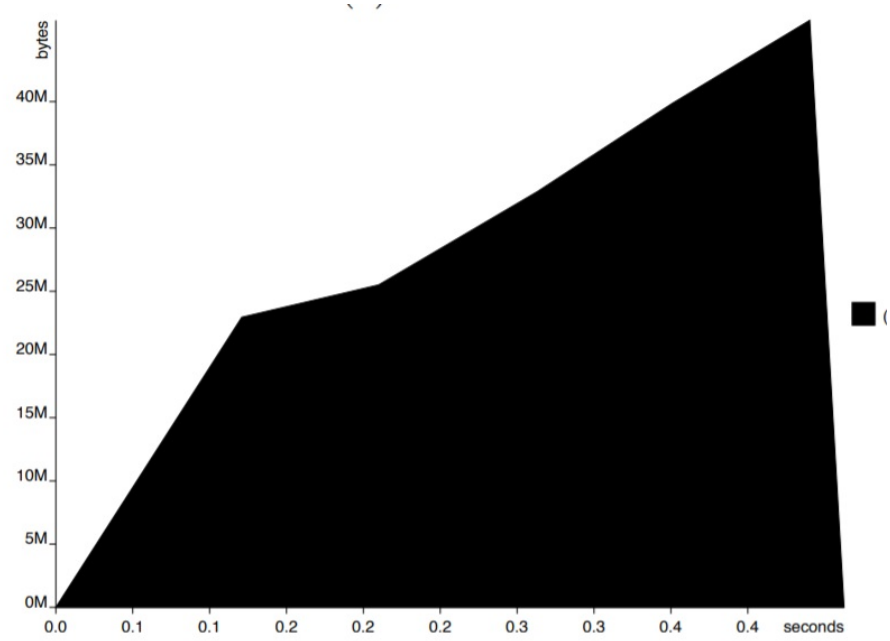
En [4] se plantea que para obtener mejores resultados en términos de tiempo de ejecución utilizando la técnica de memorización es necesario sacrificar en términos de consumo de memoria, asegurando que la versión de *repmin* que memoriza sus atributos, consume más memoria que la original. En las Figuras 2.8, 2.9 y 2.10 se exponen los resultados presentados en [4] luego de aplicar heap profiling para los tres casos vistos. Como se menciona en [4], se puede observar que la versión sin memorización es la que consume menos memoria, alcanzando un máximo aproximado de 8 MB mientras que la versión Full Memoization ronda entorno a los 50 MB. Por último, la versión de Selective Memoization presenta un descenso respecto a la anterior, alcanzando un máximo aproximado de 18 MB. Veremos más adelante que este resultado en realidad refiere a los picos de memoria alcanzados por cada versión durante su ejecución, y no al consumo total de memoria. Por el contrario, veremos que al memorizar atributos el consumo de memoria total disminuye al igual que lo hace el tiempo de ejecución.

Más allá de esta observación, de [4] surgen dos preguntas clave que nos conducen al resto de la



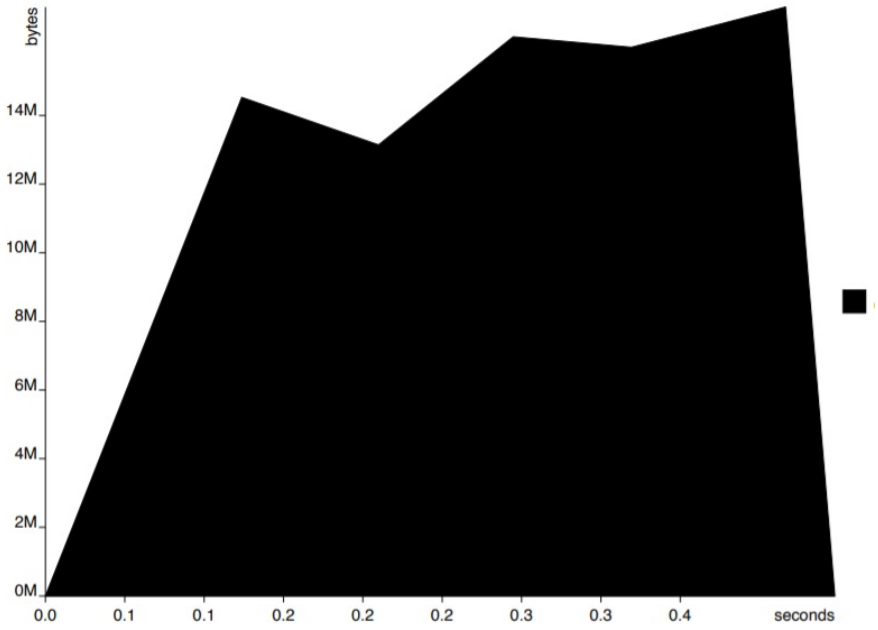
(a) Non memoized.

Figure 2.8: Profiling repmin



(b) Full Memoization.

Figure 2.9: Profiling repmin full memoization



(c) Selective Memoization.

Figure 2.10: Profiling repmin memorizacion selectiva

investigación y motivan el estudio realizado en este proyecto:

- ¿Qué sucede si además de ser selectivos con los atributos a memorizar, también lo somos con los nodos en los que vamos a memorizar? En este caso en particular, ¿es necesario almacenar el resultado de *globmin* tanto en los nodos de tipo *Fork* como en los de tipo *Leaf*?

- Esta AG resulta fácil de entender a simple vista y nos ilustra de forma clara el problema de memorización, lo cual nos permite razonar y modificar manualmente su código para lograr una memorización efectiva. ¿Qué sucede si consideramos gramáticas más complejas, por ejemplo [16], que no resulten tan triviales de analizar? ¿Qué criterios utilizaremos en estos casos para lograr una memorización efectiva?

Para responder a las preguntas anteriores, analizaremos y testearemos tres AGs con características bien distintas. Cada una de ellas nos ayudará a sacar distintas conclusiones. Para este análisis construiremos un módulo llamado *TestModule*, el cual nos ayudará a automatizar las pruebas en busca de la mejor estrategia.
La primera AG que probaremos es *repmin*. La segunda es un compilador de un lenguaje de programación imperativo ficticio llamado *KLanguage* que construimos para este trabajo. Por

último analizaremos una AG más compleja también presentada en [4] llamada *HTMLFormatter*.

Chapter 3

Test module

Este módulo fue construido para responder a las dos preguntas anteriormente planteadas. Queremos encontrar la mejor estrategia de memorización para una AG cualquiera a partir de resultados empíricos, testeando distintas combinaciones de nodos y atributos.
Una AG puede resultar compleja de entender, tanto sintáctica como semánticamente. Esta dificultad se acrecienta cuando además de comprenderla debemos encontrar una forma efectiva de memorizar sus atributos. Atributos que además, tienen interdependencias entre sí, lo cual dificulta aún más la tarea.
La idea es, de forma similar a como lo hace *GridSearch* de *Python* en el marco del aprendizaje automático [15] , realizar un testeo exhaustivo entre todas las estrategias posibles que resulten de combinar atributos y nodos y seleccionar la mejor.

3.1 Cálculo de cantidad de estrategias

Siguiendo con el caso de *repmin* como ejemplo, consideremos la lista de sus atributos y la de sus nodos. Para cada lista, calculemos las distintas combinaciones que podemos lograr a partir de sus elementos.

- Atributos : [*globmin*, *locmin*, *replace*]. Combinaciones: [(*globmin*, *locmin*, *replace*), (*globmin*, *locmin*), (*globmin*, *replace*), (*locmin*, *replace*), (*globmin*), (*locmin*), (*replace*)]
- Tipos de nodos : [*Fork*, *Leaf*]. Combinaciones: [(*Fork*,*Leaf*),(*Fork*),(*Leaf*)]

Podemos observar que para cada lista con n elementos, la cantidad de combinaciones distintas para i entre 1 y n se expresa con la siguiente fórmula:

$$\sum_{i=1}^{n} \binom{n}{i} = \sum_{i=1}^{n} \frac{n!}{i! * (n-i)!} \tag{3.1}$$

Por lo tanto, en el caso de *repmin* tenemos 7 combinaciones para el caso de los atributos y 3 para los nodos. Si cruzamos estas dos listas, obtenemos un total de 7 * 3=21 estrategias distintas que podemos crear. A este número hay que sumarle aquella estrategia que no implementa memorización, que también es una estrategia válida y podría resultar ser la mejor, por lo tanto en total terminan siendo 22 estrategias. La primera estrategia por ejemplo podría ser ([*globmin*,*locmin*,*replace*] , [*Fork*,*Leaf*]), aquella que memoriza todos los atributos en todos los nodos. La segunda ([*globmin*,*locmin*,*replace*], [*Fork*]) memoriza todos los atributos sólo en los nodos de tipo *Fork*.

Lo ideal sería probar las 22 estrategias en forma simultánea utilizando el mismo set de datos en cada una, y tomar las primeras que terminen como las mejores. Esta fue la intención inicial de este módulo y seria lo teóricamente correcto de hacer, de hecho, el módulo permite realizar esta prueba. El problema es que el número de estrategias a probar puede resultar demasiado grande para ciertas AGs. Sin ir más lejos, considerando la tercera AG de la Sección 4, ésta cuenta con 4 atributos y 27 nodos. Utilizando la fórmula antes presentada, resulta en un total de 65535 * 511 = 33.488.385 estrategias distintas a probar, lo cual es computacionalmente muy difícil de abordar.

El módulo por lo tanto permite crear conjuntos de estrategias un poco más específicos. Podemos indicar la lista de nodos sobre los cuales generar las combinaciones, de esta forma es posible dejar de lado aquellos nodos que de antemano sabemos no nos van a aportar nada, como el caso de *Leaf* en *repmin*. Lo mismo podemos hacer con los atributos.

Según la prueba que el usuario quiera hacer, se generan N instancias distintas y se lanzan a correr en paralelo sobre un set de datos de entrada también definido por el usuario mediante la biblioteca QuickCheck [8]. Las primeras que finalicen la ejecución serán seleccionadas como las mejores. Esta medición es en términos de tiempo de ejecución, pero no necesariamente en términos de consumo de memoria. De todas formas el módulo presenta los resultados de ambas mediciones.

3.2 Parámetros de entrada y código

La función *evalMemo* es la única función pública que tiene el módulo y es la que invocamos para realizar las pruebas. La firma de la función es la siguiente:

```
evalMemo :: Bool -> Int -> [[c]] -> [m] -> [Int] -> ( Int -> Gen t )
                -> ( m -> [c] -> t -> a ) -> (a -> b) -> IO ()
```

Y un ejemplo de una llamada es:

```
evalMemo finalLc n lc lm xs g h f
```

Donde la semántica de los datos de entrada es la siguiente:

- finalLc : Si es False, se generan a partir de *lc* todas las combinaciones posibles con sus elementos. Si es True, se utilizan los elementos de *lc* como la lista final de combinaciones a probar.
- w : Finaliza la ejecución cuando terminen las mejores n instancias
- lc : Lista de listas de nodos sobre los cuales memorizar
- lm : Lista de memo tables con las que probar. Esta lista define las distintas combinaciones de atributos con las que vamos a testear.
- xs : Lista de valores sobre los cuales iterar (eje x de la gráfica resultados).
- g : Función que genera a partir de los valores de *xs*, las estructuras de datos de entrada a la AG que queremos probar. Esta función debe ser construida por el usuario utilizando el módulo Test.QuickCheck [8]. Para *repmin* por ejemplo, los valores de *xs* podrían representar la profundidad del árbol a construir, o la cantidad de nodos.
- h : Construye a partir de un m, *[c]* y un t una instancia con determinada estrategia a probar.
- f : Es la AG que queremos probar, recibe como parámetro el a construido por h

La parte principal del código se observa en la Figura 3.1. Primero se generan los casos de prueba, luego se reducen hasta su forma normal antes de lanzar a correr los hilos para evitar que Lazy Evaluation interfiera en las mediciones. Luego se definen dos variables de tipo *MVar*, a través de las cuales los hilos se comuinican. Una sirve para reportar los resultados de un hilo una vez finalizada su ejecución y la otra se utiliza como semáforo para lanzar a correr las N instancias a la misma vez. Por último, *iterStrategies* crea las N estrategias a probar, asignando un hilo para cada una. Los hilos se bloquean en la variable *startS* hasta recibir la señal de arranque. Si bien no se muestra el código, *iterStrategies* utiliza la operación *forkIO* de *Control.Concurrent* para crear un nuevo hilo. Una vez creados los N hilos, el hilo principal notifica al resto mediante *startS* para comenzar la ejecución.
En el Anexo A se encuentra el código completo del módulo *TestModule* y algunos ejemplos de cómo utilizarla para probar *repmin* se pueden encontrar en el Anexo B. También algunas funciones de generación de casos de prueba con *QuickCheck*.

Figure 3.1: evalMemo

```
evalMemo finalLc w (lc:lcs) lm xs g h f =
do
    -- Se define la lista de nodos de cada estrategia
    lc' <- if finalLc then
             return (lc:lcs)
           else
             return (combs lc)

    -- Se generan los casos de prueba megiante g
    ts  <- genCases xs g

    -- Se generan N listas de ts, una por cada estrategia.
    tsm <- genCasesMemo lc' lm ts h

    -- Evaluamos tsm a su forma normal para que los pasos anteriores se ejecuten
    -- antes de lanzar las estrategias a correr. De lo contrario por Lazy Evaluation
    -- tsm no se va a reducir hasta que se precise y esto puede afectar las mediciones
    evaluate (rnf tsm)

    -- Definimos una MVar para notificar la finalizacion de un hilo
    m <- newEmptyMVar
    let log = Logger m

    -- Definimos otra MVar para comenzar todas las instancias a la vez. Los hilos se bl
    -- mediante startS hasta recibir la se al que les permite comenzar. Esta MVar act
    startS <- newEmptyMVar

    -- Lanzamos a correr los N hilos.
    iterStrategies startS log xs tsm f

    -- Se notifica a los hilos el comienzo de la ejecuci n mediante startS.
    putMVar startS "start"

    -- Finaliza la ejecuci n luego de terminadas las w primeras instancias.
    s <- firstN log w []
```

3.3 Cambios en el código de la AG

Vamos a ver ahora algunas modificaciones que deben realizarse en el código de la AG para que sea compatible con el módulo de testeo. Una de ellas es redefinir la *MemoTable* para que pase de tener una estructura de tipo

```
(Maybe Int, Maybe Int, Maybe Tree)
```

a una de tipo

```
((Bool, Maybe Int), (Bool, Maybe Int), (Bool, Maybe Tree))
```

De esta forma el módulo es capaz de crear las distintas estrategias de memorización prendiendo o apagando cada atributo en cada nodo utilizando la nueva componente de tipo *Bool*. Esta configuración se realiza en el módulo antes de correr las pruebas, en el paso *iterStrategies*. Se le pasa como parámetro a *buildm* la estrategia seleccionada (nodos y atributos a memorizar) y se construye el árbol con los atributos correspondientes prendidos en cada nodo. Luego, en tiempo de ejecución, cuando la AG debe calcular un atributo en cierto nodo, si la casilla en la *MemoTable* para este atributo está seteada en *True*, entonces se aplica memorización, es decir se chequea la tabla. De lo contrario se recalcula el atributo sin utilizar la *MemoTable*.
El código de *buildm* por lo tanto queda de la siguiente manera:

```
buildm :: MemoTable -> [Consm]-> Tree -> Treem MemoTable
buildm mt selL (Leaf n) = Leafm (selMT (CLeafm 0) selL mt) n
buildm mt selL (Fork l r) = Forkm (selMT CForkm selL mt) (buildm mt selL l) (buildm mt

selMT :: Consm -> [Consm] -> MemoTable -> MemoTable
selMT c [] m = emptyM
selMT c (x:xs) m =  if x == c then
                      m
                    else
                      selMT c xs m

emptyM = ((False,Nothing),(False,Nothing),(False,Nothing))
```

La *MemoTable mt* que recibe como parámetro *buildm* ya tiene seteados los atributos a memorizar, por ejemplo en *repmin* si memorizamos únicamente *globmin*, esa mt va a tener la siguiente forma:

```
((True,Nothing),(False,Nothing),(False,Nothing))
```

Luego, *selMT* chequea si el nodo que se está construyendo pertenece a la lista de nodos a memorizar *lc* o no. En caso de pertenecer, utiliza el parámetro *mt* como su *MemoTable*, en caso contrario utiliza *emptyM*, que "apaga" todos los atributos.
Recordemos que en la Sección 2.2 para aplicar memorización selectiva lo que hicimos fue modificar el código y re-compilar. En este caso esa tarea se realiza en tiempo de ejecución, lo cual nos permite probar muchas instancias de forma automática.
Por último debemos modificar la función *memo*. Hay que chequear si luego de calcular un atributo en cierto modo, es necesario aplicar memorización y guardar el resultado en la *MemoTable* o por el contrario, ignorar esta acción.

```
memo :: attr -> AGTreem MemoTable a -> AGTreem MemoTable a
memo attr eval z =
  case mlookup attr (getMemoTable z) of
    (_,Just v) -> (v,z)
    (x,Nothing)->  let (v,z0) = eval z
                   in  if x then
                         (v, modifym z0 (updMemoTable (massign attr v)))
                       else
                         (v,z0)
```

Realizando los cambios mencionados podemos entonces comenzar a probar distintas estrategias utilizando *TestModule*. En el Anexo B se encuentran algunos ejemplos de como probar el caso de *repmin*.

Chapter 4

Resultados

En este capítulo presentaremos las tres gramáticas construidas a lo largo del proyecto así como sus resultados luego de testearlas con el módulo de testeo.

4.1 Repmin

Como se mencionó anteriormente, para el caso de *repmin* tenemos 22 estrategias distintas de memorización. Se realizaron cuatro pruebas, en cada una de ellas el eje de las x representa la cantidad de nodos del árbol de entrada al algoritmo y su crecimiento es siempre lineal. El eje de las y para la gráfica superior representa el tiempo de ejecución en segundos, mientras que para la gráfica inferior representa el consumo de memoria en bytes.
La primera prueba memoriza todos los atributos y varía las distintas combinaciones de nodos. El resultado obtenido se muestra en la Figura 4.1. Se observa que la mejor estrategia (tanto en tiempo de ejecución como en consumo de memoria) es la que memoriza únicamente en los nodos *Fork*, como era de esperar. La sigue aquella que memoriza en los dos nodos [*Fork*, *Leaf*]. Estas dos estrategias presentan un crecimiento mínimo y prácticamente permanece en 0 segundos su tiempo de ejecución, mientras que las otras dos (aquellas que no memorizan en *Fork*) presentan un crecimiento exponencial en ambas mediciones.
En la segunda prueba dejamos fijos los nodos y variamos los atributos a memorizar, el resultado se expone en la Figura 4.2. Como se observa, la mejor estrategia resulta ser aquella que memoriza únicamente el atributo *globmin*. Las tres mejores instancias son muy similares, ya que todas ellas memorizan el atributo *globmin* que es el atributo que importa. La diferencia se encuentra en el overhead que provoca memorizar por ejemplo el atributo *replace* en la instancia 2. Este atributo

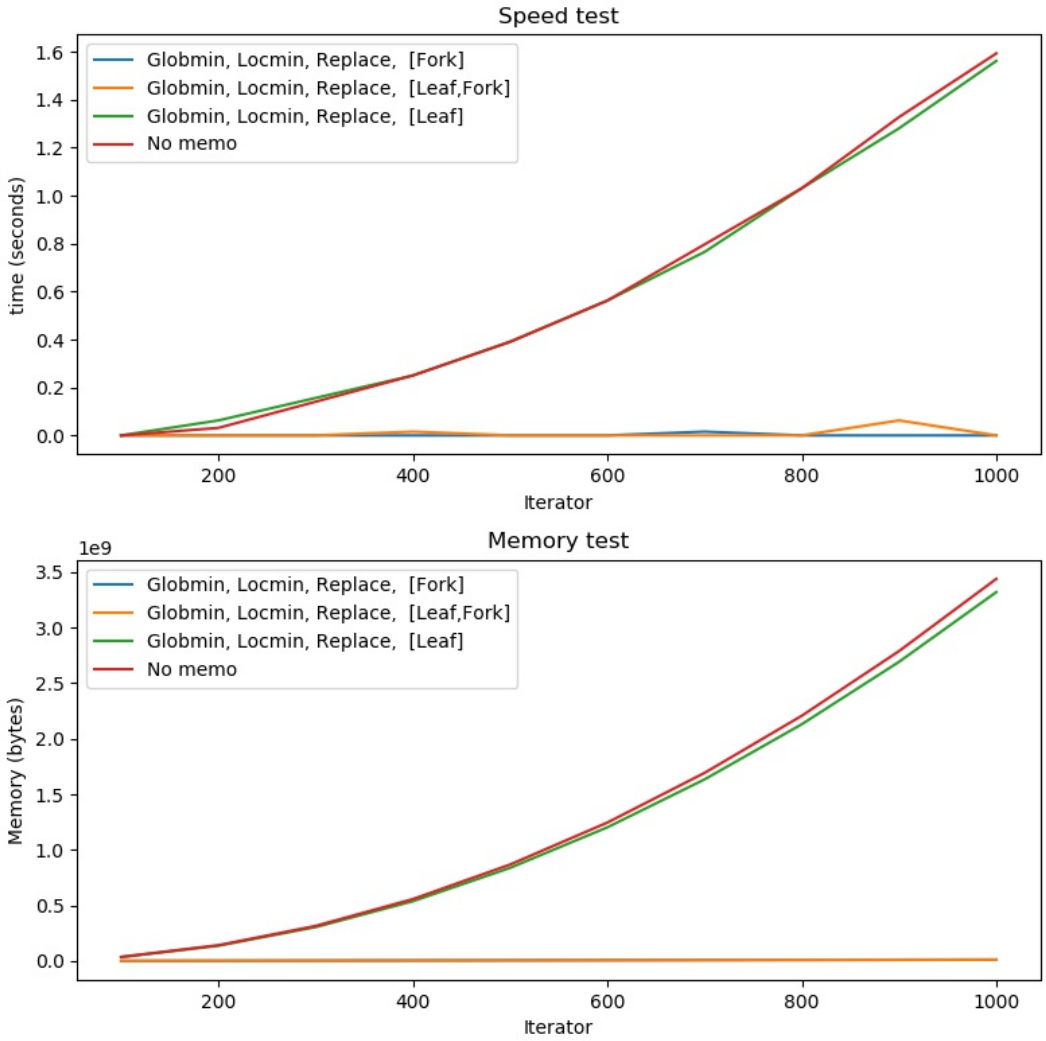


Figure 4.1: repmin resultado por nodo

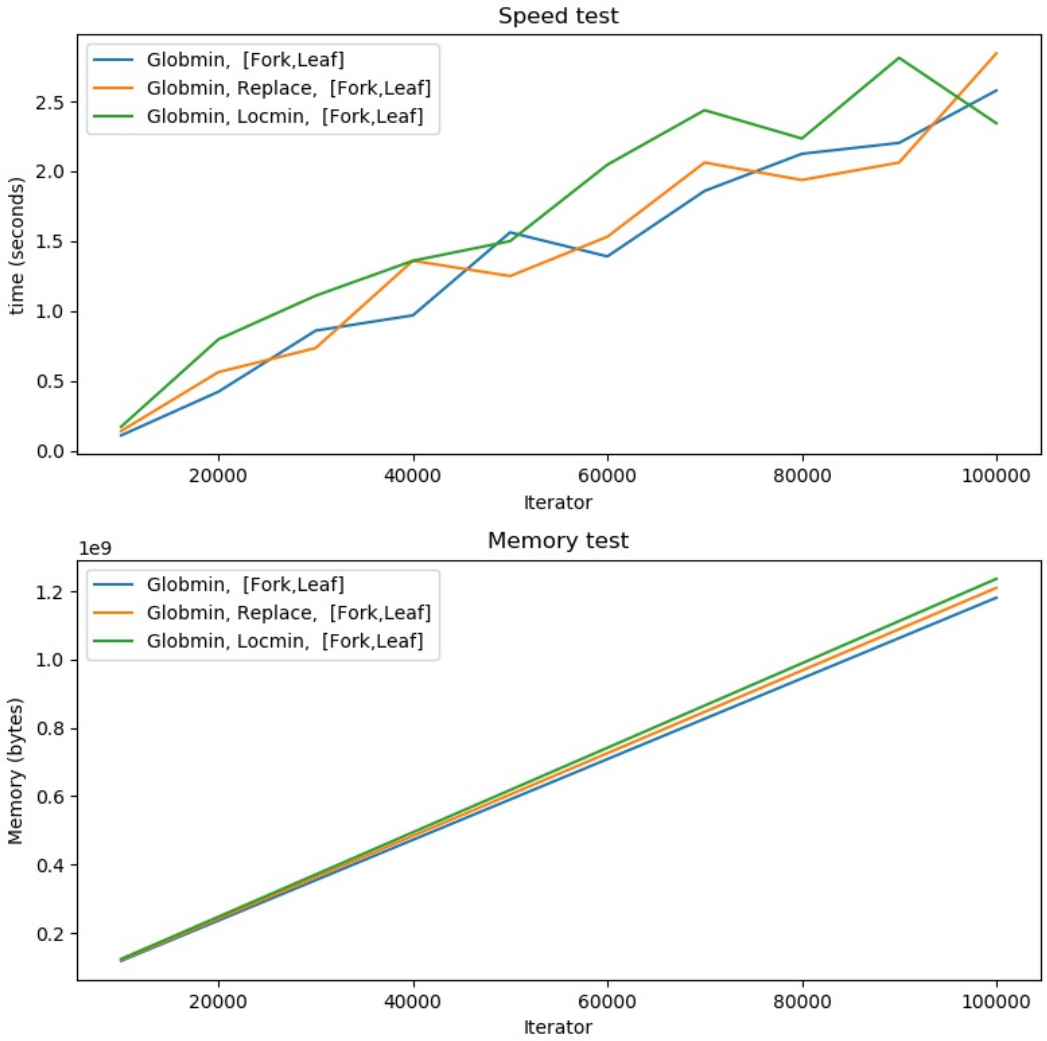


Figure 4.2: repmin resultado por atributo

se memoriza innecesariamente ya que no se vuelve a utilizar.
En la tercera prueba realizamos un testeo exhaustivo con las 22 instancias, tomando únicamente el resultado de las tres primeras. La mejor estrategia resultó ser la esperada, aquella que memoriza únicamente el atributo *globmin* en los nodos *Fork*s, como se observa en la Figura 4.3.
El hecho de memorizar un atributo en un nodo genera un overhead tanto en tiempo de ejecución como en consumo de memoria, debido a las operaciones de almacenamiento y consulta de los resultados. Este overhead tiene que justificarse generando una mejora en el rendimiento global del algoritmo. Es decir, tiene que valer la pena memorizar ese atributo en ese nodo de tal forma que la AG mejore respecto a la alternativa de no memorizarlo. En muchos casos, este overhead no se justifica, ya sea porque el valor memorizado en ese nodo no se utiliza nunca, o se utiliza muy poco y no termina siendo conveniente su memorización.
Por último vamos a comparar para *repmin* el caso óptimo de memorización hallado ([*Fork*], [*globmin*]) versus el caso original, es decir aquel que no utiliza memorización. Como se observa en la Figura 4.4, la ganancia es significativa en ambos resultados, siendo el crecimiento de ([*Fork*], [*globmin*]) lineal y cercano a cero mientras que el de la estrategia *NoMemo* crece exponencialmente.
Recordemos que en [4] se menciona que ganar en tiempo de ejecución implica perder en consumo de memoria. En este caso el resultado es el opuesto, se observa una ganancia en ambos casos. La diferencia radica en que nuestra prueba considera el consumo total de memoria, es decir la integral de la gráfica de [4]. Mientras que en [4] se considera únicamente el mayor pico alcanzado en la ejecución, que resulta inferior para el caso *NoMemo*.
Existe un equilibrio entre los consumos de memoria que provocan las distintas estrategias. Por un lado tenemos la memoria utilizada para guardar resultados de atributos, es decir al emplear memorización, y por el otro si no memorizamos consumimos memoria por cada llamada recursiva que hagamos. Consideremos por ejemplo la versión *FullMemo*, uno podría pensar que al memorizar todos los atributos en todos los nodos del árbol estamos consumiendo más memoria que la versión *NoMemo*. Pero hay que tener en cuenta que ésta última, al no memorizar resultados y tener que recorrer el árbol entero por cada llamada a *globmin*, consume memoria para navegar la estructura de datos, porque tiene que guardar por cada movimiento el contexto actual del nodo, es decir los subárboles asociados. Esto en definitiva está provocando un consumo mucho mayor comparado con la memorización de atributos en sí, y como se observa en la Figura 4.4, el crecimiento de la curva para la versión *NoMemo* es exponencial mientras que la versión *FullMemo* se mantiene lineal.
En resumen, la conclusión hallada es que el rendimiento por parte de la instancia ([*Fork*], [*globmin*]) es mejor que la versión que no aplica memorización tanto en términos de tiempo de ejecución como de consumo de memoria. Esto se debe a que navegar el árbol por completo en cada llamada

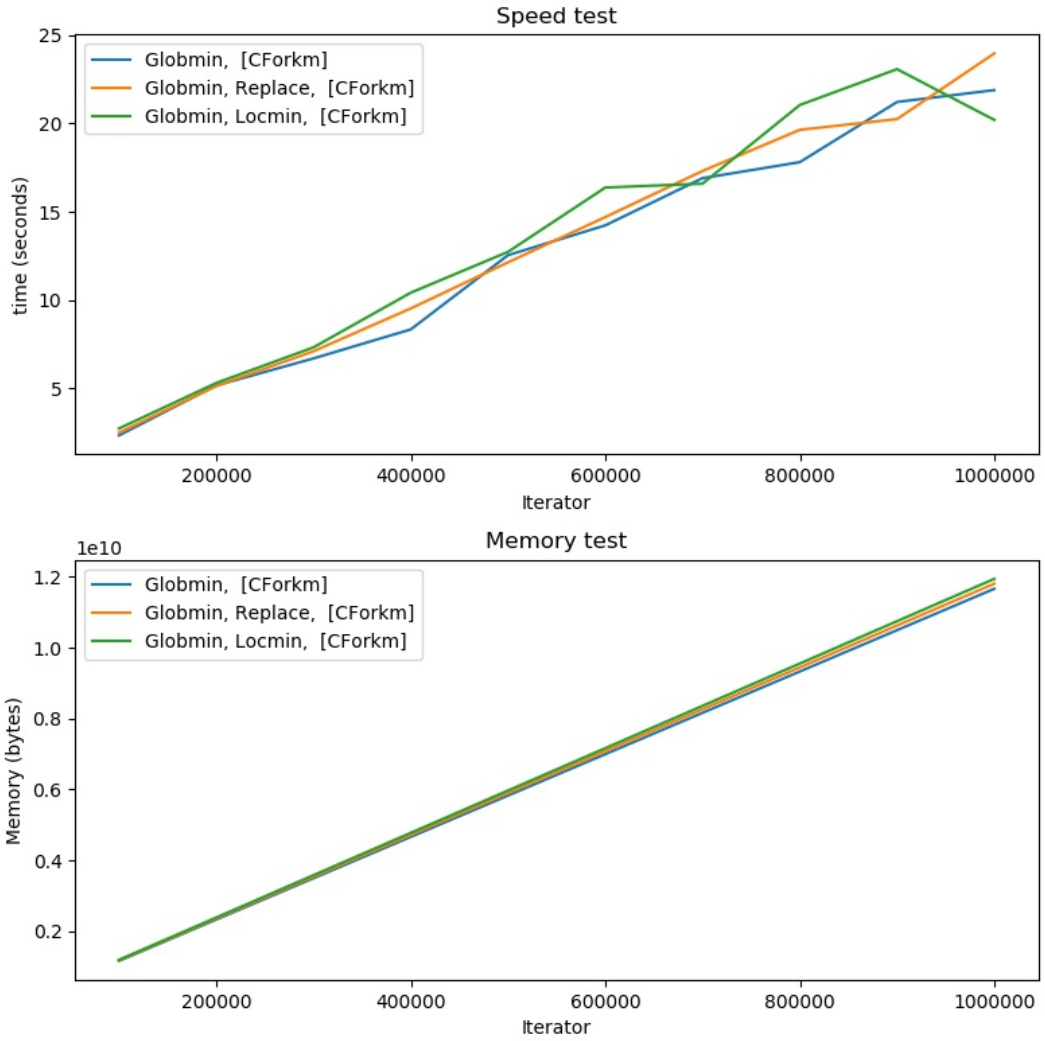


Figure 4.3: repmin mejores estrategias

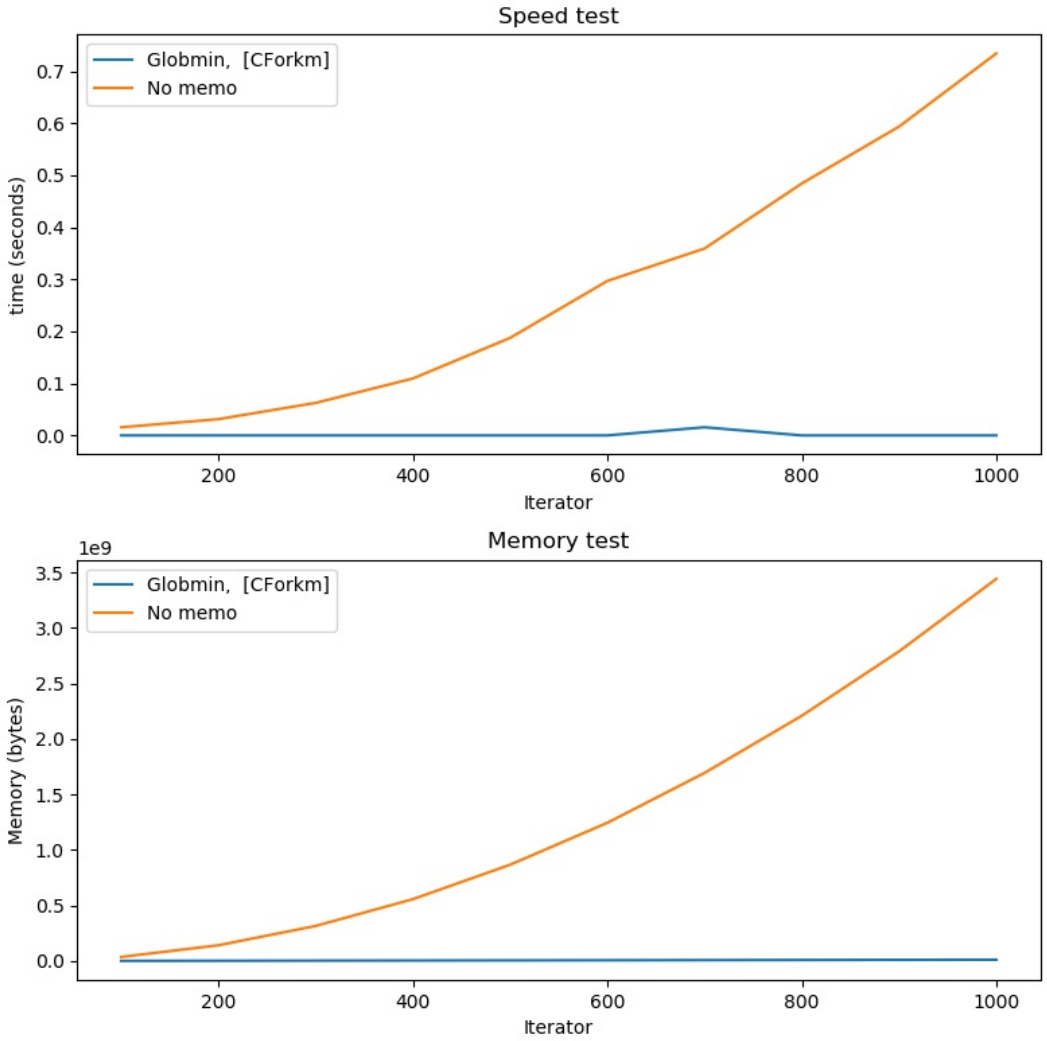


Figure 4.4: repmin memo vs no memo

a *globmin* resulta más costoso que memorizar los resultados en las tablas de memorización. Por otro lado, existe una mejora en ambas mediciones al aplicar memorización selectiva respecto a la memorización completa. Esto se debe a que estamos evitando el overhead y consumo innecesario de memoria para aquellos atributos que no son necesarios memorizar.

4.2 Compilador KLanguage

4.2.1 Introducción

En este ejemplo vamos a presentar una AG que realiza la tercera fase del ciclo de un compilador, el análisis semántico, particularmente aplicado a un lenguaje de programación llamado *KLanguage*. *KLanguage* es un lenguaje de programación imperativo construido como ejemplo ficticio para este proyecto. Permite definir constantes y variables de distintos tipos como números enteros, variables booleanas y strings. Podemos asignar valores a estas variables y escribir expresiones complejas utilizando operadores de suma, resta, multiplicacion, and, or y concat para el caso de los strings. También podemos definir bloques *If* y *LetIn*. Los bloques *LetIn* almacenan en una variable dada el resultado de ejecutar un bloque de código, y siempre contienen una sentencia *Return* al final. Dentro de los bloques *If* y *LetIn* se consideran las definiciones de variables declaradas hasta el momento y un scope local, donde las declaraciones hechas en este último son tomadas en cuenta sólo dentro del bloque.
Algunos ejemplos de programas escritos en este lenguaje se pueden observar en las Figuras 4.5 y 4.6.

Figure 4.5: Programa 1

```
Define Var x Bool;
Let {
    Define Var h String;
    h = "local";
    k = False;
    Return k;
} In x;
Define Var y String;
Return y;
```

Como se explicó en la Sección 1.1, dentro de la compilación de un programa existen varias fases. Primero los programas deben ser validados léxicamente, luego estructuralmente y por último semánticamente. Los dos ejemplos presentados cumplen las dos primeras fases correctamente, pero presentan errores semánticos en la tercera fase. En *Programa 1* la variable *k* dentro del

Figure 4.6: Programa 2

```
Define Var x String;
Define Var y Bool;
x = "Hello World";
y = x And False;
Return y;
```

bloque *LetIn* se utiliza sin haber sido definida. Mientras que en el segundo ejemplo podemos observar que se utiliza una variable de tipo *String* (x) en una expresión booleana. Estos errores deben ser detectados por el compilador en tiempo de compilación y devolver mensajes del tipo "Variable k is not defined." y "Variable x has incorrect type.".
Este trabajo es el que realiza nuestra AG, dado un programa ya validado léxica y estructuralmente, debemos realizar un chequeo sobre cada variable y constante que aparezca en una una expresión de cualquier tipo.
Tomando como ejemplo el Programa 2, luego de las primeras dos fases mencionadas nuestra AG recibe como input un término de la siguiente forma:

```
Program_2 =  Root (
  DotComma  (Define (Var "x") TString)
  (DotComma  (Define (Var "y") TBool)
  (DotComma (Assign (Var "x") ( StringExp ( StrConst "Hello World" ))
  (DotComma (Assign (Var "y") ( BoolExp (And ( BoolExp (Var "x")) (BoolConst False) ))
    (Return (Var "y"))))))))
```

Que representa el árbol de sintaxis abstracta (AST) del programa escrito según las reglas de la gramática. Debemos recorrer esta estructura en busca de variables duplicadas y no definidas así como validar que los tipos de las variables utilizadas en todas las expresiones sean los correctos. Para esta tarea la AG realiza las siguientes acciones :

- El atributo sintetizado *compile* recorre el AST del programa para chequear todas las variables y constantes que aparezcan. En cada aparición se chequea su tipo mediante el atributo *env*.
- El atributo heredado *env* contiene una lista con todas las variables definidas hasta el momento con sus tipos correspondientes.

- Para cada definición de una variable, se chequea *env* para verificar que no haya sido definida anteriormente.
- Para cada ocurrencia de una variable en una expresión de un determinado tipo, se chequea contra *env* que la variable tenga el tipo correcto.

En la Figura 4.7 se muestra el flujo de ejecución del atributo *compile* sobre Programa 1, particularmente en el nodo donde se define la variable *y*. Se observa que una vez detectado el nodo *Define "y"* dentro del flujo de *compile*, se invoca al atributo *env* para consultar las definiciones de variables hasta el momento. Para calcular *env*, si se trata de un nodo distinto a *DotComma*, se retorna el valor de *env* en el nodo padre. Si se trata de un nodo de tipo *DotComma*, primero se invoca *env* en el padre al igual que en el caso anterior. Luego se invoca sobre su hijo derecho la función *getDef*, que chequea si en esa línea hay una definición (tipo *Define*). En caso de haber una definición, la agrega a la lista retornada anteriormente, en caso contrario se ignora esta acción.
En el Anexo C se encuentra el código principal de la AG así como el tipo de dato *Table*. La idea más intuitiva al analizar el còdigo es memorizar el atributo *env* en algún lado para no tener que re-calcularlo en cada chequeo. Vamos a dejar fuera el resto de los atributos porque ya sabemos que no nos van a servir, por ejemplo *compile* se ejecuta una única vez. Respecto a los nodos, es difícil a simple vista darnos cuenta en qué nodos memorizar. La lista es extensa, son 27 en total, algunos de ellos son : *CRoot, CDotComma, CDefine, CAssign, CSum, COr, CAnd*, etc. Tenemos para este ejemplo, 27 + 1 (NoMemo) = 28 estrategias posibles de memorización.
El input de la AG en las pruebas son programas del estilo de *Programa 1* y *Programa 2* generados aleatoriamente utilizando el módulo *QuickCheck* [8] y combinando aleatoriamente todos los tipos de sentencias, con una frecuencia equitativa para cada una de ellas. El iterador utilizado es la cantidad de nodos de la estructura de entrada y su crecimiento al igual que en el ejemplo anterior es siempre lineal.

4.2.2 Pruebas

En la primera prueba memorizamos el atributo *env* y probamos con cada nodo por separado, quedándonos con aquellos que mejores la performance, ya que probar las combinaciones de los 27 nodos nos resulta complejo por el error que provoca. A esta estrategia de probar caso por caso en busca de un óptimo global se la conoce como *greedy* o algoritmo voraz. Esta estrategia de testeo no conduce a una solución global óptima pero se aproxima a ella en un tiempo razonable, hablaremos mas adelante de este tema.
El resultado de la primera prueba se muestra en la figura 4.8. Podemos observar que la mayoría de

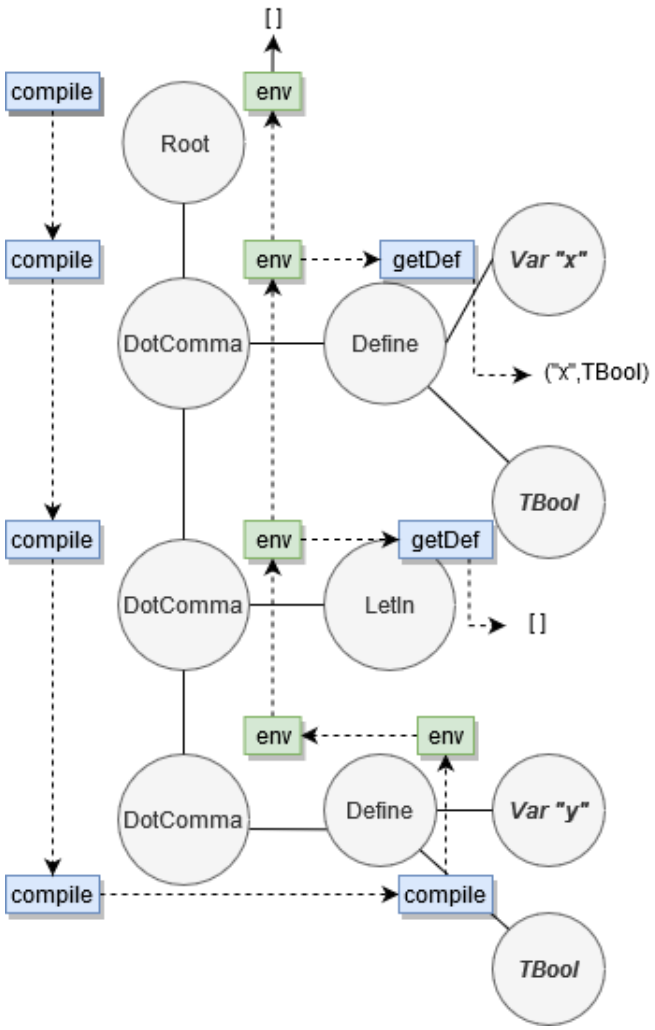


Figure 4.7: KLanguage flujo compile en nodo (Define Var "x")

los nodos parecen ser mejores que la estrategia *NoMemo*, en particular el nodo *DotComma* resulta ser el mejor.

Para la segunda prueba lo que se hizo fue tomar el nodo *DotComma* como base y combinarlo con el resto de los nodos probando uno por uno, viendo si disminuían los tiempos o aumentaban. Se llegó a la conclusión que el conjunto de nodos *[CDotComma, CAssign, CIf, CLetIn, CConcat, CSum, CMult, COr, CIf, CAssign, CAnd, COr]* es el que menor tiempo consume (y también memoria). Comparándolo con la estrategia de memorizar únicamente *DotComma*, el resultado de esta segunda prueba se muestra en la Figura 4.9. Como observación podemos mencionar que la mejor estrategia resulta ser aquella que memoriza en los nodos internos del árbol y no las hojas.

Por lo tanto la mejor estrategia que hemos hallado es la recién presentada, y comparándola con aquella que no utiliza memorización, el resultado se muestra en la Figura 4.10. Se puede observar una notoria ganancia en ambas mediciones.

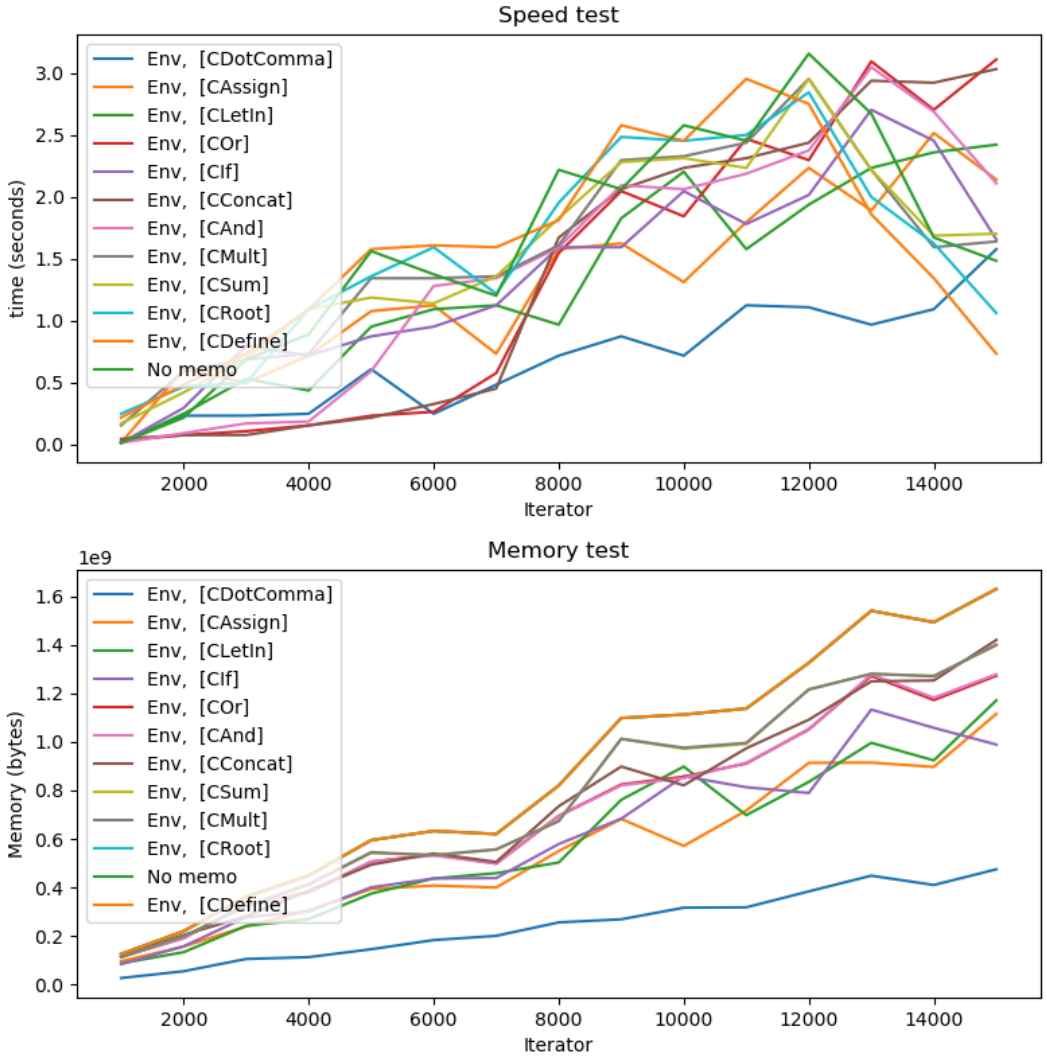


Figure 4.8: KLanguage mejores estrategias por nodo

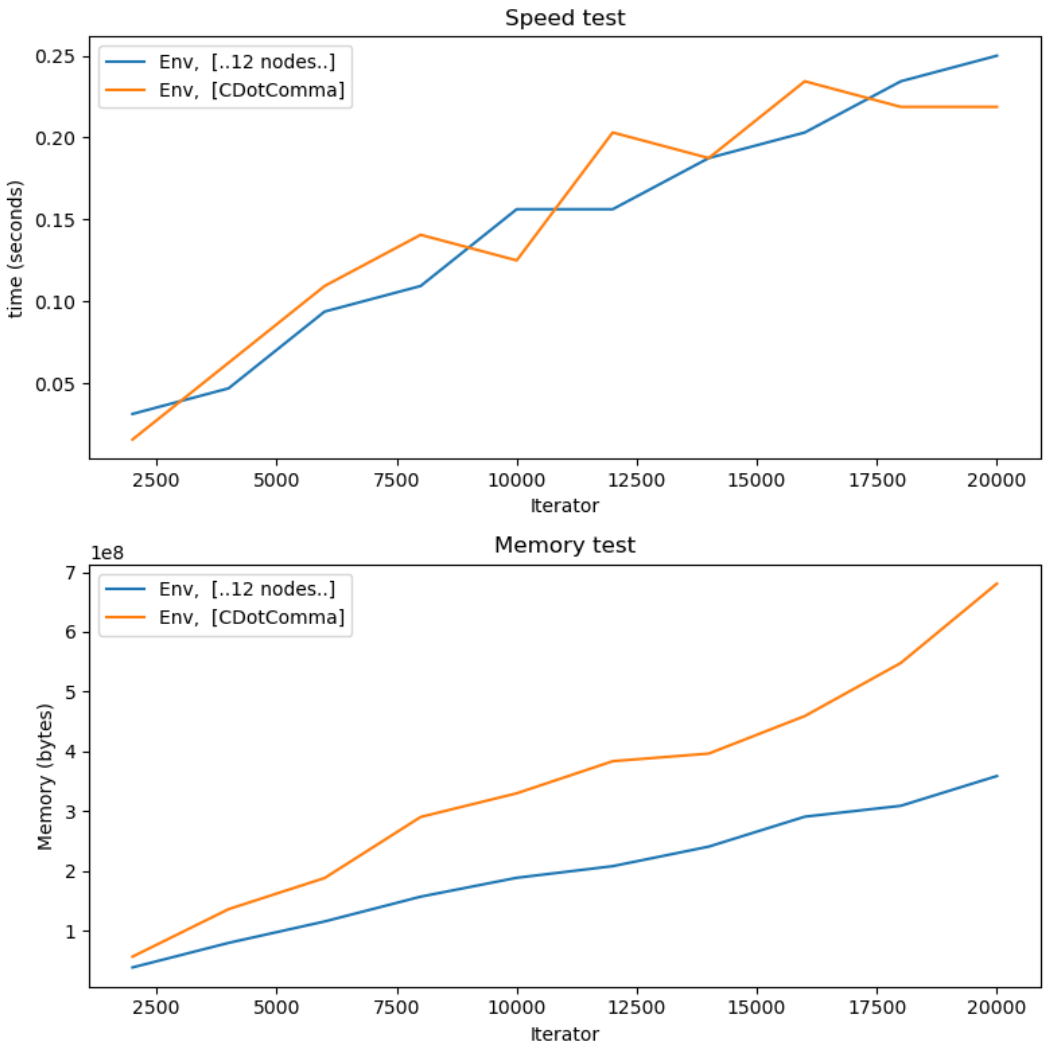


Figure 4.9: KLanguage DotComma vs DotComma + otros nodos

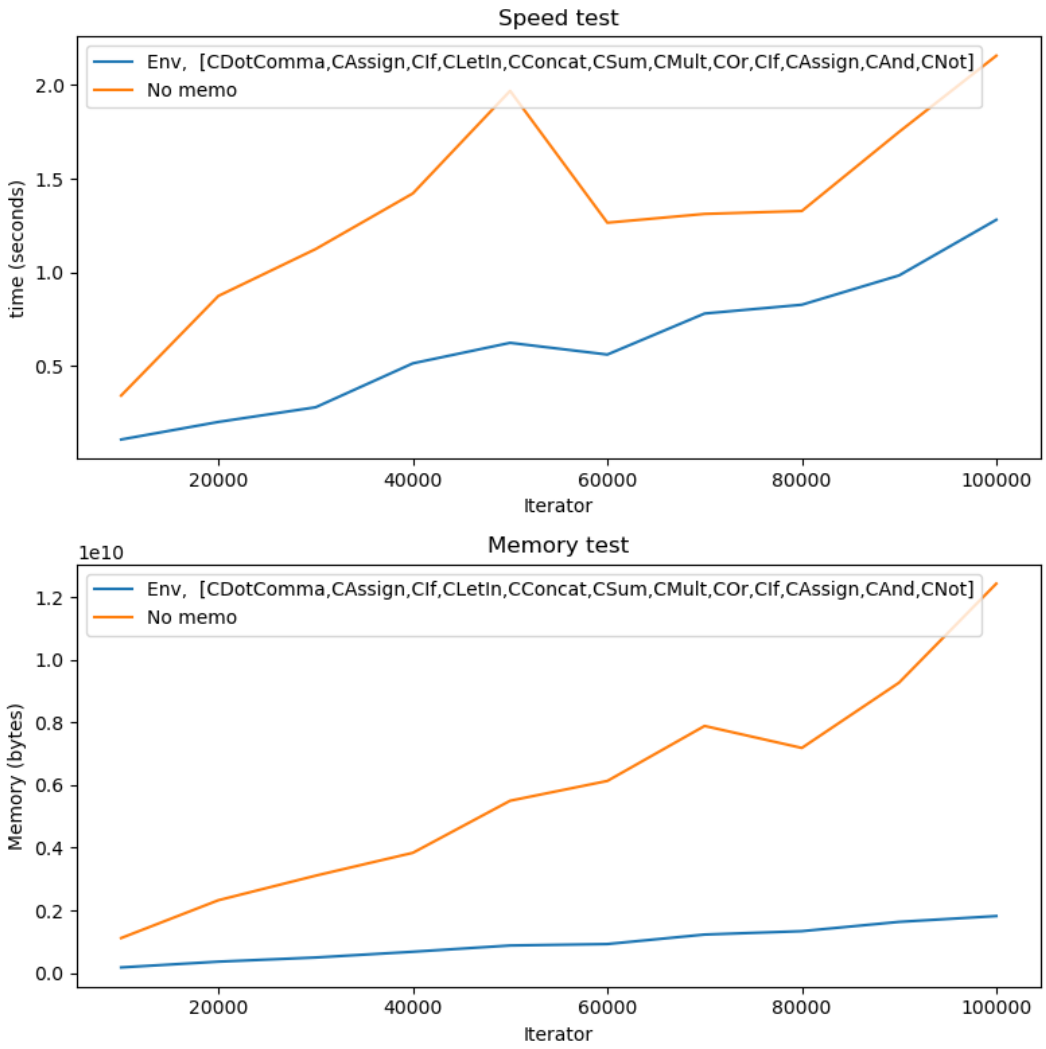


Figure 4.10: KLanguage memo vs no memo

4.3 HTMLFormatter

4.3.1 Introducción

Por último presentaremos la AG más compleja que estudiamos en este proyecto, HTMLFormatter. El código inicial fue tomado de [12]. Esta AG recibe como input una estructura que representa un HTML simple y lo renderiza en la consola. Este HTML contiene tablas que a su vez contienen filas, columnas, casillas con strings y casillas con tablas anidadas. Por lo tanto podemos formar estructuras bastante complejas y de varios niveles de profundidad.
La AG se encarga de calcular las medidas de cada fila y columna a renderizar según los tamaños de los strings en sus casillas y los tamaños de sus tablas anidadas. Un ejemplo de un HTML con algunos textos y una tabla anidada es el siguiente:

```
<!DOCTYPE html>
<html>
<body>
    <table>
      <tr>
        <td>"This is some text on a table!"</td>
        <td>
          <table>
            <tr>
              <td>Some more random text!</td>
            </tr>
            <tr>
              <td>Some more random text!</td>
            </tr>
            <tr>
              <td>Some more random text!</td>
            </tr>
          </table>
        </td>
      </tr>
      <tr>
        <td>"And even more random text!"</td>
```

```
        <td>"This is a big phrase just to make sure works."</td>
      </tr>
      <tr>
        <td>This is a big phrase etc etc.</td>
      </tr>
    </table>
</body>
</html>
```

Y su representación como tipos algebraicos en Haskell:

```
table = RootR (RootTable
        (ConsRow (OneRow (ConsElem (elem1) (ConsElem (NestedTable nt) (NoElem))))
        (ConsRow (OneRow (ConsElem (elem2) (elem3))) (
        ConsRow (OneRow (ConsElem (TableText "This is a big phrase etc etc.") NoElem))
        (NoRow)))))

elem1 = TableText "This is some text on a table!"
elem2 = TableText "And even more random text!"
elem3 = ConsElem (TableText "This is a big phrase just to make sure works.")
        (NoElem)
ny = RootTable
        (ConsRow (OneRow (ConsElem (TableText "Some more random text!")(NoElem)))
                 (NoRow))
```

Siendo esta última el input de nuestra AG. Por último, el output generado por la AG en consola es el siguiente:

```
||-------------------------------------------------------------------------
||This is some text on a table!||||----------------------                  ||
||                             ||||Some more random text!||                 ||
||                             ||||----------------------                  ||
||-------------------------------------------------------------------------
||And even more random text!   ||This is a big phrase just to make sure works.||
||-------------------------------------------------------------------------
||This is a big phrase etc etc.||                                           ||
||-------------------------------------------------------------------------
```

Como podemos observar, en el código representado con tipos algebraicos a priori no tenemos información alguna sobre los anchos y altos a utilizar en la renderización. Todo debe ser calculado a partir de los strings que la estructura contenga, la cantidad de columnas, filas y sus tablas anidadas.
Los atributos para esta AG son 16 y los nodos 9. Tenemos un total de 65535 * 511 = 33.488.385 estrategias posibles. Nuevamente debemos acotar las pruebas de alguna forma.

4.3.2 Pruebas

El código de esta AG puede dividirse en dos partes. Estas dos partes a su vez utilizan conjuntos independientes de atributos, lo cual vamos a tener en cuenta para las pruebas.

1. Parte 1: El algoritmo recibe la estructura del HTML y a partir de ella se normaliza la cantidad de columnas. Con normalizar la cantidad de columnas nos referimos a que una fila puede tener menos elementos que otra, entonces se crean las columnas necesarias restantes con un texto vacío para normalizarlas. El flujo de la AG para esta primera parte puede observarse en la Figura 4.11.

2. Parte 2: Es la encargada de ejecutar el resto del algoritmo. Es decir, calcular anchos y altos de cada columna y fila según los tamaños de los elementos y renderizar el resultado en consola.

Comenzaremos probando la Parte 1 del algoritmo por separado, ya que no depende de la segunda. Analizaremos el resultado obtenido empíricamente y verificaremos con el flujo de la Figura 4.11 (creado a partir del análisis del código) que el resultado sea coherente.
Por último vamos a probar el algoritmo entero considerando las dos partes.

4.3.3 Parte 1

En la Figura 4.11 se observa el flujo de esta primera parte del algoritmo aplicado a un HTML que contiene dos filas simples, con 1 y 2 elementos respectivamente. Esta parte de la AG ejecuta 3 atributos:

1. *r2*: Atributo sintetizado que recorre el árbol para normalizar. Cada vez que se encuentra con un elemento de tipo *NoElem* se llama a *ane_Inh* para consultar la cantidad de columnas global. Luego agrega las columnas necesarias. Para cada *NestedTable* se invoca a sí mismo nuevamente.

2. *ane_Inh*: Atributo heredado que retorna la cantidad de columnas globales de la tabla. Al ser un atributo heredado, consulta el valor a su padre hasta llegar a la raíz de la estructura, donde invoca al atributo *n_Syn*.

3. *n_Syn*: Atributo sintetizado que recorre el árbol y retorna el máximo número de columnas de una fila en la tabla.

Como se observa en la Figura 4.11, *r2* recorre el árbol y en cada ocurrencia de un nodo *NoElem* invoca al atributo *ane_Inh*. *ane_Inh* (flujo verde) comienza a subir en la estructura hasta encontrar el nodo *RootTable*, en el cual invoca al atributo sintetizado *n_Syn* (azul). Por último *n_Syn* recorre el árbol y retorna el máximo de la cantidad de columnas de cada fila.
El input ejemplo consta de 2 filas, lo que provoca que el atributo *ane_Inh* sea invocado dos veces al final de cada una. Estos dos flujos de *ane_Inh* comienzan a subir hasta encontrarse en el nodo de tipo *ConsRow*, marcado con verde oscuro. En este nodo de la estructura se solapan dos flujos y luego como podemos ver se duplica también el flujo del sintetizado *n_Syn*. Por lo tanto vamos a querer memorizar el resultado de la primer ejecución en algún nodo, posiblemente en el recién mencionado *ConsRow*, y consultarlo en la segunda recorrida.
Para la primer prueba, se testearon todos los atributos por separado memorizando en todos los nodos. Solo para verificar que nuestro planteo es correcto y los atributos *ane_Inh* y *n_Syn* resultan útiles de memorizar. El resultado se observa en la Figura 4.12 y los tiempos de las dos primeras instancias (*ane_Inh* y *n_Syn*) son sensiblemente mejores que las del resto.
En la segunda prueba (Figura 4.13) testeamos las distintas combinaciones entre estos dos atributos y nuevamente dejamos los nodos fijos. La mejor instancia resultó ser aquella que memoriza únicamente el atributo *ane_Inh*.
Como tercera y última prueba en esta primera etapa vamos a testear las distintas estrategias que resultan de cruzar *ane_Inh* con todas las combinaciones entre los nodos de la estructura. El resultado se observa en la Figura 4.14, donde concluimos que la mejor estrategia de memorización para la Parte 1 del algoritmo resulta ser aquella que memoriza el atributo *ane_Inh* únicamente en el nodo *ConsRow*, lo cual coincide con nuestra suposición inicial analizando el flujo de ejecución en el diagrama.

4.3.4 Parte 2

Vamos a realizar ahora tres pruebas sobre la AG completa, incluyendo dentro del conjunto de atributos de la primera parte únicamente *ane_Inh*, que pertenece a la mejor estrategia.
En la primera testeamos todos los atributos por separado dejando fijos los nodos. El resultado se

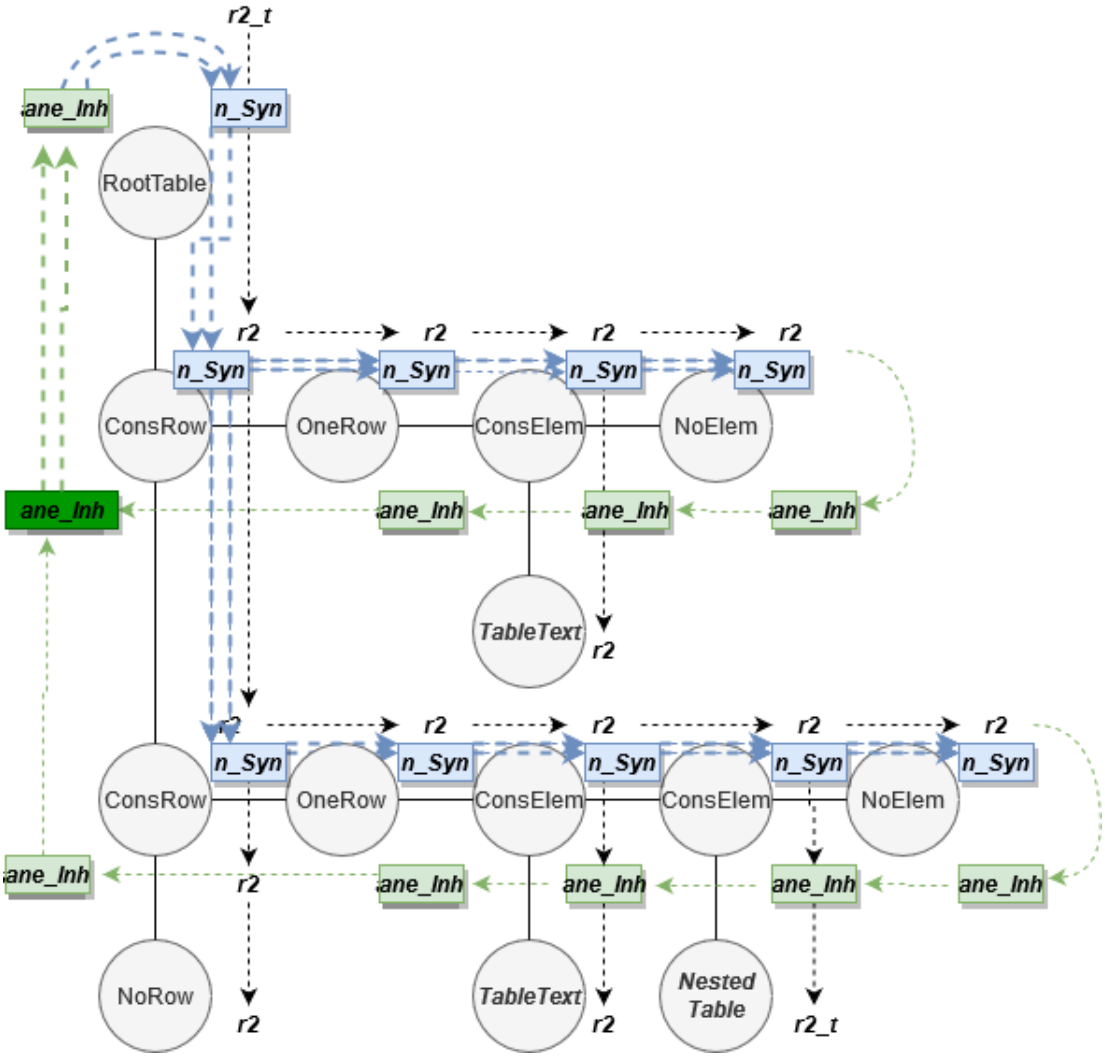


Figure 4.11: HTMLFormatter diagrama r2

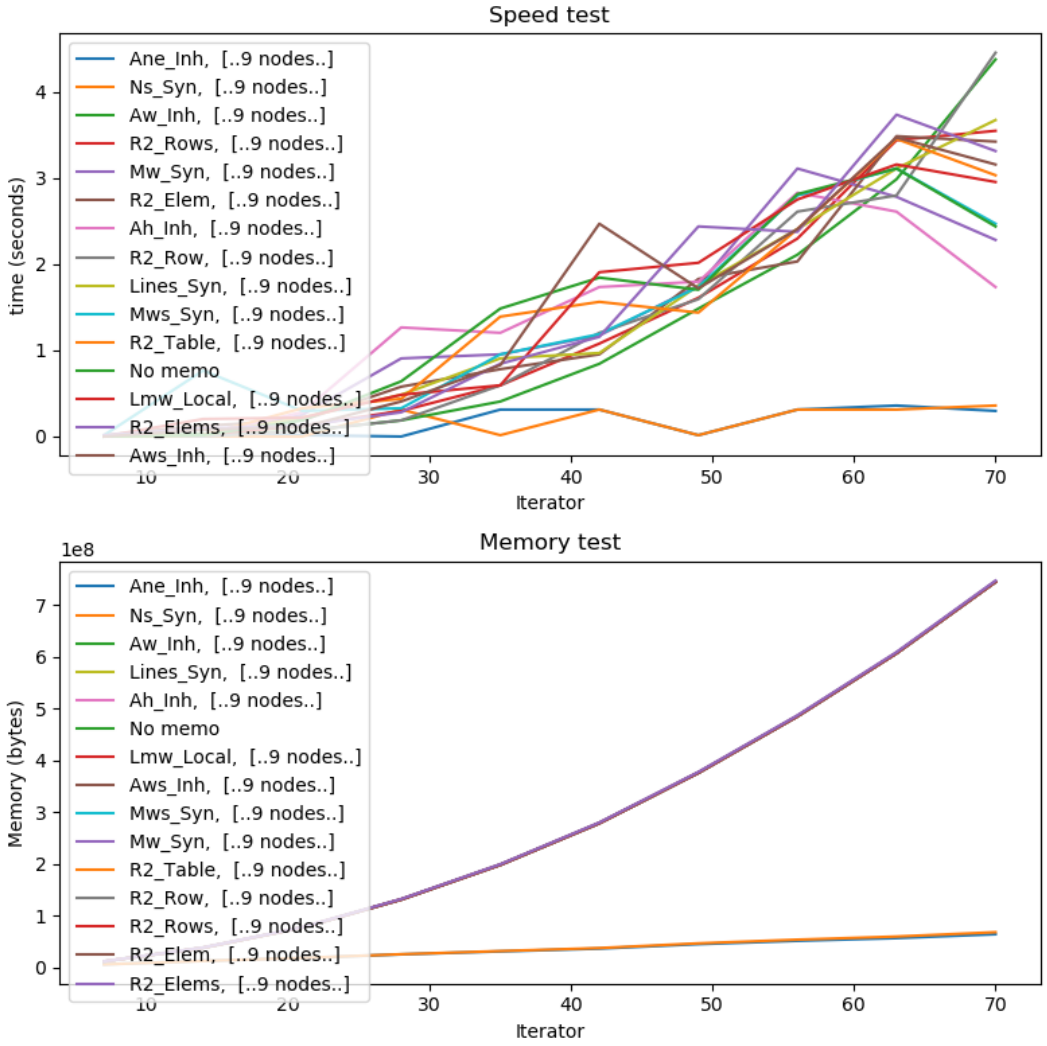


Figure 4.12: HTMLFormatter mejores atributos

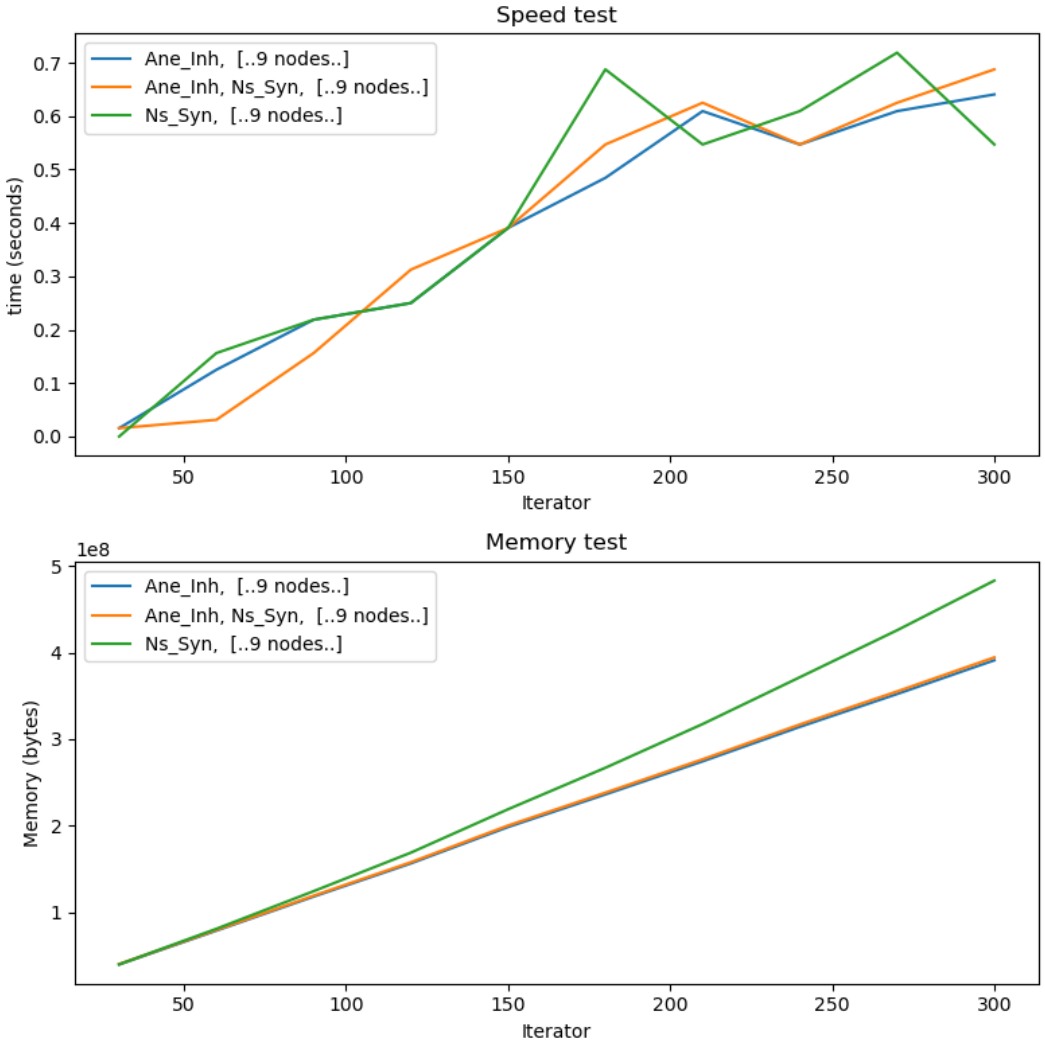


Figure 4.13: HTMLFormatter combinaciones n_Syn y ane_Inh

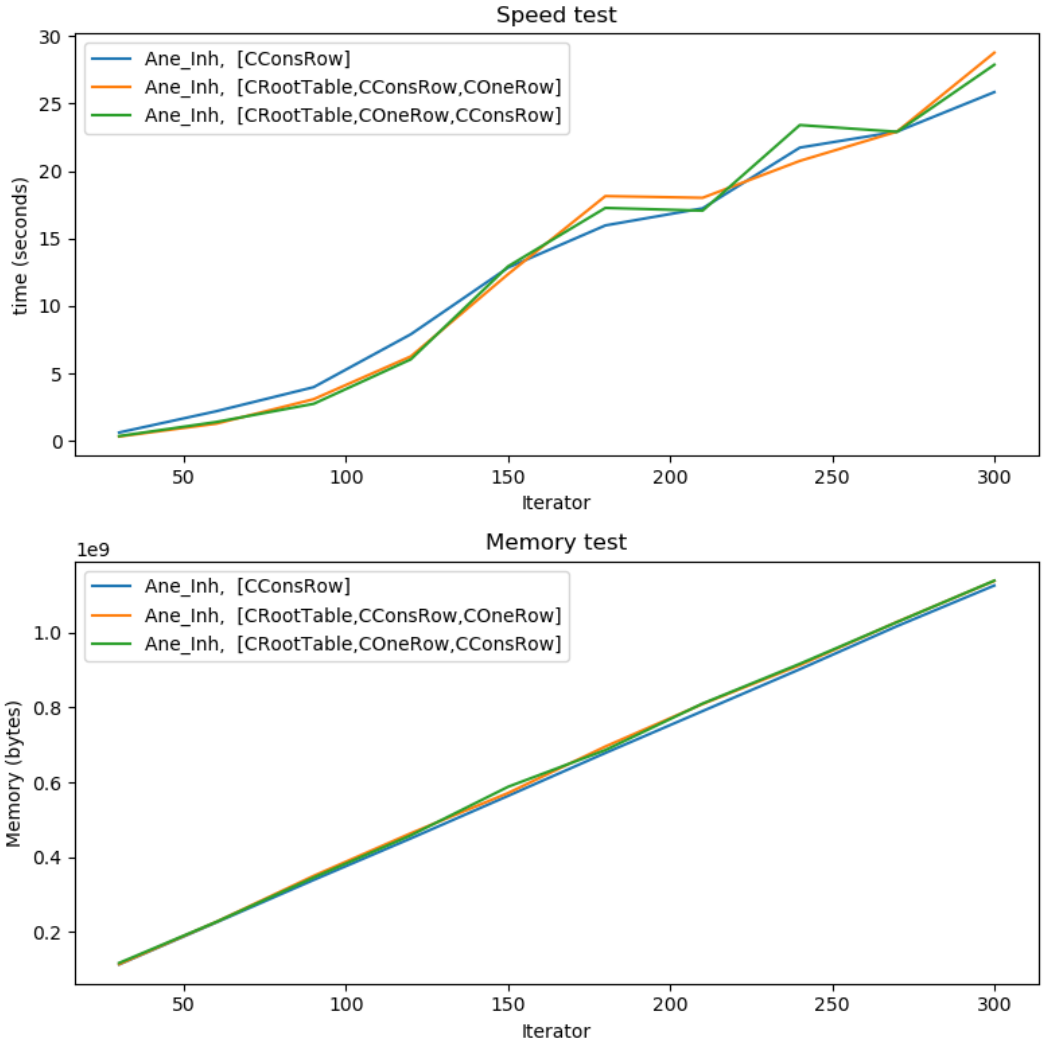


Figure 4.14: HTMLFormatter ane_Inh y combinaciones de nodos

muestra en la Figura 4.15, donde los atributos que resultan útiles de memorizar se encuentran por arriba de la etiqueta *NoMemo*.

Como segunda prueba, tomamos estos atributos y los cruzamos con las distintas combinaciones de la lista de nodos candidatos. Esta última está formada por aquellos nodos que no están en los extremos de la gramática, como por ejemplo *NoElem* o *NoRow*. Es importante aclarar que a lo largo del proyecto se tuvieron que hacer consideraciones de este tipo para acortar la lista de nodos y evitar así probar demasiadas instancias a la vez. Probar con conjuntos de instancias muy grandes provoca un aumento en el error de las mediciones. Cuando se prueban menos instancias, los resultados son más precisos.

El resultado de esta segunda prueba se muestra en la Figura 4.16 y podemos concluir que el conjunto de atributos [*CConsRow , COneRow , CConsElem, CNestedTable*] resultó ser el mejor. Notar que dentro de la lista se encuentra *ConsElem* que fue el hallado en las pruebas de la Parte 1.

Como último test, comparamos la mejor estrategia de memorización hallada vs la estrategia *NoMemo*. El resultado se aprecia en la Figura 4.17. Nuevamente se observa una ganancia importante en ambas mediciones al aplicar memorización. Al igual que el caso de *repmin*, la versión *NoMemo* crece exponencialmente mientras que la estrategia propuesta se mantiene lineal y cercana a cero.

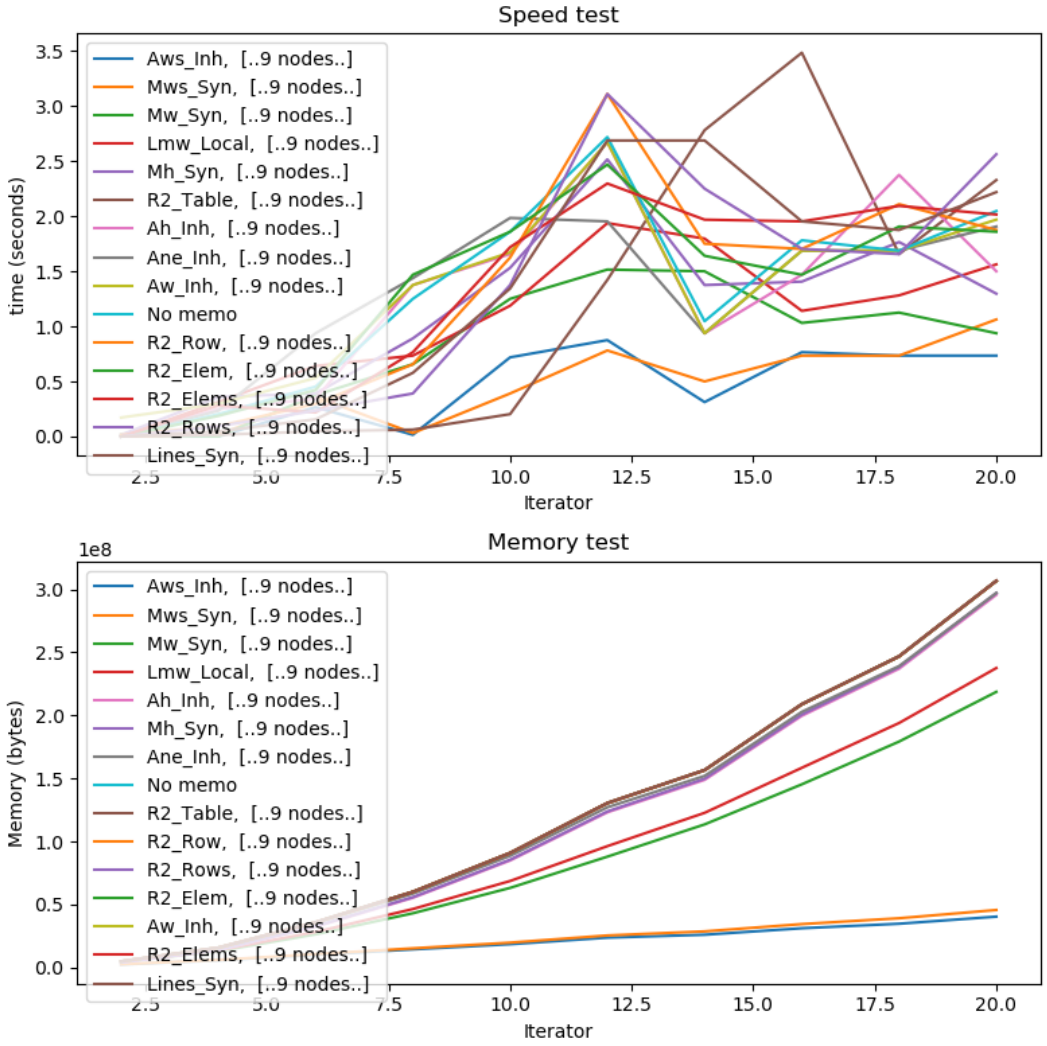


Figure 4.15: HTMLFormatter mejores estrategias por atributo

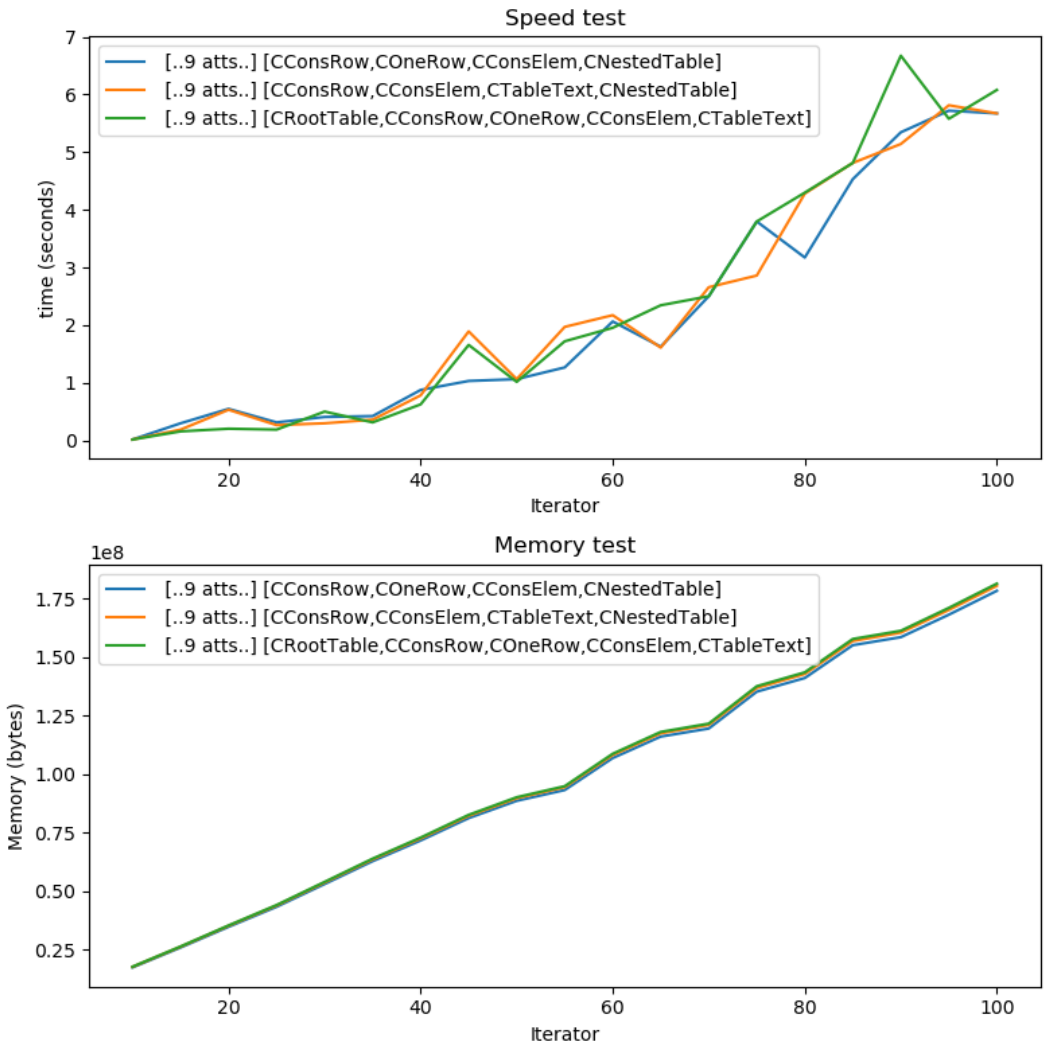


Figure 4.16: HTMLFormatter mejores estrategias por nodo

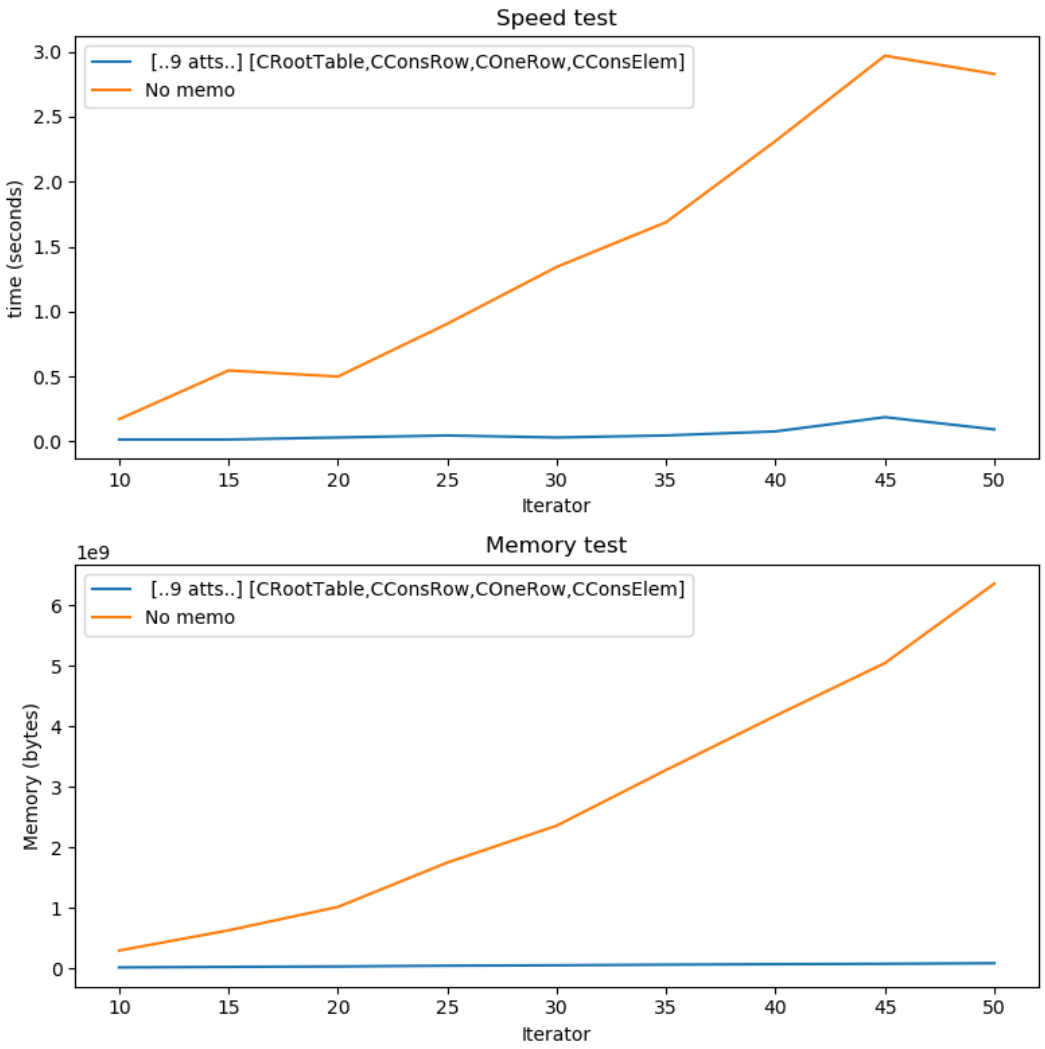


Figure 4.17: HTMLFormatter Memo vs NoMemo

Chapter 5

Conclusiones y trabajos futuros

La primera conclusión en este proyecto es que se desarrolló un método para probar zipper based AGs de forma empírica. Esto nos permitió probar distintas estrategias de memorización y sacar conclusiones a partir de los resultados observados. El módulo tiene la capacidad de correr varias instancias en paralelo y detectar cuales son las mejores, tomando como criterio el tiempo de ejecución. Calcula también el consumo total de memoria para cada instancia y presenta al final de la prueba ambos resultados.

También observamos que en algunos casos la cantidad de estrategias es muy grande y resulta computacionalmente muy difícil detectar cual es la mejor. Para estos casos lo que se hizo fue dejar de lado atributos y nodos que de antemano sabíamos no era necesario incluirlos, lo cual nos disminuyó la cantidad de estrategias sensiblemente. Otra de las alternativas utilizada para estos casos fue aplicar manualmente una estrategia tipo *greedy*. Las estrategias *greedy* o algoritmos voraces parten de un conjunto de elementos e iteran sobre éste quitando elementos y verificando si el resultado global mejora, en caso de mejorar, el elemento es agregado al conjunto solución. El resultado final de un algoritmo de este tipo no es precisamente el resultado óptimo global, pero sabemos que se aproxima. En nuestro caso los elementos utilizados fueron nodos y atributos, y lo que hicimos fue dejar siempre fija una lista e iterar sobre la otra. Notar que con este método no estamos teniendo en cuenta las dependencias que puedan llegar a tener los elementos entre sí.

De las pruebas realizadas podemos concluir que la técnica de memorización selectiva de atributos representa efectivamente una mejora tanto en términos de tiempo de ejecución como de consumo de memoria. Por un lado, hemos propuesto discriminar por tipo de nodo además de por atributo, lo cual nos ha llevado a observar que existe una mejora cuando dejamos fuera ciertos nodos. Además, en contraposición a lo que se plantea en [4], hemos probado que memorizar para ganar en tiempo

de ejecución no implica mayor consumo de memoria. Por el contrario, vimos que al memorizar ganamos en los dos aspectos. Existe un equilibrio entre el consumo de memoria provocado por la navegación mediante *zippers* y la memorización de atributos. Al memorizar un atributo, estamos utilizando memoria para alojar los resultados en la *MemoTable* en los nodos implicados, mientras que al navegar por el árbol (que sería el caso en que optamos por recalcular y no memorizar) estamos utilizando memoria para guardar los contextos de navegación. Estos dos tipos de consumo de memoria generan un equilibrio.

Otra observación importante que hemos realizado a lo largo de las pruebas es que los conjuntos de datos de prueba juegan un rol importante en los resultados y la selección de la mejor estrategia se puede ver sesgada por estos. Es importante generar datos de prueba lo mas aleatorios posible para que las pruebas sean justas.

Por último concluir que la solución de embeber el DSL como un shallow embedding resulta bastante simple de abordar. Así como también la adaptación del código para utilizar el módulo de testeo. Si se trata de una AG que utilice la biblioteca *Generics.Zippers* [13], con pequeñas modificaciones en el código y definiendo una función que genere casos de prueba con *QuickCheck* [8] ya estaríamos en condiciones de probar distintas estrategias de memorización.

Dentro de los trabajos futuros, hay uno que es para nosotros el más interesante y seria de gran ayuda en este proceso de encontrar la mejor estrategia de memorización para una AG dada. Una AG, como se explicó anteriormente, es una gramática libre de contexto decorada con atributos para proveer de la parte semántica y así formar una Attribute Grammar. Esta AG puede ser esquematizada mediante el flujo de ejecución que realiza sobre la estructura de entrada, como se muestra en las Secciones 4.2 y 4.3. Analizando este flujo podrían encontrarse puntos donde un atributo es recalculado (como en 4.3), de hecho, el recálculo de un atributo se da cuando dos flujos distintos se encuentran en un nodo.

Si lográramos realizar éste análisis automáticamente, podríamos encontrar qué atributos son recalculados en qué nodos exactamente, incluso podríamos hacer un conteo de la cantidad de re-cálculos realizados por atributo en cada nodo.

Para lograr esto tendríamos primero que ser capaces de representar una Attribute Grammar mediante un deep embedding DSL en vez de un shallow embedding como el que usamos en este proyecto. De esta forma obtendríamos una representación tipada para una AG, lo que nos permitiría realizar el análisis de flujo deseado. Un trabajo parecido es el realizado en UUAG [18], donde se define un DSL externo para escribir AGs y luego es compilado a código Haskell. En nuestro caso utilizaríamos un EDSL en vez de un DSL externo.

Respecto a este proyecto existen algunas mejoras que se pueden realizar:

1. Hemos mostrado cómo podemos ser selectivos en la memorización tanto por nodo como por atributo. Pero esta solución no contempla el hecho de poder memorizar un atributo en cierto nodo y no en otro dentro de la estrategia seleccionada. Para verlo más claro, consideremos la estrategia que memoriza los atributos *A1* y *A2* en los nodos *N1* y *N2*. Según nuestra implementación, los dos atributos se van a memorizar en los dos nodos, no siendo capaces de memorizar por ejemplo *A1* únicamente en *N1* y *A2* en *N2*. Implementar esto último supondría ser aún más selectivos a la hora de memorizar y por lo tanto obtener una estrategia más efectiva.

2. Sería de gran ayuda para el módulo de testeo poder estudiar y reducir el error generado en las pruebas, sobre todo cuando se prueban conjuntos grandes de estrategias. Esto ha sido una limitante a lo largo del proyecto ya que en contraposición a lo que se planteó inicialmente de testear todas las combinaciones de memorización automáticamente, fue imposible en muchos casos llevar a cabo esta tarea debido al error obtenido. Tuvimos que considerar criterios para seleccionar dentro de la lista de nodos o atributos candidatos, los mejores. También se realizaron pruebas tipo *greedy* por atributos y nodos para evitar probar todas las posibles combinaciones entre ellos. Esto último, si bien facilitó las pruebas, conceptualmente no es del todo correcto ya que no estamos considerando las dependencias entre atributos y/o nodos.

Bibliography

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. In Alex Aiken and Greg Morrisett, editors, *Conference Record of the 30th Symposium on Principles of Programming Languages*, pages 14–25. ACM, 2003.

[2] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Implementation and Application of Functional Languages*, pages 57–74, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] João Paulo Fernandes and João Saraiva. Tools and libraries to model and manipulate circular programs. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '07, page 102–111, New York, NY, USA, 2007. Association for Computing Machinery.

[4] João Paulo Fernandes, Pedro Martins, Alberto Pardo, João Saraiva, and Marcos Viera. Memoized zipper-based attribute grammars and their higher order extension. *Science of Computer Programming*, 173:71 – 94, 2019. Special issue on the Brazilian Symposium on Programming Languages (SBLP '15+16).

[5] Haskell. Control-Monad-Maybe. http://hackage.haskell.org/package/MaybeT-0.1.2/docs/Control-Monad-Maybe.html. June 2020.

[6] Haskell. Data.Map. https://downloads.haskell.org/ ghc/6.12.2/docs/html/libraries/containers-0.3.0.0/Data-Map.html. June 2020.

[7] Haskell. Heterogeneous lists. https://hackage.haskell.org/package/hlist. June 2020.

[8] Haskell. Test.QuickCheck. https://hackage.haskell.org/package/QuickCheck-2.13.2/docs/Test-QuickCheck.html. June 2020.

[9] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, implementation and evaluation of the FNC-2 attribute grammar system. *SIGPLAN Not.*, 25(6):209–222, June 1990.

[10] Uwe Kastens and Carsten Schmidt. Vl-eli:: A generator for visual languages. *Electronic Notes in Theoretical Computer Science*, 65(3):139 – 143, 2002. LDTA 2002, Second Workshop on Language Descriptions, Tools and Applications (Satellite Event of ETAPS 2002).

[11] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[12] Pedro Martins. HTMLFormatter code from ZipperAG: An implementation of Attribute Grammars using Functional Zippers. https://hackage.haskell.org/package/ZipperAG.

[13] Pedro Martins, João Paulo Fernandes, and João Saraiva. Zipper-based attribute grammars and their extensions. In André Rauber Du Bois and Phil Trinder, editors, *Programming Languages - 17th Brazilian Symposium, SBLP 2013, Brasília, Brazil, October 3 - 4, 2013. Proceedings*, volume 8129 of *Lecture Notes in Computer Science*, pages 135–149. Springer, 2013.

[14] Arie Middelkoop, Atze Dijkstra, and S. Doaitse Swierstra. Iterative type inference with attribute grammars. In Eelco Visser and Jaakko Järvi, editors, *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, pages 43–52. ACM, 2010.

[15] Python. Tuning the hyper-parameters of an estimator. https://scikit-learn.org/stable/modules/grid$_search.html.June$2020.

[16] João Saraiva and S. Doaitse Swierstra. Generating Spreadsheet-Like Tools from Strong Attribute Grammars. In Frank Pfenning and Yannis Smaragdakis, editors, *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*, volume 2830 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2003.

[17] S. Doaitse Swierstra, Pablo R. Azero Alcocer, Joao Saraiva, Doaitse Swierstra, Pablo Azero, and João Saraiva. Designing and implementing combinator languages. In *Third Summer School on Advanced Functional Programming, volume 1608 of LNCS*, pages 150–206. Springer-Verlag, 1999.

[18] Universiteit Utrecht. Attribute Grammar System of Universiteit Utrecht. https://hackage.haskell.org/package/uuagc. June 2020.

Appendix A

Test Module

```
evalMemo :: Bool -> Int -> [[c]] -> [m] -> [Int] -> ( Int -> Gen t )
                 -> ( m -> [c] -> t -> a ) -> (a -> b)
                 -> IO ()
evalMemo finalLc w (lc:lcs) lm xs g h f =
  do
    --setNumCapabilities 4
    lc'<- if finalLc then return (lc:lcs) else return $ combs lc
    ts  <- genCases xs g
    tsm <- genCasesMemo lc' lm ts h
    trees <- return $ map thd tsm
    evaluate (rnf trees)
    m  <- newEmptyMVar
    startS  <- newEmptyMVar
    let log = Logger m
    iterStrategies startS log xs tsm f
    putMVar startS "start"
    s  <- firstN log w [] -- Wait for first N
    printChart $ sortOn (\x-> sum (map snd' (thd x))) s--1


firstN :: (Show c, Show m) => Logger c m -> Int
                           -> [ ( [c], m,[(Int,Double,Int)]) ]
```

```
                              -> IO [ ( [c], m,[(Int,Double,Int)]) ]
firstN (Logger m) 0 lr = do return lr
firstN (Logger m) n lr =  do
                            cmd <- takeMVar m
                            case cmd of
                              Result msg ->
                                do
                                  firstN (Logger m) (n-1) (lr ++ [msg])

genCases :: [Int] -> ( Int -> Gen t ) -> IO [t]
genCases [] _ = return []
genCases (x:xs) f =  do
                       t  <- generate $ f x
                       ts <- genCases xs f
                       return ([t] ++ ts)

genCasesMemo :: [[c]] -> [m] -> [t] -> ( m -> [c] -> t -> a ) -> IO [(m,[c],[a])]
genCasesMemo [] _ _ _ = return []
genCasesMemo (lc:lcs) lm ts h =  do
                                   t  <- genCasesMemoAux lc lm ts h
                                   ts <- genCasesMemo lcs lm ts h
                                   return (t ++ ts)

genCasesMemoAux :: [c] -> [m] -> [t] -> ( m -> [c] -> t -> a ) -> IO [(m,[c],[a])]
genCasesMemoAux lc [] _ _ = return []
genCasesMemoAux lc (m:ms) ts h =  do
                                    as <- return $ map (h m lc) ts
                                    xs <- genCasesMemoAux lc ms ts h
                                    return ([(m,lc,as)] ++ xs)

iterStrategies :: MVar String -> Logger c m -> [Int]
                              -> [(m,[c],[a])] -> (a -> b)
                              -> IO ()
iterStrategies startS log xs [] f = return ()
```

```
iterStrategies startS log xs (s:ls) f =  do
                                          forkIO $ evalStrategie startS log xs s f
                                          iterStrategies startS log xs ls f
                                          return ()

evalStrategie :: MVar String -> Logger c m -> [Int] -> (m,[c],[a]) -> (a -> b) -> IO ()
evalStrategie startS log xs (m,lc,ts) f =
  do
    tid <- myThreadId
    j <- readMVar startS -- Used to start all the threads at the same time
    t <- getTime ProcessCPUTime
    putStrLn $ "Started " ++ show tid ++ " " ++ show t
    ys <- runTestL ts f
    logResult log (lc, m , zipInto xs ys)

data Logger c m = Logger (MVar (LogResult c m))

data LogResult c m = Result ( [c], m,[(Int,Double,Int)])

logResult :: Logger c m -> ( [c], m,[(Int,Double,Int)]) -> IO ()
logResult (Logger j) s = putMVar j (Result s)

iterCons :: (c -> IO ()) -> [c] -> IO ()
iterCons _ [] = do return ()
iterCons f (c:cs) =  do
                      iterCons f cs
                      runThread f c
                      return ()

runThread :: (c -> IO ()) -> c -> IO ()
runThread f c =  do
                  x <- forkIO $ f c
                  return ()
```

```
runTestL :: [a] -> (a -> b) -> IO [(Double,Int)]
runTestL [] _ = return []
runTestL (x:xs) f =  do
                        (y,ac) <- runTest x f
                        ys <- runTestL xs f
                        return ([(y,ac)] ++ ys )

runTest :: a -> (a -> b) -> IO (Double,Int)
runTest x f =
  do
    setAllocationCounter 0
    start <- getTime ProcessCPUTime -- getCPUTime --
    evaluate $ rnf $ runEval $ rparWith rdeepseq (f x)
    --evaluate $ rnf $ (f x)
     end <- getTime ProcessCPUTime --getCPUTime --
    ac <- getAllocationCounter
    return $ ((fromInteger $ (toNanoSecs (end - start))) / 1000000000, - (fromIntegral


evalNBest :: MVar String -> Logger c m -> [Int] -> [ ( [c], m ) ] ->
                                          [a] -> (a -> b) -> IO ()
evalNBest startS log xs ((lc,m):ls) ts f =
  do
    forkIO $ evalStrategie startS log xs (m,lc,ts)  f
    evalNBest startS log xs ls ts f
    return ()
evalNBest _ _ _ [] _ _ = return ()

----------Other functions----------

thd :: (a,b,c) -> c
thd (_, _, c) = c

snd' :: (a,b,c) -> b
```

```
snd' (_, b, _) = b

zipInto :: [a] -> [(b,c)] -> [(a,b,c)]
zipInto (a:as) ((b,c):bs) = [(a,b,c)] ++ zipInto as bs
zipInto [] _ = []

fst3 :: (a, b, c) -> a
fst3 (x, _, _) = x

snd3 :: (a, b, c) -> b
snd3 (_, x, _) = x

printChart :: (Show m, Show c) => [ ( [c], m,[(Int,Double,Int)]) ] -> IO ()
printChart r =  do
                  writeFile "result.txt" ""
                  printChartAux r

printChartAux [] = return ()
printChartAux ((lc,mt,ys):xs) =
  do
    appendFile "result.txt" ( show mt ++ ";" ++ show lc ++ ";"  ++ show ys ++ "/")
    printChartAux xs

combs :: [c] -> [[c]]
combs l = (combsAux (length l) l)

combsAux :: Int -> [c] -> [[c]]
combsAux 0 _ = []
combsAux t x = (combsAux (t-1) x) ++ subsets t x

subsets 0 _ = [[]]
subsets _ [] = []
subsets n (x : xs) = map (x :) (subsets (n - 1) xs) ++ subsets n xs
```

Appendix B

Test repmin

```
main =  do putStrLn "Test repmin"
          --Test 1: All the combinations
          evalMemo False 3 c_test1 m_test1 (map (*10000) [1..10]) genByNode buildm repm
           --Test 2: Best combinations
          evalMemo True 3 c_test2 m_test2 (map (*100000) [1..10]) genByNode buildm repm
          --Test 3: Selective by node
          evalMemo False 4 c_test3 m_test3 (map (*100) [1..10]) genByNode buildm repmin
          --Test 4: Selective by att
          evalMemo True 3 c_test4 m_test4 (map (*10000) [1..10]) genByNode buildm repmi
          --Test 5: Memo vs no memo
          evalMemo False 2 c_test5 m_test5 (map (*100) [1..10]) genByNode buildm repmin

-- Test 1   ------------------------------------------------------
m_test1 =  reverse [((True,Nothing),(True,Nothing),(True,Nothing)),
             ((True,Nothing),(True,Nothing),(False,Nothing)),
             ((True,Nothing),(False,Nothing),(True,Nothing)),
             ((True,Nothing),(False,Nothing),(False,Nothing)),
             ((False,Nothing),(True,Nothing),(True,Nothing)),
             ((False,Nothing),(True,Nothing),(False,Nothing)),
             ((False,Nothing),(False,Nothing),(True,Nothing))]
c_test1 = [CForkm, CLeafm 0]
-----------------------------------------------------------------
```

```
-- Test 2  ---------------------------------------------------
m_test2 =  [((True,Nothing),(False,Nothing),(False,Nothing)),
            ((True,Nothing),(False,Nothing),(True,Nothing)),
            ((True,Nothing),(True,Nothing),(False,Nothing)),
            ((True,Nothing),(True,Nothing),(True,Nothing))]
c_test2 = [CForkm]
-----------------------------------------------------------------

-- Test 3  ---------------------------------------------------
m_test3 = [((True,Nothing),(True,Nothing),(True,Nothing))]
c_test3 = [CLeafm 0,CForkm]
-----------------------------------------------------------------

-- Test 4  ---------------------------------------------------
m_test4 =  reverse [((True,Nothing),(True,Nothing),(True,Nothing)),
                    ((True,Nothing),(True,Nothing),(False,Nothing)),
                    ((True,Nothing),(False,Nothing),(True,Nothing)),
                    ((True,Nothing),(False,Nothing),(False,Nothing)),
                    ((False,Nothing),(True,Nothing),(True,Nothing)),
                    ((False,Nothing),(True,Nothing),(False,Nothing)),
                    ((False,Nothing),(False,Nothing),(True,Nothing))]
c_test4 = [CForkm, CLeafm 0]
-----------------------------------------------------------------

-- Test 5  ---------------------------------------------------
m_test5 = [((True,Nothing),(False,Nothing),(False,Nothing))]
c_test5 = [CForkm]
-----------------------------------------------------------------

instance Arbitrary Tree where
    arbitrary = sized genByNode
```

```
genByDepth :: Int -> Gen Tree -- Generates a tree with n levels
genByDepth 0 = liftM Leaf arbitrary
genByDepth n =  liftM2 Fork t t
                  where t = genByDepth (n - 1)

genByNode :: Int -> Gen Tree -- Generates a tree with n nodes
genByNode 0 = liftM Leaf arbitrary
genByNode n =  do
                 x  <- choose (0,n-1) -- Random balance
                 t1 <- genByNode (n - x - 1)
                 t2 <- genByNode x
                 return $ Fork t1 t2


genConst :: Int -> Int -> Gen Tree -- Generate the same tree using n
genConst n _ =  genByDepth n
```

Appendix C

Klanguage

```
---------- KLanguage structure ----------

data Root = Root ListExp
          deriving (Show, Typeable, Data, Generic)

data ListExp = DotComma Expression ListExp
             | Return Variable
             | Exp Expression -- for if cases
          deriving (Show, Typeable, Data, Generic)

data Expression =
   Define Variable Type |
   Assign Variable GeneralExpression |
   LetIn Variable ListExp |
   If BoolExpression ListExp
   deriving (Show, Typeable, Data, Generic)

data GeneralExpression =
    BoolExp BoolExpression |
    StringExp StringExpression |
    AritmExp ArithmeticExpression
   deriving (Show, Typeable, Data, Generic)
```

```
data BoolExpression =
    And BoolExpression BoolExpression |
    Or BoolExpression BoolExpression |
    Not BoolExpression |
    BoolVar Variable |
    BoolConst Bool
   deriving (Show, Typeable, Data, Generic)

data StringExpression =
    Concat StringExpression StringExpression |
    StrVar Variable |
    StrConst String
   deriving (Show, Typeable, Data, Generic)

data ArithmeticExpression =
   Mult ArithmeticExpression ArithmeticExpression |
   Sum ArithmeticExpression ArithmeticExpression |
   AritmConst Int |
   AritmVar Variable
   deriving (Show, Typeable, Data, Generic)

data Type = TInt | TStr | TBool
   deriving (Show, Typeable, Data, Eq, Generic)
data Variable = Var String
   deriving (Show, Typeable, Data, Generic)

---------- Attributes ----------

compile :: AGRootm m [String]
compile = memo Compile $ \t -> case (constructor t) of
    CRoot      -> compile .@. ( t.$2 )
    CDotComma -> compileChilds t
    CExp       -> compile .@.  ( t.$2 )
```

```
CReturn   -> compile .@. ( t.$2 )
CDefine   -> case isDefined .@. ( t.$2 ) of
               (True,v) -> ([ getError DuplicatedDefinition ( t.$2 ) ],v)
               (False,v) -> ([],v)
CAssign   -> case compileChilds t of
               ([], v) -> let (t1, v0) = getType .@. ( v.$2 )
                              (t2, v00) = getType .@. ( v0.$3 )
                          in if ( t1 == t2 ) then
                               ([],v00)
                             else
                               ([ getError InvalidType ( t.$2 ) ],t)
               (ls, v) -> (ls,t)
CLetIn    -> compileChilds t
CIf       -> compileChilds t
CBoolExp  -> compile .@. ( t.$2 )
CStringExp -> compile .@. ( t.$2 )
CAritmExp -> compile .@. ( t.$2 )
CAnd      -> compileChilds t
COr       -> compileChilds t
CNot      -> compile .@. ( t.$2 )
CConcat   -> compileChilds t
CStrVar   -> checkDefAndType t TStrm
CBoolVar  -> checkDefAndType t TBoolm
CAritmVar -> checkDefAndType t TIntm
CVar      -> case isDefined t of
               (True, v) -> ([],v)
               (False,v) -> ([ getError NotDefined t ],v)
CAritmConst-> ([],t)
CBoolConst -> ([],t)
CStrConst -> ([],t)
CMult     -> compileChilds t
CSum      -> compileChilds t
```

```
compileChilds t =  let (l1, v) = compile .@. ( t.$2 )
                       (l2, v1) = compile .@. ( v.$3 )
                   in (l1 ++ l2,v1)

checkDefAndType v t =  case getType .@. ( v.$2 ) of
                         (Just t1, v0) ->  if ( t1 == t ) then
                                             ([], v0)
                                           else
                                             ([getError InvalidType ( v.$2 )], v)
                         (Nothing, _) -> ([getError NotDefined ( v.$2 )], v)

env :: AGRootm m VarsTable
env = memo Env $ \t ->  case (constructor t) of
                          CRoot -> ([],t)
                          otherwise ->   let (envP, v) = ( env  `atParent` t
)
                                         in if (t.|2) then -- Left sibling
                                           (envP, v)
                                         else -- Right sibling
                                           (envP ++ getDef ( left t ) , v)

isDefined :: AGRootm m Bool
isDefined = memo IsDefined $ \t ->  case ( getHole t :: Maybe (Variablem m) ) of
                                      Just( Varm m x ) ->  let (l,v) =
env t
                                                           in  case getVarFromEnv x l c
                                                                 Nothing
-> (False, v)
                                                               _
-> (True, v)


getType :: AGRootm m (Maybe Typem)
getType = memo GetType \$ \t ->  case ( constructor t ) of
```

```
CAnd -> (Just TBoolm, t)
COr -> (Just TBoolm, t)
CNot -> (Just TBoolm, t)
CTBool -> (Just TBoolm, t)
CBoolExp -> (Just TBoolm, t)
CBoolConst -> (Just TBoolm, t)
CBoolVar -> (Just TBoolm, t)
CConcat -> (Just TStrm, t)
CTStr -> (Just TStrm, t)
CStrConst -> (Just TStrm, t)
CStringExp -> (Just TStrm, t)
CStrVar -> (Just TStrm, t)
CAritmExp -> (Just TIntm, t)
CMult -> (Just TIntm, t)
CSum -> (Just TIntm, t)
CAritmConst -> (Just TIntm, t)
CAritmVar -> (Just TIntm, t)
CTInt -> (Just TIntm, t)
CVar -> let (l,v) = env t
        in case getVarFromEnv ( getName t ) l of
             Just(varType) -> (Just varType, v)
             Nothing -> (Nothing, v)
```