Editor Genérico de Estructuras

Nancy Tatiana Aparicio Yuja

Departamento de Ingeniería de Sistemas
Universidad Católica Boliviana, Regional Cochabamba
e-mail: aparicio@ucbcba.edu.bo

Resumen

En el presente trabajo se explotan las ventajas de la programación funcional, para construir un editor, que permite la manipulación directa de estructuras libres de contexto con una interfaz gráfica. Como resultado de este trabajo se obtiene un modelo genérico (en el sentido que es muy fácil adaptarlo a estructuras particulares) y al mismo tiempo sencillo (se identifican claramente un *modelo de datos* en que cada componente de la estructura guarda "su" estado y un *modelo de programa* en que las operaciones esenciales de edición están definidas por álgebras que son pasadas a una función fold que trabaja sobre la estructura). El trabajo apunta a la obtención de un programa generador de editores que reciba una gramática libre de contexto y retorne un editor para el lenguaje que obedece a tal gramática.

1 Introducción

Cada vez que se diseña un nuevo lenguaje, surge la necesidad de desarrollar la aplicación que permita editar objetos que obedezcan a su estructura; es por ello que resultaría de gran utilidad para los diseñadores de lenguajes, con tan solo la especificación sintáctica del nuevo lenguaje mediante una gramática libre de contexto, generar su editor sin ningún costo. Apuntando a tal aplicación, en el presente artículo se presenta un editor de estructuras de árboles rosa[1] que puede ser fácilmente adaptado a estructuras particulares.

La propuesta se implementa en el lenguaje funcional Haskell porque el paradigma de la programación funcional, con sus poderosas herramientas de abstracción de datos y programas, es ideal para la obtención de aplicaciones generales, lo que es el primer paso para la posterior construcción de aplicaciones genéricas. Algunos argumentos que corroboran lo anterior son los siguientes:

[1] Arboles con cualquier cantidad de descendientes por nodo.

Figura 1: El objeto a editarse tiene tres representaciones: el objeto estructurado, la estructura árbol que lo modela y la presentación.

- La especificación de una estructura mediante una gramática libre de contexto tiene su correspondencia directa con un tipo de datos del lenguaje funcional Haskell. Una producción de la gramática $p : X \to Y$ puede ser interpretada funcionalmente como una definición del tipo X o una función constructora g con tipo $g :: Y \to X$.
- Libera al programador de administrar memoria, tarea que distrae su atención en aspectos no esenciales a la solución del problema [3, 9].
- Las estructuras de datos y las funciones son manejadas como valores de primer orden, lo que da mayor libertad para manipular estos elementos de programación. Esta característica permite que una función pueda ser argumento o resultado de otra, lo cual es una herramienta útil para la obtención de funciones generales [3, 9].
- La programación funcional permite evaluación "lazy" para reducir expresiones [3], en virtud a esta característica es posible definir funciones con expresiones que usan valores calculados por las propias expresiones, lo que nos da mayor poder de expresividad al especificar modelos.

2 Especificación del editor

En un editor se pueden identificar dos componentes principales: la estructura a editarse y las operaciones para manipularla.

2.1 La estructura

Se desea obtener un editor que trabaje sobre objetos que obedecen a gramáticas libres de contexto. Todo objeto que obedece a una gramática libre de contexto puede ser modelado como árbol (detrás de una frase que obedece al lenguaje, existe un árbol de derivación). Por tanto, la estructura que está siendo editada, además de tener una representación para ser visualizada en pantalla, puede tener una representación en una estructura general: un árbol. En este contexto, podemos identificar tres tipos de representaciones para una estructura libre de contexto a editarse: la estructura, la estructura modelada como árbol y su presentación en pantalla, como puede observarse en la Figura 1.

En el presente trabajo nos ocuparemos de la *estructura árbol* porque al obtener un editor para un árbol se obtiene automáticamente un editor para cualquier objeto que pueda ser modelado como tal.

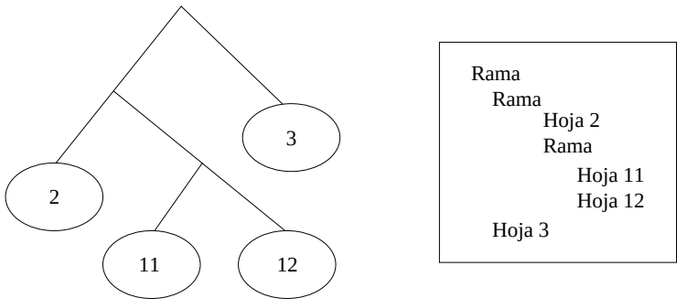

Figura 2: Un árbol y su presentación en pantalla.

Aunque es un aspecto particular de la estructura que se edita, se trabajará en la *presentación* para experimentar con mecanismos de interacción en programación funcional[2] y obtener un producto que permita hacer una manipulación directa del objeto, utilizando el mouse.

2.1.1 Especificación de la estructura árbol

Se construirá un editor para un árbol rosa que cumpla las siguientes características:

1. Cada nodo tiene n descendientes que pueden ser direccionados individualmente (n puede variar de nodo a nodo).
2. Solo se guarda información en nodos terminales, los no terminales guardan la estructura.
3. Todos los datos guardados son del mismo tipo.

2.1.2 Especificación de la presentación

Se mostrará el árbol como texto, desplegando cada elemento en una fila diferente y utilizando indentaciones para representar la profundidad del elemento (ver Figura 2).

2.2 Las operaciones

Se implementarán operaciones de navegación, manipulación y despliegue.

2.2.1 Operaciones de navegación/selección

Se hará la navegación mediante el mouse, así el usuario tendrá libertad y facilidad para mover el foco de selección en la estructura. Se considerarán las siguientes características

[2]La implementación de aplicaciones interactivas en programación funcional se complica por el manejo de estados que implica.

de enfoque:

1. Una selección siempre incluirá hojas (la selección de un nodo no terminal, implica la selección de toda su descendencia).
2. Una selección siempre abarcará un subárbol cada vez.

2.2.2 Operaciones de manipulación/transformación

El área seleccionada de la estructura puede ser modificada en contenido (modificar el contenido de una hoja) o en estructura (transformar una hoja en rama, transformar una rama en hoja).

2.2.3 Operaciones de despliegue

Los comandos de despliegue permitirán llevar a cabo las anteriores operaciones (selección y transformación) mediante la manipulación directa de la estructura por el usuario, por lo que las operaciones que se considerarán, entre otras, son las siguientes:

1. Calcular posiciones de despliegue en pantalla, de cada uno de los componentes del árbol.
2. Desplegar, en posiciones exactas de la pantalla y con un color específico, cada nodo del árbol.
3. Borrar el árbol para poder desplegar las transformaciones realizadas por el usuario.

3 Modelo de la solución

3.1 Modelo de datos

Para implementar las operaciones indicadas, se requiere mantener la siguiente información de estado: posiciones de despliegue de cada componente del árbol, ubicación de cada componente dentro de la estructura, componentes del árbol seleccionados y valor que guarda cada componente.

Lo más sencillo es guardar esta información en el propio árbol, cada nodo conocerá "su" estado: posición de despliegue, ubicación en la estructura, si está seleccionado o no y el valor que guarda. Para ello se propone el modelo de datos presentado en el Algoritmo 1 e ilustrado en la Figura 3.

3.2 Modelo de programa

Para implementar las operaciones del editor se trabajará composicionalmente [4]. Trabajar composicionalmente consiste en tomar ventaja de la estructura y hacer que cada

Algoritmo 1 : Modelo de Datos que corresponde a una definición de tipo en Haskell. Cada nodo del árbol guardará la tupla (posición, valor, marca), que corresponde a (posición en la ventana donde se desplegará el elemento, valor que guarda, valor lógico que indica seleccionado / no seleccionado).

```
data Arbol a = Nodo a [Arbol a]
type Estado = (Posicion, String, Marca)
type Posicion = (Int,Int)
```

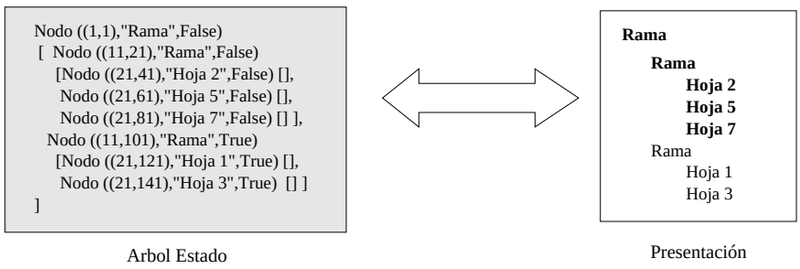

Figura 3: Un ejemplo de un árbol representado en el modelo de datos y su correspondiente presentación en pantalla. Obsérvese que no es necesario guardar la ubicación de cada componente en la estructura porque está implícita en el árbol.

uno de los componentes de la misma tenga una tarea asignada, de modo que se haga una composición del trabajo. Para implementar esto, será necesario realizar lo siguiente:

1. Identificar los puntos estructurales del objeto, que se corresponden directamente con los constructores del tipo de datos definido. En el caso específico del modelo de datos usado, se tienen tres puntos estructurales: $Nodo$, (:), [] (en base a estos elementos se puede construir un árbol).

2. Reemplazar cada punto estructural por una función de acuerdo a la tarea que se desee realizar (las funciones que reemplazarán a los constructores se disponen en una tupla denominada álgebra). Cada función que reemplaza a cada constructor tendrá como argumentos, además de la información de estado, los resultados de lo calculado por su padre (atributo heredado) y de lo calculado por sus hijos (atributos sintetizados). En virtud a la evaluación "lazy", será posible que un mismo atributo sea heredado y sintetizado al mismo tiempo (ver atributo fila en la definición de el Algoritmo 3).

Esta idea aplicada al árbol rosa está esquematizada en la Figura 4 y codificada mediante la función $fold\_Tree$ (Algoritmo 2). Para implementar cada una de las operaciones de edición se hará uso de esta función escogiendo el álgebra adecuada (*nodo*, *cons*, *vacía*) que reemplazará a los constructores $Nodo$, (:), [] respectivamente.

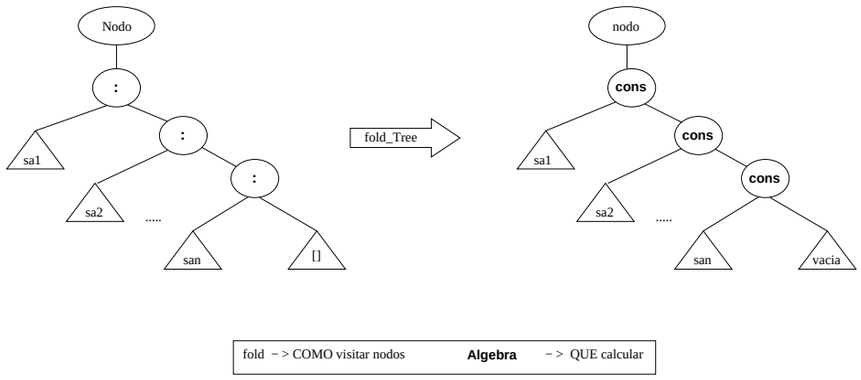

Figura 4: *fold_Tree* reemplaza los constructores *Nodo*, (:) y [] (árbol izquierdo) por las funciones *nodo*, *cons* y *vacia* (árbol derecho) respectivamente.

Algoritmo 2: La función *fold_Tree* reemplaza los constructores (*Nodo*, (:), []) por las funciones del álgebra (*nodo*, *cons*, *vacía*). La función nodo indica qué hacer al visitar un nodo, cons indica qué hacer al visitar cada hijo en la lista de descendientes del nodo, *vacía* indica qué hacer cuando la lista de descendientes se termina.

```
fold_Tree (nodo,cons,vacia) = fold
   where fold (Nodo x xs)=nodo x (foldLista cons vacia (map fold xs)))
foldLista f a [ ] = a
foldLista f a (x:xs)= f x (foldLista f a xs)
```

En las siguientes secciones se define cada una de las operaciones de edición utilizando el modelo de programa presentado. Se hace una explicación minuciosa únicamente de una de las operaciones (cálculo de las posiciones de despliegue), para detalles del resto consultar la referencia [1].

3.3 Operaciones de despliegue

3.3.1 Cálculo de las posiciones de despliegue para la presentación

Para calcular la posición de despliegue de cada componente, se deben resolver los siguientes problemas:

- El cálculo de la fila y columna depende del espacio que ocupa cada carácter que se despliega en pantalla, lo que a su vez depende del tipo de letra usado.
- El cálculo de la columna donde se debe desplegar un elemento, estará en función al nivel de profundidad que éste tiene en el árbol y el cálculo de su fila, en función a la regla: "desplegar los subárboles en el mismo orden en que aparecen en la

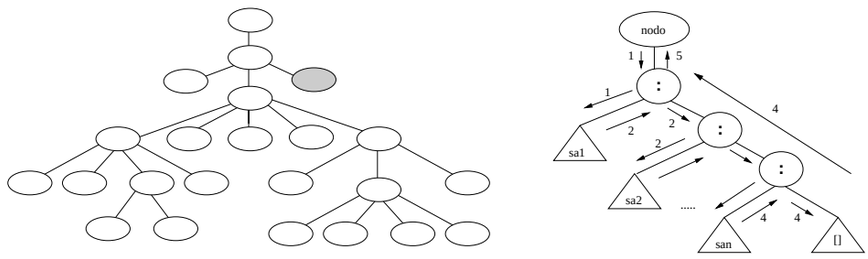

Figura 5: Observando la gráfica izquierda, se puede apreciar que calcular la posición del nodo sombreado no es fácil, sólo se podrá saber su fila una vez conocidas las filas de todos los demás. La gráfica de la derecha describe el modo de calcular las posiciones de despliegue de cada nodo del árbol rosa. El subárbol sa1 hereda la fila de su padre Nodo (flecha 1), el subárbol sa1 sintetiza su última fila y la pasa a su hermano (flecha 2), el subárbol sa2 hereda la fila de su hermano sa1, el último subárbol san pasa la fila a su padre (flecha 4), para que éste a su vez la pase a su padre.

lista". Notar que sólo se podrá saber la fila del subárbol (n+1), una vez que se conozca la fila del subárbol n (Figura 5 izquierda).

El primer problema será resuelto mediante el uso de un único tipo de letra, para saber exactamente el alto y ancho de cada carácter (información que estará en h_offset y v_offset respectivamente). Para generalizar a cualquier tipo de letra, bastará abstraer el cálculo de estos dos valores en una función.

El segundo problema, se resuelve fácilmente en virtud a la evaluación "lazy" y al modelo de programa utilizado. Se calcularán las posiciones de despliegue como se ilustra en la Figura 5 (derecha).

Por tanto, el árbol original que desea editarse será mapeado a un árbol con información de estado y será sobre éste que se trabajará durante la interacción con el usuario.

La idea anteriormente explicada se plasmó en la función *arbolInformado*, que se implementa en el Algoritmo 3.

3.3.2 Álgebra de la función arbolInformado: (info_nodo, info_cons, info_vacia)

info_nodo e es c f : Recibe la información de estado del nodo (e), la lista de descendientes del nodo (es), la columna (c) y fila (f) donde debe desplegarse el nodo. Retorna una tupla (f,arb), donde:

f=siguiente fila a la última que ocupó el árbol cuya raíz es el nodo en cuestión.

arb=árbol que obedeciendo a la estructura del árbol original, guarda en sus nodos la información de estado.

Algoritmo 3 : Código Haskell de la función que calcula la información de estado del árbol. Recibe un árbol a, una columna c y fila f, que es el punto en pantalla donde empieza a desplegarse el árbol. Calcula un nuevo árbol que guarda en sus nodos la tripla $(pos, cad, False)$ que corresponde a la posición de depliegue, cadena a desplegarse para representar el nodo y no seleccionado respectivamente. Notar que en la función $info\_nodo$, el valor fi que es usado en el resultado es calculado por la propia función, esto es posible en virtud a la evaluación "lazy". Notar en las líneas 4, 8 y 12 que e y es están en la propia estructura y c, f son atributos heredados.

```
 1) arbolInformado::Arbol Estado -> Int -> Int -> Arbol Estado
 2) algebraArbolInformado=(info_nodo,info_cons,vacia)
 3) arbolInformado a c f=snd(fold_Tree algebraArbolInformado a c f)
 4) info_nodo e es c f
 5)     = let (fi,xs) = es (c+h_offset)(f+v_offset)
 6)     cad=if length xs==0 then "Hoja"++show e else "Nodo"
 7)     in (fi,Nodo ((c,f),cad,False) xs)
 8) info_cons e es c f
 9)     = let (fi,x) =e c f
10)    (fi2,xs)=es c fi
11)     in (fi2,x:xs)
12) vacia c f =(f,[ ])
```

Analizando el código del Algoritmo 3, en la línea 5 (expresión derecha) info_nodo pasa a sus descendientes (es) la fila y columna donde deben empezar su despliegue, por tanto éstos son atributos heredados y en la línea 5 (expresión izquierda) info_nodo recibe de sus descendientes (es) la última fila (fi) que ocuparon, por tanto fila es un atributo sintetizado.

info_cons e es c f : Recibe el primer árbol e y el resto de los árboles (es) de la lista de descendientes que acompaña a Nodo, la columna (c) y fila (f); retorna una tupla (f,lista), donde:

f=fila donde se despliega el último elemento de la lista de árboles (lista).

lista=lista de árboles descendientes con su respectiva información de estado.

Analizando el código del Algoritmo 3, en la línea 9 (expresión derecha) info_cons pasa a su primer descendiente (e) la columna y fila donde debe empezar a desplegarse, por tanto c y f son atributos heredados. En la línea 9 (expresión izquierda) info_cons recibe de su primer descendiente (e) la siguiente fila a la última que ocupa e (fi), por tanto fi es un atributo sintetizado. En la línea 10 (expresión derecha) info_cons pasa al resto de sus descendientes (es) la columna (que recibió de su padre) y la fila (fi)que recibió de su anterior descendiente (e). En la línea 10 (expresión izquierda), info_cons recibe del resto de sus descendiente (es) la siguiente fila a la última que ocuparon (fi2), por tanto fi2 es un atributo sintetizado.

vacia c f: Recibe una columna (c) y fila (f) que es heredada de su padre y retorna una tupla (f,[ ]) para terminar de construir la lista de descendientes (linea 5 : xs) y retornar la última fila que ocupó la lista de árboles descendientes de Nodo.

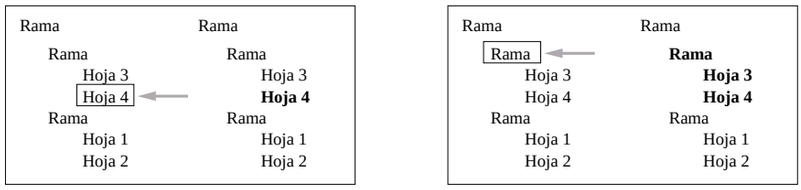

Figura 6: Al seleccionar una hoja, ésta se muestra de diferente color, al seleccionar un nodo no terminal, éste y toda su descendencia se muestran de diferente color.

Algoritmo 4: Despliega el árbol a en la ventana w, mostrando con color c lo seleccionado (marca=True) y con color neutro, lo no seleccionado (marca = false). Las funciones *escribe*2, *rojo*, *neutro* se pueden encontrar en la referencia [1].

```
neutro=rojo
muestraArbol::Arbol Estado ->Window -> RGB -> IO()
algebraMuestraArbol=(muestraNodo,muestraCons,muestraVacia)
muestraArbol a w c=fold_Tree algebraMuestraArbol a w c
muestraNodo (pto,v,m) xs w color
   =(if m then escribe color else escribe neutro)>>(xs w color)
   where escribe=ctext2 w pto v
muestraCons e es w color = (e w color) >> (es w color)
muestraVacia w color=return()
```

3.3.3 Despliegue del árbol y sus componentes seleccionados

Cada nodo se despliega de acuerdo a su información de estado; se muestran las secciones seleccionadas de un color y las no seleccionadas de otro, mediante la función muestraArbol (Algoritmo 4):

Para borrar el árbol de la pantalla, bastará con desplegarlo utilizando el color de fondo de la pantalla, utilizando la funcion muestraArbol (Algoritmo 4).

3.4 Operaciones de navegación/selección

3.4.1 Aspectos a considerar

Para lograr que el usuario pueda seleccionar el elemento del árbol que desea editar, se determinarán áreas sensibles a la presión del mouse y colores que indiquen las zonas seleccionadas. Si el usuario presiona el mouse sobre un nodo, éste con toda su descendencia se marcará de otro color (ver Figura 6):

Para permitir que el usuario pueda seleccionar áreas de la estructura:

- Se debe detectar el punto de la pantalla donde el usuario presionó el mouse así como verificar si éste está en un componente del árbol.

- En función a esta selección, desplegar el árbol de otro color para que la acción del usuario tenga eco en la interfaz usada para la edición.
- En caso de que el componente seleccionado sea un nodo no terminal, se debe asumir que el usuario desea seleccionarlo junto con toda su descendencia.

Para saber si el mouse tocó un componente del árbol, se necesita conocer el área que éste ocupa en pantalla. El cálculo del área estará en función al tipo de letra usado y a la cantidad de caracteres que se estén desplegando para representar el nodo. De este modo, si anchoArea es la cantidad de pixeles que ocupa la cadena que se despliega para representar el componente y altoArea es la cantidad de pixeles que tiene el alto de un carácter, entonces se sabría que el mouse (colMouse,filaMouse) tocó el componente cuya esquina superior izquierda es (col,fila) cuando se cumplan simultáneamente las siguientes condiciones:

ColMouse >= col
ColMouse <= col+anchoArea
FilaMouse >= fila
FilaMouse <= fila+altoArea

Para resolver el problema de que la selección de un nodo no terminal debe implicar la selección de toda su descendencia, se adoptará la siguiente idea:

- Si un nodo fue tocado por el mouse, el valor de su marca toma el valor True.
- Si un nodo no fue tocado por el mouse, hereda el valor de marca de su padre.

En el Algoritmo 5 se presenta la función que abstrae lo explicado[3]:

Algoritmo 5: La función *marcaArbol* recibe un arbol (a), el punto donde se presionó el mouse (*mouse*) y el tipo de operación deseada (*mark*). Marca (si mark es True) o desmarca (si mark es False) el nodo del árbol a que fue tocado por el mouse y todos los elementos descendientes de él. La función devolverá un árbol con su información de estado modificada de acuerdo a la selección.

```
marcaArbol::Arbol Estado -> Bool -> Pos -> Bool -> Arbol Estado
algebraMarcaArbol =(marcaNodo,marcaCons,marcaVacia)
marcaArbol a mPadre mouse mark=fold_Tree algebraMarcaArbol mPadre mouse mark
marcaNodo (pto,v,m) es marcaPadre ptoMouse mark
   =Nodo (pto,v,nm) (es nm ptoMouse mark)
   where nm=if (toca ptoMouse pto) then mark else marcaPadre
marcaCons x xs marcaPadre ptoMouse mark
   = (x marcaPadre ptoMouse mark):(xs marcaPadre ptoMouse mark)
marcaVacia marcaPadre ptoMouse mark=[ ]
```

[3]Cabe destacar que esta función podrá ser usada para dos efectos diferentes: marcar y desmarcar.

3.5 Operaciones de manipulación/transformación

En general, cada vez que se realiza una modificación en el árbol, se debe hacer una actualización del mismo, de modo que se pueda continuar la interacción con la nueva versión.

Para transformar el árbol, es requisito que el componente sobre el que se realizará la transformación, esté marcado. Por tanto, cada una de las operaciones de transformación que se describen a continuación tienen efecto sobre el componente "seleccionado" de la estructura.

3.5.1 Modificar el contenido de una hoja

Requiere buscar la hoja elegida, cambiar su contenido y actualizar el despliegue (Algoritmo 6).

Algoritmo 6: Cambia el valor que guarda la hoja seleccionada de a por el valor nv. Si el nodo está marcado (m==True) y es una hoja, cambia su contenido por nv, en caso contrario el contenido del nodo n no es modificado. Nótese que nv es un parámetro heredado.

```
cambiaHoja::Arbol Estado -> Valor -> Arbol Estado
algebraCambiaHoja=(cHojaNodo,cHojaCons,cHojaVacia)
cambiaHoja a nv= fold_Tree algebraCambiaHoja a
cHojaNodo n@(pto,v,m) xs nv
   =if m && length (xs nv)==0 then Nodo (pto,"Hoja"++nv,False) (xs nv)
         else Nodo n (xs nv)
cHojaCons e es nv=(e nv):(es nv)
cHojaVacia nv=[ ]
```

3.5.2 Transformar una hoja en rama

La transformación de una hoja en rama requiere, al igual que en el anterior caso, identificar la hoja seleccionada, borrar la versión anterior del árbol y redesplegar la nueva, ya que las posiciones de despliegue de los elementos ubicados en filas posteriores al afectado bajan tantos niveles como componentes tenga la nueva rama (ver Algoritmo 7).

3.5.3 Transformar una rama en una hoja

Las posiciones de despliegue de los elementos ubicados en filas posteriores al elemento afectado deben subir tantos niveles como filas haya ocupado la rama transformada (ver Algoritmo 8).

En los tres casos de transformación, el árbol devuelto queda sin marcas.

Algoritmo 7: Si el nodo está marcado (m=True), es vulnerable a la operación, por lo que es transformado en no terminal con nh hojas que guardan el valor 0. Para los nodos no marcados (m=False), si la posición de despliegue (c, f) está en una fila posterior a la posición del nodo modificado (cm, fm), se cambia el valor de su posición de despliegue bajando su fila nh posiciones para dar espacio a los nuevos elementos que se han creado.

```
1) h2rArbol::Arbol Estado -> Int -> Pos -> Arbol Estado
2) algebra_h2r=(h2rNodo,h2rCons,h2rVacia)
3) h2rArbol = fold_Tree algebra_h2r
4) h2rNodo e@((c,f),v,m) es nh (cm,fm)
5)     =if m then Nodo ((c,f),"Nodo",False)(foldr g [ ] [1..nh])
6)     else Nodo((c,nf),v,m) (es nh (cm,fm))
7)     where
8)       nf=if f>fm then (f+nh*v_offset) else f
9)       g x xs=(Nodo((c+h_offset,f+x*v_offset),"Hoja 0",False)[ ]):xs
10) h2rCons t ts nh ptoMouse=(t nh ptoMouse):(ts nh ptoMouse)
11) h2rVacia nh ptoMouse=[ ]
```

Lo interesante de la solución planteada es que para las tres operaciones de transformación, la identificación de la selección (sobre la que se debe aplicar la transformación) es automática, es decir, no se usa una función especial para identificar el área seleccionada, se aplica la función de transformación a todo el árbol y únicamente los nodos seleccionados se ven afectados[4].

4 Hacia el editor genérico

Una buena táctica para desarrollar aplicaciones generales es seguir el ciclo abstracción-especialización; en un proceso de *abstracción* en el que se eliminan detalles irrelevantes, se obtiene un producto de propósito general (un modelo esencial); en el proceso de *especialización* se instancia el producto general para satisfacer requerimientos particulares [2]. Lo atractivo de esta forma de trabajo es poder reusar estos productos generales que al tener la condición de "esenciales" pueden ser usados como patrones de programas.

Es en búsqueda de este producto de propósito general que la presente sección tiene su justificación. En efecto, se presenta un escenario para lograr un generador de editores de estructuras que pueda editar un árbol de cualquier tipo. En un proceso de especialización, se toma la aplicación de propósito general (la abstracción) y se la modifica para que se convierta en una aplicación de propósito específico (la especialización). Las modificaciones que se deban hacer sobre la aplicación general para obtener el producto especializado es un indicador de cuan buena es la abstracción usada y puede servir como un instrumento para llevar a cabo un proceso de refinamiento de este modelo esencial. En esta sección se lleva a cabo la especialización del editor rosa a editores particulares

[4] Otras implementaciones similares usan un modelo que obliga a que la identificación de la selección requiera de una función especial, ver referencias [8, 7, 5].

Algoritmo 8: Si la marca del nodo (m) es True, éste debe ser transformado en hoja, retornando Nodo ((c,f),"Hoja 0",False). Si la posición de despliegue del nodo en cuestión (c,f) está en una fila de la pantalla posterior a la fila de la posición de la rama que se está transformando (cm,fm), entonces se recalcula (c,f), disminuyéndole tantas filas como descendientes tenga el nodo (niv).

```
r2hArbol:: Arbol Estado -> posición -> Int -> Arbol Estado
algebra_r2h=(r2hNodo,r2hCons,r2hVacia)
r2hArbol a (cm,fm) niv= fold_Tree algebra_r2h a (cm,fm) niv
r2hNodo ((c,f),v,m) es (cm,fm) niv
   =if m then Nodo ((c,f),+"Hoja 0",False) [ ] else arbol
      where nf=if f>fm then (f-niv*v_offset) else f
         arbol=Nodo ((c,nf),v,m) (es (cm,fm) niv)
r2hCons t ts ptoMouse niv=(t ptoMouse niv):(ts ptoMouse niv)
r2hVacia ptoMouse niv = [ ]
```

y siguiendo el ciclo abstracción-especialización, se muestra un proceso de refinamiento hasta la obtención de una abstracción razonable. La idea general que se seguirá para llevar a cabo la especialización es la aplicación de "mapeamientos" de valores del tipo x al tipo rosa y viceversa: la idea es convertir cualquier entrada que obedezca al tipo x al tipo rosa, manipular la estructura con el editor rosa y finalmente convertir la estructura rosa al tipo x. Lo indicado se ilustra en la Figura 7:

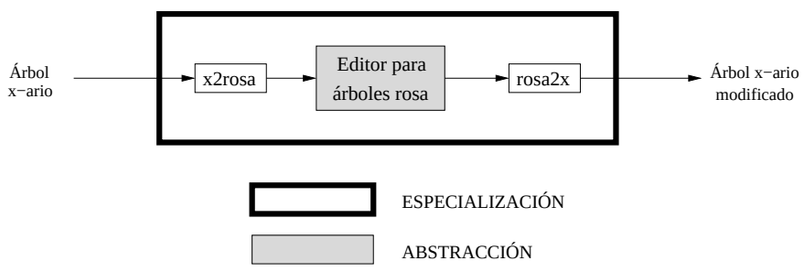

Figura 7: Modelo Abstracción-Especialización aplicado al editor de estructuras.

Siguiendo esta idea, se presentan a continuación algunos ejemplos de especializaciones tomando como abstracción el editor presentado en la sección precedente.

4.1 Especialización cambiando el código fuente

4.1.1 Abstracción

Tomando como abstracción el editor cuyas funciones esenciales se presentan en la sección 3 y que tiene como función principal de interacción la siguiente:

Algoritmo 9 : Código Haskell de la función principal del editor de árboles rosa que permite la interacción y hace referencia a las funciones de edición esenciales explicadas en la sección 3. *La función completa puede ser encontrada en la referencia [1].*

```
1) edita a
2)     = let loop1 w ai
...................
4)                Closed -> return ai
...................
50)             ejec (h2rArbol olda (car2num car) pt) loop1 w olda
57)                let ai=arbolInformado a 1 1
58)                w <- openWindow "Edicion de Arboles n-arios"(620,400)
59)                muestraArbol ai w azul
60)                menu w listaOpciones agua (colMenu,filaMenu)
...................
```

4.1.2 Especialización a editor de árboles binarios

Para especializar a un árbol binario[5] se debería convertir el árbol binario a árbol rosa, manipular la estructura como árbol rosa mediante la abstracción, controlar que durante la edición el árbol cumpla la condición de ser binario (que las longitudes de las listas sean 2 o 0) y finalmente convertir el árbol rosa en árbol binario.

Para implementar lo indicado, se deberían definir las funciones de conversión y hacer algunos cambios en el código de la función edita (Algoritmo 9).

Definición de las funciones

bin2rosa (Hoja x)= Nodo x [ ]
bin2rosa (Rama e i d)= Nodo e [bin2rosa i, bin2rosa d]
rosa2bin (Nodo x [ ]) = Hoja x
rosa2bin (Nodo x i:d:[ ]) = Rama x (rosa2bin i) (rosa2bin d)

Cambio en el código El código de la función edita (Algoritmo 9) debería cambiar como se ilustra en el Algoritmo 10:

Lo malo de la abstracción usada es que el hacer especialización implica cambiar el código, lo cual es un indicador de que ésta podría ser refinada.

4.2 Especialización mediante parametrización

Si se colocan como parámetros de la función edita las *funciones de transformación* y el *grado del árbol* entonces una especialización solo requeriría llamar a la función edita con

[5]Definición de tipo de un árbol binario: data B a = Bhoja a | Brama (B a)(B a).

Algoritmo 10: Cambian las líneas 4, 50 y 57, se borran las líneas 45 a 49. Notar que las partes esenciales del programa que se explican en la sección 3 no se ven afectadas.

```
1) edita a
2)     = let loop1 w ai
.....................
4)                Closed -> return (rosa2bin ai)
.....................
50)             ejec (h2rArbol olda 2 pt) loop1 w olda
57)                let ai=arbolInformado (bin2rosa a) 1 1
.....................
```

los parámetros adecuados y no sería necesario cambiar el código. Para ello se tendría que cambiar la función edita (Algoritmo 10) como se muestra en el Algoritmo 11.

4.2.1 Abstracción

Algoritmo 11: Código Haskell de la abstracción parametrizada. Las funciones de conversión de estructuras (*x2rosa* **y** *rosa2x*) así como el grado (*nroHijos*) son parámetros de la función (línea 1).

```
1) edita x2rosa rosa2x nroHijos a
2) = let loop1 w ai
.......
4)        Closed -> return (rosa2x ai)
.......
50)        ejec (h2rArbol olda nroHijos pt) loop1 w olda
56) in do
57)        let ai=arbolInformado (x2rosa a) 1 1
.........
```

En estas condiciones, el editor puede ser utilizado para manejar cualquier árbol, siempre y cuando se den como parámetros las funciones de conversión, de acuerdo al tipo de árbol a editarse.

4.2.2 Especialización a un editor de árboles de grado n

rg (edita **bin2rosa rosa2bin 2** arbolTipoB) (Especialización a un árbol binario)
rg (edita **ter2rosa rosa2ter 3** arbolTipoT) (Especialización a un árbol ternario)

La abstracción ahora obtenida es mejor en el sentido que se obtiene una especialización tan solo pasando los parámetros adecuados a la expresión que usa el editor general; sin embargo, las funciones que convierten de un tipo de estructura a la otra tienen diferentes nombres cuando en esencia tienen el mismo rol.

4.3 Especialización mediante clases (primera versión)

Se podría crear una clase para sobrecargar las funciones de transformación x2rosa y rosa2x, en cuyo caso éstas ya no tendrían que ser parámetros de edita. El editor automáticamente reconocería la estructura y trabajaría con la instancia x2rosa, rosa2x correspondiente.

4.3.1 Abstracción

Se añade la clase del Algoritmo 12 y se borran los parámetros (x2rosa, rosa2x y nroHijos) de la línea 1 del código del Algoritmo 11.

Algoritmo 12 : La clase *ToFromRosa* agrupará todas las estructuras que el editor reconocerá.

```
class ToFromRosa a b where
x2rosa::b->RosaTree a
rosa2x::  RosaTree a -> b
nroHijos::Int
```

4.3.2 Especialización a un árbol binario

```
instance ToFromRosa a (B a) where
    x2rosa (Bhoja y)= Nodo y [ ]
    x2rosa (Brama y i d)= Nodo y [x2rosa i, x2rosa d]
    rosa2x (Nodo y [ ])= Bhoja y
    rosa2x (Nodo y [i,d])=Brama y (rosa2x i)( rosa2x d)
    nroHijos=2
```

4.3.3 Especialización a un árbol ternario

```
instance ToFromRosa a (T a) where
    x2rosa (Thoja y)= Nodo y [ ]
    x2rosa (Trama i m d)= Nodo y [x2rosa i, x2rosa m, x2rosa d]
    rosa2x (Nodo y [ ])= Thoja y
    rosa2x (Nodo y [i,m,d])=Trama y (rosa2x i) (rosa2x m)(rosa2x d)
    nroHijos=3
```

Notar que, en esta oportunidad la especialización se vuelve transparente al usuario. Bastará con instanciar en la clase ToFromRosa el nuevo tipo de datos, lo que básicamente significa definir las funciones de conversión y el grado del árbol. Ahora se

podrá utilizar la función edita con cualquier estructura cuyo tipo de datos pertenezca a la clase ToFromRosa:

rg (edita **(Brama (Bhoja2) (Bhoja 3))**) [6]

rg (edita **(Trama (Thoja2) (Thoja 3) (Thoja 4))**) [7]

Lo negativo de esta solución es que sólo permite manejar árboles con grado constante.

4.4 Especialización mediante clases (segunda versión)

Se podría generalizar la abstracción de modo que permita manejar árboles con grado menor a una constante, con lo cual se podrían hacer especializaciones a una gran gama de editores.

4.4.1 Abstracción

Añadir a la clase ToFromRosa (Algoritmo 12) la función *compara* :: $Int \rightarrow Int \rightarrow Bool$.

Esta función daría la libertad de manejar árboles de grado constante (==) o árboles de grado menor o igual a una constante (<=).

4.4.2 Especialización a un árbol ternario

```
instance ToFromRosa a (T a) where
    x2rosa (Thoja y)= Nodo y [ ]
    x2rosa (Trama i m d)= Nodo " "[x2rosa i,x2rosa m,x2rosa d]
    rosa2x (Nodo y [ ])= Thoja y
    rosa2x (Nodo y [i,m,d])=Trama y (rosa2x i)(rosa2x m) (rosa2x d )
    nroHijos=3
    compara=(==)
```

4.4.3 Especialización a un árbol bin-ternario (del tipo BT)

Definición:
data BT a=BThoja a | BTrama2 a (BT a)(BT a) | BTrama3 a (BT a)(BT a)(BT a).

Nótese que:

- Añadiendo a la clase ToFromRosa la función compara, se da mayor generalidad a la abstracción. La idea que subyace a esta función es permitir especializar árboles de grado constante o menor o igual a una constante.

[6] Definición de tipo de un árbol binario: data B a = Bhoja a | Brama (B a)(B a).
[7] Definición de tipo de un árbol ternario: data T a = Thoja a | Trama (T a)(T a)(T a).

```
instance ToFromRosa a (BT a) where
    x2rosa (BThoja y)= Nodo y [ ]
    x2rosa (BTrama2 y i d)= Nodo y [x2rosa i, x2rosa d]
    x2rosa (BTrama3 y i m d)= Nodo y [x2rosa i, x2rosa m, x2rosa d]
    rosa2x (Nodo y [ ])= BThoja y
    rosa2x (Nodo y [i,d])=BTrama2 y (rosa2x i)( rosa2x d)
    rosa2x (Nodo y [i,m,d])=BTrama3 y (rosa2x i) (rosa2x m)(rosa2x d)
    nroHijos=3
    compara=(<=)
```

- Aún existen árboles que no podrían ser especializados mediante la instanciación, es el caso de aquellos que tuvieran por ejemplo nodos con 5 hijos y nodos con 3 hijos. Sin embargo, esto también podría ser generalizado añadiendo a la clase una lista que guardaría los grados que se aceptan en el árbol. En tal caso la función "compara" ya no sería necesaria.

- Se ha llegado a un punto en que la especialización no requiere modificación alguna al código, en efecto, especializar se reduce a definir las funciones de transformación y el grado (como instancias de la clase ToFromRosa).

5 Conclusiones

1. El editor de estructuras implementado es razonablemente general, de hecho trabaja con muchos tipos de árboles. Si se quiere que el editor trabaje sobre un tipo de árbol particular, basta con instanciar la nueva estructura que se desea editar en la clase ToFromRosa.

2. Los resultados obtenidos en el trabajo son independientes de la interfaz, es suficiente una terminal de caracteres con capacidad para direccionar de modo directo cada punto de la pantalla.

3. El modelo de programa propuesto es básicamente independiente de la herramienta gráfica que se esté utilizando, en efecto, haciendo pequeñas modificaciones en áreas concretas del programa (que no tocan la esencia del editor), se obtiene la aplicación con otras herramientas[8].

4. Una versión más compacta de la aplicación obtenida podría construirse en base a la composición de fragmentos utilizados en esta aplicación. Por ejemplo, se podría intentar implementar más de una operación con un solo fold, bastaría con ver la forma de combinar las álgebras que aquí se han construido. Lo rescatable de la solución obtenida es que todas las funcionalidades esenciales están escritas en términos de álgebras.

[8] Para este trabajo se utilizó la librería de gráficos graphicsLib [6].

5. En base a la experiencia obtenida al desarrollar el editor, se recomienda el siguiente método para llegar a soluciones "generales":

   - **Buscar un buen modelo de implementación con ejemplos simples que den lugar a una aplicación particular:** Analizar modelos y soluciones utilizando un ejemplo sencillo. Esto permite que el programador no se distraiga en detalles de un producto complicado y se concentre en aspectos esenciales para encontrar un buen modelo.
   - **Aplicar el modelo para obtener una aplicación general:** Una vez encontrado un buen modelo, aplicarlo para el caso más general.
   - **Aplicar el ciclo abstracción - especialización en busca de una aplicación genérica:** Para verificar que el producto obtenido sea una abstracción lo suficientemente general, tratar de especializarlo a productos particulares, buscando que esta especialización no implique modificar la aplicación general y sea lo más transparente posible para el usuario.

6 Trabajo futuro

Todas las extensiones que se proponen a continuación son detalladas en la referencia [1].

- Se puede, a partir de esta propuesta, construir un *generador de editores de estructuras* que reciba como entrada una gramática (tipo de datos) y retorne un editor de objetos que obedezcan a la misma. Esto implicaría la aplicación programación politípica (utilizar tipos de datos como argumentos de funciones) o convertir los tipos en valores de primera clase.

- Se podría permitir que la estructura guarde elementos de diferentes tipos, bastaría con hacer una abstracción de datos que los agrupe en uno solo.

- La solución propuesta podría ser extendida para obtener un editor de un árbol que permita guardar información en nodos no terminales. Esto en virtud a que el modelo usado para la solución guarda información en cada nodo (independientemente de que éste sea terminal o no), simplemente que no se implementan las funcionalidades propias de un árbol de esas características. Por lo anterior, lo único que habría que hacer es dar la funcionalidad que corresponde sin necesidad de cambiar el modelo.

- Permitir que el foco de selección pueda abarcar más de un subárbol a la vez es tarea fácil, para lograrlo, solo habría que activar una tecla como (ctrl+ la presión del mouse) para indicar que la selección continúa, guardar en una lista todos los puntos que han sido escogidos por el mouse y modificar la función de selección de modo que en lugar de recibir un punto (ptoMouse), reciba una lista de puntos. En tal caso solo se tendría que modificar la función "toca" de la operación de selección (ver referencia [1]), para que trabaje con una lista de puntos, pero el álgebra no necesita ser modificada.

- En cuanto a la interfaz, solo se manejan objetos rectangulares o cuadrados (texto). Para manejar objetos de diferentes formas, bastaría con tener los parámetros necesarios para calcular sus áreas. El trabajo que se deba hacer puede ser capturado en la función "toca" sin que el resto del código cambie.

- En el editor de estructuras logrado, se tiene una sola forma de desplegar la estructura: cada componente es desplegado en una fila diferente y se usan indentaciones para reflejar el nivel que tiene el componente dentro de la estructura. Todo esto es capturado en la función arbolInformado. Se podrían implementar otras formas de desplegar la estructura, siguiendo la misma idea. Otras formas de despliegue se pueden encontrar en la referencia [1].

- Es posible utilizar el editor de árboles rosa obtenido para editar documentos html, expresiones, lenguajes de programación, etc. Sin embargo, el usar el editor para estos objetos en las condiciones en las que está, significa trabajar artesanalmente ya que se deben elaborar cada una de sus construcciones desde cero. Por lo anterior, resultaría de mucha utilidad construir una librería de "templates" para los componentes estándares que tiene el objeto para el cual se quiere el editor.

Referencias

[1] N.T. Aparicio. Un editor genérico de estructuras. Tesis de Maestría, Universidad Mayor de San Simón, 2001.

[2] R. Backhouse, P. Jansson, J. Jeuring, y L. Meertens. *Generic Programming - An Introduction.* 1998.

[3] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall Europe, University of Oxford, 2da. edición, 1998.

[4] R. Van Geldrop, J. Jeuring, y D. Swiestra. Deel 1 gramticas & ontleden deel, 1998.

[5] W. Kahl, O. Braun, y J. Scheffczyk. Editor combinators - a first account. Reporte técnico, Universität def Bundeswehr, Junio, 2000.

[6] A. Reid. The hugs graphics library. Reporte Técnico CT 06520, Departament of Computer Science, New Haven.

[7] T. Reps y T. Teitelbaum. *The Synthesizer Generator- Reference Manual.* Department of Computer Science, Cornell University.

[8] B. Sufrin y O. de Moor. Modeless structure editing. Programming Research Group.

[9] S. Thompson. *The Craft of Functional Programming.* Addison Wesley Longman Limited, 2da. edición, 1999.