Desarrollo de Navegador Web con Haskell*

Carlos Gómez✉[1]

[1]Universidad Mayor de San Simón , Facultad de Ciencias y Tecnología

✉Correspondence: Carlos Gómez <carliros.g@gmail.com>

Abstract

Este artículo describe el desarrollo de un Navegador Web con un lenguaje de programación académico e innovador como lo es el lenguaje de programación funcional Haskell.
A pesar de que normalmente se utilizan lenguajes imperativos y convencionales para el desarrollo de éste tipo de programas, Haskell ha sido de bastante utilidad en el desarrollo, beneficiando al proyecto con varias de sus características, entre las más importantes: código modular, funciones de alto-orden, evaluación no estricta y emparejamiento de patrones.
El desarrollo consiste básicamente en aplicar un conjunto de transformaciones a una entrada hasta obtener una salida renderizable para la pantalla de un computador.
La entrada se encuentra descrita en una versión simple de un lenguaje de marcado como lo es HTML/XHTML junto con una descripción de CSS. A ésa entrada se aplican diferentes transformaciones a los tipos de datos hasta completar la información para la renderización. Como resultado del desarrollo, se ha obtenido un Navegador Web con soporte de un subconjunto de la gramática de XHTML y CSS, y 48 propiedades de CSS.
Palabras clave: *Navegador Web (Web Browser), Programación Funcional, Haskell, Renderización, XHTML, CSS.*

Introducción

Un Navegador Web es un programa informático del lado del cliente, que se encarga de renderizar documentos que pueden estar hospedados tanto en la Internet o en la misma computadora Sebesta (2006). Los actuales Navegadores Web (ej. Firefox, Google Chrome, Internet Explorer, Opera, etc.) son programas gigantes, porque dan soporte a una amplia cantidad de funcionalidad (ej. HTML, XHTML, CSS, DOM, JavaScript, Flash, etc.). El desarrollo de un

*Presentado al concurso de tesis del *Congreso Nacional de Ciencias de la Computación en Bolivia* (**CCBOL**), 2011.

proyecto gigante (Navegador Web) normalmente tiene un alto costo de desarrollo, de manera que la elección del lenguaje de programación para el desarrollo, es una decisión importante para el éxito del proyecto.

En los actuales Navegadores Web se ha utilizado lenguajes de programación imperativa (C++, Java, C), lo cual demuestra que viendo el rendimiento y funcionalidad implementada, los lenguajes de programación imperativa realizan un buen trabajo en el desarrollo de estos programas gigantes.

Por otro lado, los lenguajes de programación funcional (ej. Haskell) incorporan muchas de las innovaciones recientes del diseño de lenguajes de programación Peyton Jones (2002), de manera que pueden colaborar a reducir los costos de desarrollo de un programa.

De esa manera, en éste proyecto, se ha pretendido experimentar las capacidades y características, librerías y herramientas del lenguaje de programación funcional Haskell en el desarrollo de un Navegador Web.

En las siguientes secciones de este documento, primeramente se describirá el desarrollo del proyecto, luego se mencionará los resultados obtenidos, conclusiones, limitaciones y finalmente se dará a conocer las recomendaciones para trabajos futuros.

Desarrollo del Proyecto

La principal tarea de un Navegador Web es mostrar un documento de texto en la pantalla de un computador. Esta tarea es conocida como la renderización o el proceso de renderización que sigue un conjunto de normas o reglas definidas en la especificación de CSS.

El proceso de renderización definido en el proyecto es básicamente realizado aplicando un conjunto de transformaciones a la entrada hasta obtener un formato visual de la misma. La *Figura 1* muestra las principales transformaciones, donde cada rectángulo representa una transformación junto con sus principales tareas.

En esta sección se describirá, a grandes rasgos, la forma en que se ha desarrollado la principal tarea de un Navegador Web: renderizar.

Para lo cual, primeramente se comentará sobre las principales herramientas y librerías utilizadas en el proceso de desarrollo, luego se detallará varias estructuras, modelos y tipos de datos desarrollados para el proyecto, y finalmente se describirá en mas detalle el proceso de renderización.

Principales Herramientas y Librerías

Las principales herramientas y librerías utilizadas en el proceso de desarrollo del proyecto son:

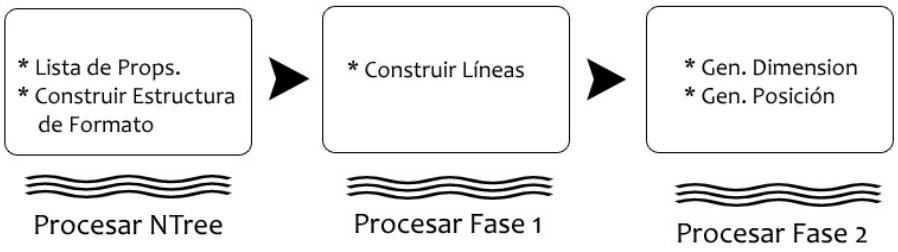

Figure 1: Transformaciones principales para renderización

- **Librería uu-parsinglib** Swierstra (2011b): Es una librería EDSL (Embeded Domain Specific Language) para Haskell que permite implementar el parser para la gramática de un lenguaje a través de una descripción similar a la gramática concreta de un lenguaje Swierstra (2008). Con esta librería se ha implementado el parser para XHTML, CSS y valores de propiedades.
- **Herramienta UUAGC** Swierstra (2011a): Es una herramienta que genera código Haskell a través de una descripción de gramática de atributos de un comportamiento. Se ha utilizado esta herramienta para la especificación del comportamiento de la mayor parte del proyecto.
- **Librería WxHaskell** Leijen (2011): Es una librería para Haskell que permite implementar la Interfaz Gráfica de Usuario (GUI) de un programa. Esta librería ha sido utilizada para implementar toda la parte gráfica del proyecto.
- **Librería libcurl** Finne (2011): Es una librería para Haskell que permite interactuar con los protocolos de red, tales como HTTP, File, FTP, etc. Esta librería ha permitido obtener todos los recursos de la Web (archivos HTML, archivos de Hojas de Estilo de CSS e imágenes) a través de una dirección URL.
- **Estructura Map**: La estructura Map es parte de la librería *Containers* de Haskell. Esta estructura permite almacenar elementos de tipo clave-valor, y provee un conjunto de funciones eficientes para manipular los elementos almacenados. Esta estructura es utilizada para almacenar la lista de propiedades de CSS.

Estructuras principales de transformación

Estructura NTree

La estructura `NTree` es una estructura rosadelfa Bird (2000) que guarda la información de los elementos de HTML en los nodos. Se tiene 2 tipos de nodos:

`NTag`, que guarda el nombre, tipo(`replaced`) y atributos del elemento. El nodo `NText` se utiliza para representar cualquier texto dentro de un elemento.
A continuación se muestra los tipos de datos que representan la estructura `NTree`, las cuales están descritas utilizando la herramienta UUAGC.

```
DATA NTree
    | NTree Node ntrees: NTrees

TYPE NTrees = [NTree]

DATA Node
    | NTag   name       : String
             replaced   : Bool
             atribs     : {Map.Map String String}
    | NText  text       : String
```

Estructura de Formato de Fase 1

Esta estructura es similar a la estructura `NTree`, la diferencia esta en que no utiliza nodos para guardar la información. Además, esta estructura guarda información sobre el contexto de un elemento, por ejemplo, si se tiene el elemento `div` de HTML, su contexto de formato seria bloque, pero si fuera el elemento `span`, su contexto seria el de linea.
Los elementos con contexto en bloque son acomodados verticalmente uno debajo del otro, pero los elemento con contexto en linea, son acomodados horizontalmente uno seguido del otro.

A continuación se muestra los tipos de datos para la Estructura de Formato de Fase 1.

```
DATA BoxTree
  | BoxContainer    name    : String
                    fcnxt   : {FormattingContext}
                    props   : {Map.Map String Property}
                    attrs   : {Map.Map String String}
                    bRepl   : Bool
                    boxes   : Boxes
  | BoxText     name    : String
                props   : {Map.Map String Property}
                attrs   : {Map.Map String String}
                text    : String

TYPE Boxes = [BoxTree]

DATA FormattingContext
  | InlineContext | BlockContext | NoContext
```

Estructura de Formato de Fase 2

De igual manera, esta estructura es similar a la anterior, porque almacena casi la misma información. Sin embargo, a este nivel, los elementos se guardan de acuerdo al contexto de formato, es decir, en lineas o bloques. Además, cada elemento puede ser separado en partes, y se utiliza el tipo `TypeContinuation` para indicar la parte a la que pertenece.

```
DATA WindowTree
  | WindowContainer     name    : String
                        fcnxt   : {FormattingContext}
                        props   : {Map.Map String Property}
                        attrs   : {Map.Map String String}
                        tCont   : {TypeContinuation}
                        bRepl   : Bool
                        elem    : Element
  | WindowText  name    : String
                props   : {Map.Map String Property}
                attrs   : {Map.Map String String}
                tCont   : {TypeContinuation}
                text    : String

DATA TypeContinuation
  | Full  | Init  | Medium | End

DATA Element
  | EWinds winds: WindowTrees
  | ELines lines: Lines
  | ENothing

TYPE WindowTrees = [WindowTree]

TYPE Lines       = [Line]

DATA Line
  | Line winds: WindowTrees
```

Hojas de Estilo en Cascada (CSS)

Uno de los documentos importantes para el proyecto fue la especificación de CSS W3C (2009), la cual provee varios modelos para renderización, un lenguaje para hojas de estilos y varias propiedades de CSS.

Modelo en Cascada

Uno de los modelos de CSS es el de cascada, que define 3 tipos de usuarios (Origen) y 3 tipos de hojas de estilo (Tipo).

```
DATA Origen
  | UserAgent | User | Author

DATA Tipo
  | HojaExterna | HojaInterna | EstiloAtributo
```

Lenguaje para Hojas de Estilo

CSS también provee un lenguaje para especificar las hojas de estilo. A continuación se muestra los tipos de datos para representar el lenguaje para las hojas de estilo.

```
TYPE HojaEstilo = [Regla]

TYPE Regla = (Tipo, Origen, Selector, Declaraciones)

DATA Selector
  |  SimpSelector    SSelector
  |  CompSelector    SSelector  operador: String    Selector

DATA SSelector
  |  TypeSelector    nombre: String     Atributos   MaybePseudo
  |  UnivSelector Atributos MaybePseudo

TYPE Atributos = [Atributo]

DATA Atributo
  |  AtribID      id: String
  |  AtribNombre  nombre: String
  |  AtribTipoOp  nombre, op, valor : String

TYPE MaybePseudo = MAYBE PseudoElemento

DATA PseudoElemento
  |  PseudoBefore   |  PseudoAfter

TYPE Declaraciones = [Declaracion]

DATA Declaracion
  | Declaracion nombre: String Value importancia: Bool
```

```
DATA Value
  | PixelNumber   Float
  | Percentage    Float
  | KeyValue      String
  | KeyColor      rgb: {(Int,Int,Int)}
  ...
  | NotSpecified
```

Propiedades de CSS

Existe casi como 80 propiedades de CSS que se utilizan para la renderización en la pantalla, las cuales se encuentran descritas en la especificación de CSS.

En el proyecto, se ha definido el tipo de dado `Property` para representar una propiedad de CSS.

```
data Property  = Property   { name              :: String
                            , inherited         :: Bool
                            , initial           :: Value
                            , value             :: Parser Value
                            , propertyValue     :: PropertyValue
                           , fnComputedValue   :: FunctionComputed
                            , fnUsedValue       :: FunctionUsed }
```

El tipo de dato `Property` permite representar una propiedad de CSS casi de forma plana y directa, porque permite detallar casi todos los valores que la especificación de CSS requiere, incluyendo el parser y las funciones para procesar los valores `computed` y `used` de una propiedad.

Valores de una Propiedad de CSS

La especificación de CSS define 4 tipos de valores para una propiedad:

- **SpecifiedValue**: corresponde al valor especificado en las hojas de estilo. Es calculado utilizando el algoritmo cascada de CSS.
- **ComputedValue**: es el valor listo para ser heredado por otras propiedades. Es calculado utilizando la función `fnComputedValue` (*Ver definición del tipo* `Property`) de cada propiedad.
- **UsedValue**: es el valor sin ningún tipo de dependencias y listo para ser renderizado. Es calculado utilizando la función `fnUsedValue` (*Ver definición del tipo* `Property`) de cada propiedad.
- **ActualValue**: es el valor que no tiene dependencias de ninguna librería. En el proyecto aun no se esta utilizando este valor.

Estos valores son representados en Haskell a través de tipo de dato `PropertyValue`:

```
data PropertyValue
  = PropertyValue   { specifiedValue    :: Value
                    , computedValue     :: Value
                    , usedValue         :: Value
                    , actualValue       :: Value }
```

Los 4 valores de una propiedad son calculados en el proceso de renderización. Por ejemplo, los valores `specified` y `computed` son calculados al procesar el `NTree` y el valor `used` es calculado al procesar la estructura de formato de fase 1.

Definición de Propiedades

En el proyecto se da soporte a varias propiedades de CSS, que son definidas utilizando el tipo de dato `Property`.

Por ejemplo, la forma de definir la propiedad **border-top-color** seria de la siguiente manera:

```
bc = Property
      {  name          = "border-top-color"
      ,  inherited     = False
      ,  initial       = NotSpecified
      ,  value         = pBorderColor
    , propertyValue = PropertyValue {  specifiedValue = NotSpecified
                                    ,  computedValue  = NotSpecified
                                    ,  usedValue      = NotSpecified
                                   ,  actualValue    = NotSpecified }
      , fnComputedValue = computed_border_color
      , fnUsedValue     = used_asComputed }
pBorderColor
    = pColor <|> pKeyValues ["inherit"]
computed_border_color iamtheroot fatherProps locProps iamreplaced iamPseudo nm prop
    =  case specifiedValue prop of
        NotSpecified
            -> specifiedValue (locProps `get` "color")
        KeyColor (r,g,b)
            -> KeyColor (r,g,b)
```

Diagrama de renderización

La *Figura 2*, muestra el diagrama de renderización, que contiene las 3 principales transformaciones de la *Figura 1*.

Éste diagrama de renderización inicia con el circulo llamado `Program init`, el cual representa la Interfaz Gráfica de Usuario (GUI) del programa.

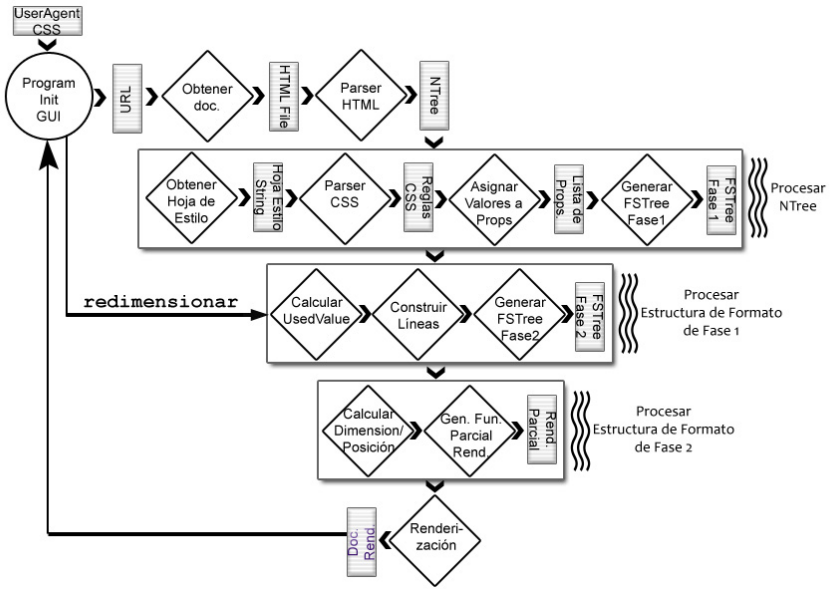

Figure 2: Diagrama de renderización

A través del GUI se inicia el proceso de renderización, como también el proceso de redimensionamiento, el cual no necesita realizar todos los pasos de renderización sino sólo los necesarios.

A continuación se describe las principales tareas del diagrama de renderización.

- **Obtener Documentos**: para renderizar documentos, primeramente se obtienen todos los documentos (archivos HTML) y recursos (hojas de estilo e imágenes) ya sea de la internet o de la misma computadora. En esta parte, se utiliza la librería libcurl junto con la dirección URL del documento.
- **Parser HTML**: luego se procede a parsear el documento obtenido y generar una estructura rosadelfa (NTree) que represente el documento.
- **Procesar NTree**: esta parte contiene varias tareas, principalmente se construye la lista de propiedades de CSS y se genera la estructura de formato de fase 1.
- **Procesar Estructura de Formato de Fase 1**: en esta parte, básicamente se construyen líneas de elementos y se genera una estructura de formato de fase 2.
- **Procesar Estructura de Formato de Fase 2**: en esta parte, se genera posiciones y dimensiones para la renderización de los elementos y también se genera una función de renderización parcial de todos los elementos.
- **Renderización**: es aquí donde se renderizan todas las propiedades de CSS de un elemento, por ejemplo en esta parte se dibujan los bordes, se asignan colores y se pinta el texto.

Resultados

Como resultado, se ha obtenido un Navegador Web con Haskell denominado **Simple San Simon Functional Web Browser**, con soporte para las siguientes características:

- Soporte para trabajar con los protocolos HTTP y File.
- Soporte para un sub-conjunto de la gramática de XHTML y XML. Se ha desarrollado un parser genérico que permite reconocer cualquier nombre de etiqueta.
- Soporte para un sub-conjunto de la gramática para el lenguaje de hojas de estilos de CSS.
- Soporte para 48 propiedades de CSS, que incluye:
  - Modificar la dimensión, posición, tipo y estilo de un elemento
  - Modificar el estilo de un texto
  - Listas
  - Generación de contenidos

El GUI del proyecto permite modificar las hojas de estilo para `UserAgent` y `User`, y también permite navegar sobre paginas ya visitadas.

Para más información, revisar: http://hsbrowser.wordpress.com.

Conclusiones

Se encontró que el lenguaje de programación funcional Haskell fue apropiado y maduro para el desarrollo de un Navegador Web. Apropiado, porque varias partes de la implementación fueron expresadas de mejor manera utilizando mecanismos de la programación funcional. Y maduro, porque en muchos casos, simplemente se utilizó las librerías ya existentes para Haskell.

Sin embargo, también se ha tenido varias dificultades en la implementación de algunos algoritmos, y algunas limitaciones de algunas librerías, las cuales se explican en la sección de *Limitaciones del Proyecto*.

Beneficios del lenguaje utilizado

Al utilizar Haskell como lenguaje de desarrollo, el proyecto se ha beneficiado con varias de sus características, entre ellas se tiene:

- Forma rica de definir los tipos de datos. Esto permite que el código sea fácil de entender y más expresivo.
- Amplia Biblioteca de funciones de Haskell.

- Varias formas de definir una función. Lo cual permite que el código sea fácil de entender y expresivo.
- Aplicación parcial de funciones. Esta característica permitió construir funciones parcialmente definidas, de manera que fueron completadas en otro punto del proceso y solo cuando eran necesarias.
- Definición de módulos. Que permite que el código este organizado en módulos, donde cada modulo tiene funciones públicas para otros módulos.

Beneficios de las herramientas y librerías

Las Herramientas y Librerías utilizadas han jugado un rol importante en la simplificación de la complejidad en el desarrollo del proyecto. A continuación se describe los beneficios de las principales herramientas y librerías utilizadas:

- **Herramienta UUAGC**
  - Permite enfocarse en la resolución del problema.
  - Código compacto (reglas de copiado de UUAGC).
  - Generar código Haskell sólo de las partes que se necesita.
- **Librería uu-parsingLib**
  - Permite que la gramática implementada sea robusta.
  - Dar soporte a la ideología de implementar todo en Haskell.
- **Librería wxHaskell**
  - Las variables de wxHaskell permitieron implementar el diagrama de renderización.

Limitaciones del Proyecto

Este proyecto corresponde simplemente a una pequeña parte de la amplia y compleja área de los Navegadores Web. En el proyecto se ha tenido varias limitaciones tanto en funcionalidad y librerías:

Limitaciones de funcionalidad: Por su complejidad, no se dio soporte a:

- Formularios, Tablas y Frames de HTML.
- Posicionamiento flotante, absoluto y fijo de CSS.

Limitaciones de la librería wxHaskell:

- En el sistema operativo Gnu/Linux, la librería wxHaskell no tiene soporte para el color transparente.
- No se encontró una función de wxHaskell que permita saber si el cambio de la fuente de un texto tuvo éxito.

A pesar de las limitaciones del proyecto, es apreciable y valórable como las características, librerías y herramientas de Haskell colaboraron a reducir los costos de desarrollo en el proyecto, permitiendo que el código sea modular, compacto y fácil de entender.

Recomendaciones para Trabajos Futuros

El trabajo presentado en este documento, puede ser extendido de varias maneras. A continuación se presenta algunas recomendaciones para trabajos futuros.

- **Parser de HTML/XHTML + DTD**. Desarrollar un parser para HTML/XHTML que implemente el comportamiento definido por un DTD(Document Type Definition).
- **Extender el soporte para la especificación de CSS**. Se puede dar soporte a: bablas, posicionamiento flotante, absoluto y fijo, y definir más propiedades de CSS (sólo se implemento 48 de las 80 propiedades para la pantalla)
- Delegar la tarea de dimensionamiento y posicionamiento a la librería gráfica, ya que actualmente este trabajo es realizado de manera manual.
- Optimizar los algoritmos de renderización del proyecto.
- Implementar y dar soporte a JavaScript.

References

Bird, Richard. 2000. *Introducción a La Programación Funcional Con Haskell*. Edited by Prendice Hall. 2da ed.

Finne, Sigbjorn. 2011. "LibCurl." http://hackage.haskell.org/packages/curl/.

Leijen, Daan. 2011. "WxHaskell, a Portable and Concise Gui Library for Haskell." http://hackage.haskell.org/package/wx/.

Peyton Jones, Simon. 2002. "Haskell 98, Language and Libraries." Haskell Community. http://haskell.org.

Sebesta, Robert W. 2006. *Programming the World Wide Web*. Edited by Adison Wesley. 3th ed.

Swierstra, S. Doaitse. 2008. "Combinator Parsing: A Short Tutorial." UU-CS-2008-044. Institute of Information; Computing Sciences, Utrecht University. www.cs.uu.nl.

———. 2011a. "Utrecht University Attribute Grammar Compiler." http://hackage.haskell.org/package/uuagc/.

———. 2011b. "Utrecht University Parser Combinator Library." http://www.cs.uu.nl/wiki/bin/view/HUT/ParserCombinators.

W3C. 2009. "Cascading Style Sheets Level 2 Revision 1 (Css 2.1) Specification." http://www.w3.org/TR/2010/WD-CSS2-20101207.