Universidad Complutense
de Madrid

Facultad de Informática

Memoria del proyecto de Sistemas Informáticos

Implementación de XPath en Haskell

Autora: Dalila Berd
Profesor Director: Rafael Caballero Roldán

Curso Académico 2009/2010

Autorización

La autora de esta memoria autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente al autor, tanto la propia memoria como el código, la documentación y/o el prototipo desarrollado.

Fdo: Dalila Berd

Resumen

XPath es un lenguaje que se utiliza para navegar a través de elementos y atributos en un documento XML. Es un lenguaje particularmente importante para las aplicaciones XML

Debido a la importancia de XPath, el principal objetivo de este proyecto es desarrollar una herramienta para evaluar expresiones XPath en base a un documento XML utilizando el lenguaje de programación funcional Haskell.

Este documento describe los pasos adoptados para desarrollar un intérprete XPath. El proyecto incluye la búsqueda de una representación adecuada utilizando el lenguaje Haskell tanto para documentos XML como para los caminos localización o búsqueda de XPath, así como el desarrollo y pruebas de la herramienta de evaluación.

Palabras Clave

XPath

XML

Representación de XPath en Haskell

Representación de XML en Haskell

Caminos de localización

Sintaxis no abreviada

Abstract

XPath is used to navigate through elements and attributes in an XML document. It is of particular importance for XML applications.

Because of the stated importance of XPath, this project's main goal is developing a tool for evaluating XPath expressions against an XML document using the functional programming language Haskell.

This report describes the steps taken to develop an XPath interpreter. The project includes finding an appropriate representation using Haskell for both XML documents and XPath location paths and the developement and testing of the evaluation tool.

Keywords

XPath

XML

XPath representation using Haskell

XML representation using Haskell

Location paths

Unabbreviated syntax

Contenido

Capítulo 1. Introducción....1
1.1 Objetivos....1
1.1.1 Resultados esperados....1
1.2 Estado del arte....2
1.2.1 Herramienta SketchPath....2
1.2.2 Haskell XML Toolbox....3
1.3 Método de trabajo y organización del proyecto....3

Capítulo 2. Representación en Haskell de XML y XPath....5
2.1 Nociones preliminares....5
2.1.1 XML....5
2.1.1.1 ¿Qué es XML?....5
2.1.1.2 Estructura de un documento XML....7
2.1.2 XPath....8
2.1.2.1 ¿Qué es XPath?....8
2.1.2.2 Caminos de Localización....9
2.2 Representación de XML en Haskell....12
2.2.1 Recursividad en XML....12
2.2.2 Representación....15
2.3 Representación de XPath en Haskell....19
2.3.1 Primer enfoque....19
2.3.2 Segundo enfoque....20
2.4 Resumen de representación....23

Capítulo 3. Desarrollo del intérprete....25
3.1 Ejes hacia delante....25
3.2 Back end....27
3.3 Ejes hacia atrás....28
3.3.1 Implementación del eje “parent”....28
3.3.1.1 Primera idea....28
3.3.1.2 Segunda idea....29
3.3.1.3 Tercera idea....30
3.3.2 Buscador “Tracker”....34
3.3.3 Funciones XPath....36
3.3.4 Expresiones booleanas....38

Capítulo 4. Conclusiones....41

Capítulo 5. Posibilidades para futuros desarrollos....43

Bibliografía....47

Lista de figuras

Figura 1.1. Objetivo del proyecto....1
Figura 1.2. Entorno gráfico de SketchPath....2
Figura 2.1. Ejemplo recursividad de XML Schema....14
Figura 2.2. Representación arbórea del ejemplo XML....16
Figura 2.3. Representación del ejemplo XML 18
Figura 2.4. Representación de XML en Haskell....23
Figura 2.5. Representación de XPath en Haskell....24
Figura 3.1. Uso de “back end”....28
Figura 3.2. Representación interna de la ejecución: child::node()/child::rogue/parent::*....31
Figura 3.3. Resultado fina de la ejecución: child::node()/child::rogue/parent::....33
Figura 3.4. Consulta con varios pasos hacia atrás....34
Figura 3.5. Árbol de indices....35
Figura 3.6. Ejemplo de uso del buscador....35
Figura 3.7. Ejemplo de función count()....36
Figura 3.8. Ejemplo de función position()....37
Figura 3.9. Ejemplo de expresión booleana con texto....38
Figura 3.10. Ejemplo de expresión booleana con valor numérico....39

Lista de tablas

Tabla 2.1. Cuadro de comparación entre XML, HTML y SGML....6
Tabla 2.2. Forward Axis en XPath....11
Tabla 2.3. Reverse Axis en XPath....11
Tabla 5.1. Equivalencias entre sintaxis abreviada y no abreviada en XPath 43

Capítulo 1. Introducción

1.1 Objetivos

El proyecto pretende definir un intérprete del lenguaje de consultas para XML XPath utilizando el lenguaje funcional Haskell. Se trata de aprovechar las posibilidades de este tipo de lenguaje tales como las funciones de orden superior, pereza, estructuras para la representación de datos, etc. para definir un intérprete que dote al lenguaje de la posibilidad de tratar con documentos XML.

1.1.1 Resultados esperados

El resultado debe ser una programa en Haskell capaz de representar las características básicas de un documento XML. En concreto, comentarios, directivas, elementos y texto. Siendo estos dos últimos los más importantes.

Este programa también sera capaz de representar un consulta XPath formada por ejes: Self, Child, Descendant, Desccendant-or-self, Attribute.

Se llevará a cabo la implementación de un intérprete, que partiendo de una consulta XPath, sea capaz de devolver el resultado correspondiente con respecto al documento XML de entrada. Se pretende hacer también un analizador sintáctico "parser" para XML y convertirlo en el tipo de datos que se haya elegido para representar este tipo de documentos en Haskell, así como incluir un "back-end" que muestre el resultado de la consulta XPath en formato texto.

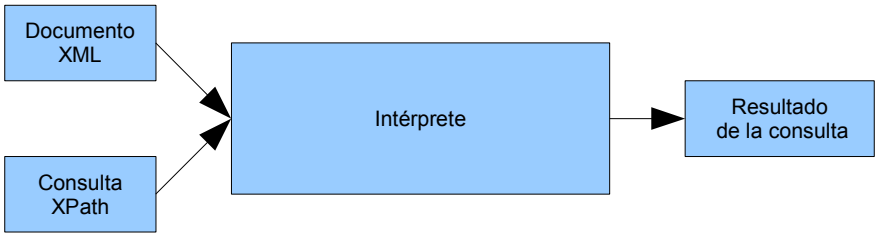


Figura 1.1. Objetivo del proyecto

1.2 Estado del arte

Actualmente, existen varios intérpretes del lenguaje de consulta para XML XPath, la mayoría implementados usando lenguajes imperativos.

La herramienta que se ha utilizado como guía para la realización de este proyecto y para la verificación de los resultados, ha sido SketchPath desarrollada en .NET.

Por lo que se refiere a interpretes de este tipo implementados en un lenguaje funcional, no hay tanto hecho. El más importante y el sin duda más utilizado es el de proyecto desarrollado por la Universidad de Ciencias Aplicadas de Wedel, Alemania, que lleva el nombre de "Haskell XML Toolbox" y que consiste en una herramienta totalmente implementada utilizando el lenguaje funcional Haskell y que sirve para el procesamiento de archivos XML

1.2.1 Herramienta SketchPath

La herramienta SketchPath es una herramienta gráfica para desarrollar y probar XPath usando uno o varios archivos XML. Su uso es sencillo y basta con elegir un archivo XML fuente sobre el que se harán las consultas XPath. Esta herramienta proporciona varias funciones para poder escribir de forma rápida y segura las expresiones y consultas XPath, entre las cuales están las funciones de auto-generación de consultas y depuración de las mismas.

El entorno gráfico de esta herramienta se muestra a continuación:

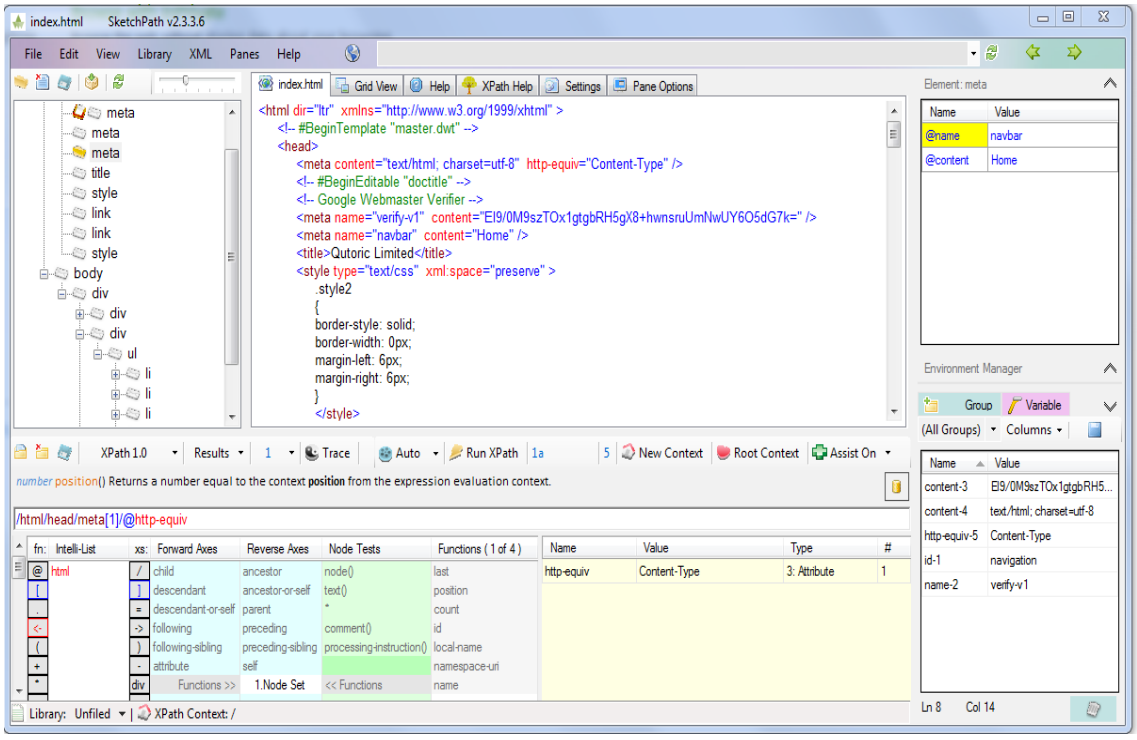


Figura 1.2. Entorno gráfico de SketchPath

1.2.2 Haskell XML Toolbox

El Objetivo principal de esta herramienta consiste en dar soporte a varios estándares de XML incluyendo, por ejemplo, Lenguaje de marcado extensible (XML) 1.0 (Segunda Edición) con su procesamiento de DDT[1] y validación, espacios de nombres en XML 1.0 (Segunda Edición), Lenguaje de consultas para XML XPath, etc.

Haskell XML Toolbox se basa en las ideas de HaXML (serie de funciones para parsear, filtrar, transformar y generar documentos XML utilizando Haskell) y HXML (analizador sin validación desarrollado para el ahorro de memoria donde se saca ventaja de la evaluación perezosa para reducir la necesidad de memoria), introduciendo un enfoque más general y flexible para el procesamiento de documentos XML. Esta herramienta utiliza un tipo de datos general para la representación de estos documentos incluyendo la parte del DTD.

1.3 Método de trabajo y organización del proyecto

El objetivo principal es la realización de un intérprete del lenguaje de consultas para XML XPath. Teniendo en cuenta que ha sido el primer contacto que he tenido con la utilización de este tipo de lenguaje, la primera parte del proyecto ha consistido en una parte teórica importante para la comprensión tanto del lenguaje de marcado XML como del lenguaje de consulta XPath.

Une vez terminada la parte teórica, el siguiente paso ha sido el de elegir una representación adecuada de los lenguajes citados anteriormente utilizando el lenguaje funcional Haskell.

Después de varios intentos de representación, elegir la representación más adecuada para los documentos XML y aquella más adecuada para la representación de XPath para a continuación desarrollar el interprete en base a ellas ha constituido la parte fundamental del desarrollo de este proyecto.

1Una DTD es el lugar en el que definimos la estructura y los elementos que forman nuestro documento XML. No son obligatorias, es decir, en XML a diferencia del SGML es posible crear documentos XML sin DTD.

Capítulo 2. Representación en Haskell de XML y XPath

En este capítulo, después de una breve presentación de los lenguajes XML y XPath, se explicará el desarrollo paso a paso del proyecto. Esto incluye tanto las pruebas y intentos de implementación de funciones que finalmente no se han incluido en el proyecto final como las partes que sí forman parte de él.

2.1 Nociones preliminares

Como citado en el punto anterior, una primera parte teórica para familiarizarme tanto con el lenguaje de marcado XML como con el de consultas XPath ha sido necesaria para la realización de este proyecto.

2.1.1 XML

2.1.1.1 ¿Qué es XML?

XML, lenguaje de marcado extensible (siglas en inglés de eXtensible Markup Language) es un conjunto de reglas para definir etiquetas semánticas que permite organizar un documento en diferentes partes. Es un metalenguaje que define la sintaxis utilizada para definir otros lenguajes de etiquetas estructurados. Un documento XML es un documento que, junto con el texto, incorpora etiquetas o marcas que contienen información adicional acerca de la estructura del texto. El estándar define cómo pueden ser esas etiquetas y qué se puede hacer con ellas. Es además especialmente estricto en cuanto a lo que está permitido y lo que no. Todo documento debe cumplir dos condiciones: ser válido y estar bien formado.

Este lenguaje fue creado por la Word Wide Web Consortium (3WC), con los siguientes objetivos:

- Ser directamente utilizable sobre Internet
- Soportar una amplia variedad de aplicaciones
- Compatible con SGML
- Debe ser fácil la estructura de programas que procesen documentos XML
- El número de características opcionales en XML debe ser absolutamente mínima, idealmente cero.
- Ser legibles y razonablemente claros
- El diseño de XML debe ser formal y conciso
- Ser fácilmente creables
- La concisión de la marca es de mínima importancia.

Se añade a continuación un cuadro-resumen de las principales diferencias que tiene XML con otros lenguajes de marcados conocidos:

	HTML/DHTML	**XML**	**SGML**
Gramática	Fija y no ampliable	Extensible	Extensible
Estructura	Monolítica	Jerárquica	Jerárquica
Nº de marcas	Fijas	Sin límite	Sin límite
Complejidad	Baja	Mediana	Alta
Diseño de páginas	Fijado por tags. Etiquetas con atributos CSS en DHTML	CSS o XSL	DSSSL
Enlaces	Simples enlaces	Poderosos enlaces (XLL)	HyTime
Exportabilidad (formatos/aplicaciones)	No	Sí	Sí
Validación	Sin validación	Pueden validarse	Obligatorio DTD
Búsquedas	Simple y a veces resuelta por *scripts* o CGI	Potente busqueda. Con capacidad para personalizarla	Son posibles potentes búsquedas
Indización/ Catalogación de páginas web	Sólo lo permite los atributos de la etiqueta <META>, e implementaciones como DC.	Una descripción abierta y personalizable con el RDF.	Algún proyecto como TEI, DLI, etc.

Tabla 2.1. Cuadro de comparación entre XML, HTML y SGML

2.1.1.2 Estructura de un documento XML

La tecnología XML busca dar solución al problema de expresar información estructurada de la manera más abstracta y reutilizable posible. Se entiende por información estructurada, aquella que se compone de partes bien definidas, y que esas partes a su vez se componen de otras partes. Se obtiene entonces un árbol de información.

Un documento XML tiene dos estructuras, una lógica y otra física. Físicamente, el documento está compuesto por unidades llamadas entidades. Una entidad puede hacer referencia a otra entidad, causando que esta se incluya en el documento. Cada documento comienza con una entidad documento, también llamada raíz. Lógicamente, el documento está compuesto de declaraciones, elementos, comentarios, referencias a caracteres e instrucciones de procesamiento, todos los cuales están indicados por una marca explícita. Las estructuras lógica y física deben encajar de manera adecuada.

Los documentos XML se dividen en dos grupos, documentos bien formados y documentos válidos:

- **Bien formados:** Son todos los que cumplen las especificaciones del lenguaje respecto a las reglas sintácticas sin estar sujetos a unos elementos fijados en un DTD. De hecho los documentos XML deben tener una estructura jerárquica muy estricta y los documentos bien formados deben cumplirla.
- **Válidos:** Además de estar bien formados, siguen una estructura y una semántica determinada por un DTD: sus elementos y sobre todo la estructura jerárquica que define el DTD, además de los atributos, deben ajustarse a lo que el DTD dicte.

Un documento XML se dice que está bien formado si encaja con las especificaciones XML de producción, lo que implica:

Estructura jerárquica de elementos

Los documentos XML deben seguir una estructura estrictamente jerárquica con lo que respecta a las etiquetas que delimitan sus elementos. Una etiqueta debe estar correctamente "incluida" en otra. Asimismo, los elementos con contenido deben estar correctamente "cerrados".

Etiquetas vacías

HTML permite elementos sin contenido. XML también, pero la etiqueta debe ser de la siguiente forma <elemento sin contenido/>.

Un solo elemento raíz

Los documentos XML sólo permiten un elemento raíz del que todos los demás sean parte. Es decir, la jerarquía de elementos de un documento XML bien formado sólo puede tener un elemento inicial.

Valores de atributos

Los valores de atributos en XML siempre deben estar encerrados entre comillas simples (') o dobles (").

Tipos de letras

XML es sensible al tipo de letra que se utiliza, es decir, trata las mayúsculas y minúsculas como caracteres diferentes. Por lo tanto, los elemento definidos como "FICHA", "Ficha", "ficha" y "fiCha" serían elementos diferentes.

Nombrando cosas

Al utilizar XML, es necesario asignar nombres a las estructuras, tipos de elementos, entidades, elementos particulares, etc. En XML los nombres tienen algunas características en común:

- No se pueden crear nombres que empiecen con la cadena "xml", "xML", "XML" o cualquier otra variante.
- Las letras y rayas se pueden usar en cualquier parte del nombre.
- También se pueden incluir dígitos, guiones y caracteres de punto, pero no se puede empezar por ninguno de ellos.
- El resto de caracteres, como algunos símbolos, y espacios en blanco, no se pueden usar.

Marcado y datos

Las construcciones con etiquetas, referencias de entidad y declaraciones se denominan "marcas". Éstas son las partes del documento que el procesador XML espera entender. El resto del documento que se encuentra entre las marcas son los datos que resultan entendibles por las personas.
Es sencillo reconocer las marcas en un documento XML. Son aquellas porciones que empiezan con "<" y acaban con ">", o bien, en el caso de las referencias de entidad, empiezan por "&" y acaban con ";".

2.1.2 XPath

2.1.2.1 ¿Qué es XPath?

Debido a la estructura jerárquica de los documentos XML, al igual que la de los archivos y carpetas en el disco, se podría decir que XPath es la versión para XML de los paths que escribimos para identificar archivos en una estructura de directorios. Tiene como fin el de

poder definir un subconjunto de nodos de un XML para su posterior procesamiento. Nos sirve para poder identificar partes específicas de un documento XML.

Un documento XML es procesado por un analizador construyendo un árbol de nodos. Este árbol comienza por un elemento raíz, de él cuelgan diferentes nodos hasta llegar a los nodos hoja que sólo pueden contener texto, comentarios, instrucciones de proceso o que también pueden estar vacíos o contener solamente atributos.

En esta misma representación arbórea del documento XML, se basa la forma en la que XPath selecciona parte del documento. De hecho los operadores de que consta XPath recuerdan la terminología utilizada a la hora de referirse a los árboles en informática: raíz, hijo, descendiente, etc. Entonces, XPath trata a todo documento como un árbol, de forma que siempre cuenta con la capacidad de poder saber si un nodo de este árbol se ajusta o no a la expresión XPath que se utilice en cada momento.

Con el objeto de seleccionar partes de un documento, XPath construye internamente un árbol de nodos llamado "árbol XPath" donde a partir de la raíz, el propio documento se diversifica a lo largo de los elementos hasta las hojas constituidas bien por valores indivisibles (autómatas) tales como, números, cadenas etc., o bien por referencias a nodos de otro documento. Por ello, XPath maneja cuatro tipos de datos:

- Cadenas (de caracteres UNICODE)
- Números (en coma flotante)
- Valores booleanos (True o False)
- Conjunto de nodos (a los que trata como una lista)

Los nodos de un árbol XPath, pueden ser de siete tipos:

- Raíz
- Elemento
- Atributo
- Texto
- Comentarios
- Instrucciones de proceso
- Espacio de nombres

En este árbol cada nodo intermedio contiene listas ordenadas de nodos hijos, siendo la relación entre un nodo padre y un nodo hijo que el padre contiene al hijo. Entonces, además del nodo raíz, solo pueden tener hijos los siguientes nodos: elementos, comentarios, texto e instrucciones de proceso, mientras que los nodos atributo y los nodos espacio de nombres se considera que solo describen a su padre por lo que se asume que no contienen nodo alguno.

2.1.2.2 Caminos de Localización

Expresiones

El bloque básico de construcción del lenguaje son las expresiones. No se les llama instrucciones ya que XPath es un lenguaje declarativo. Estas expresiones son construidas a partir de palabras clave, símbolos y operadores.

El tipo de expresiones más importantes y cuya implementación se llevará a cabo en este proyecto son las llamadas "caminos de localización" o "location paths". Existen dos tipos de camino de localización: Los absolutos y los relativos. En este proyecto se implementará los caminos de localización relativos. La sintaxis de un camino de localización se asimila mucho a la usada en Linux para describir los directorios que forman una unidad de disco. Sin embargo, solo se asimila la sintaxis. El significado, en cambio, es totalmente distinto.

Por ejemplo:

/usr/home/proyecto/docs: hace referencia a un único directorio: "docs" que cuelga del conjunto de directorio /usr/home/proyecto. Sin embargo, la siguiente expresión XPath: /libro/capitulo/párrafo hace referencia a todos los elementos "párrafo" que cuelgan directamente de cualquier elemento "capitulo" que cuelga de cualquier elemento "libro" que cuelga del nodo raíz.

El camino de localización representa el conjunto de nodos sobre los que se evalúa una expresión XPath, que obtendrá como resultado otros conjuntos de nodos, formados por aquellos que cumplen los criterios específicos en la expresión. Para dar los nodos del trayecto, se empieza designando un punto concreto, llamado nodo contexto.

Nodo Contexto

El contexto de una expresión es toda aquella información que puede incidir en el resultado de la evaluación de la misma. Un camino de localización siempre tiene un punto de partida llamado nodo contexto, que es el nodo del árbol XPath del documento desde el que se inicia la búsqueda. Por lo tanto, en los caminos de localización, a menos que se indique un camino explícito, se entenderá que el camino de localización parte del nodo que en cada momento se está procesando.

Elementos de un camino de localización

En todo camino de localización de XPath se puede distinguir tres elementos básicos:

- Ejes
- Nodos
- Predicados

Un camino de localización tendrá pues la forma siguiente:

nombreEje :: nodoComprobación [predicado(s)]

Ejes:

Los ejes son elementos encargados de especificar en el camino de localización la relación existente dentro del árbol XPath entre el nodo contexto de donde se parte y los distintos nodos que se quieren seleccionar en el camino.

En XPath existen trece tipos de ejes, que se dividen en dos grupos:

Los ejes con dirección hacia delante (Forward Axis):

Nombre	Alcance
child	Hijos del nodo actual
descendant	Todos los descendientes del nodo actual sin importar el grado, es decir hijos, hijos de hijos, etc.
attribute	Incluye solamente a los atributos
self	Nodo actual
descendant-or-self	Descendientes del nodo o el nodo mismo
following-sibling	Nodos que son hijos del nodo padre y que siguen en orden al nodo actual
following	Todos los nodos descendientes del nodo raíz que se encuentran después de la posición del nodo actual y no son descendientes del mismo.
namespace	Solo existe por compatibilidad para XPath 1.0 y contiene los nodos de namespace del nodo

Tabla 2.2. Forward Axis en XPath

y los ejes con dirección hacia atrás (Reverse Axis)

Nombre	Alcance
parent	Padre del nodo
ancestor	Todos los ancestros del nodo, padre, padre del padre etc.
preceding-sibling	Nodos que son hijos del nodo padre y que se encuentran anteriores al nodo actual
preceding	Todos los nodos descendientes del nodo raíz que se encuentran antes de la posición del nodo actual y que no son ancestros del mismo
ancestor-or-self	Ancestros del nodo o el nodo mismo

Tabla 2.3. Reverse Axis en XPath

Nodos de comprobación o búsqueda:

Los nodos de comprobación o búsqueda (node test) son los encargados de identificar un nodo o nodos concretos dentro de un eje; su función dentro del camino de localización es que cada eje pueda determinar un tipo de nodo (llamado principal) a la hora de efectuar la selección.

Este nodo de comprobación puede indicarse bien por nombre o bien por tipo.

- node(): Devuelve todos los nodos de cualquier tipo. Así para selecionar todos los nodos descendientes del tipo párrafo se escribe: child :: párrafo / descendant :: node()
- * : Llamado “wildcard”, selecciona los nodos principales de cada camino a excepción de los nodos de tipo: texto, comentario e instrucción de proceso
- text(): Devuelve cualquier nodo de tipo texto
- comment(): Devuelve cualquier nodo de tipo comentario
- processing-instruction(): Devuelve cualquier nodo de tipo instrucción de proceso

Predicados:

Los predicados son elementos que en un camino de localización permiten restringir, dentro del conjunto de nodos seleccionados por un eje, a aquellos nodos que cumplen una cierta condición debidamente especificada. Permite pues filtrar un conjunto de datos.

Estos predicados se declaran entre corchetes “[]”

2.2 Representación de XML en Haskell

Como se ha citado anteriormente una parte importante del desarrollo del proyecto ha sido encontrar una representación adecuada y sencilla para la representación del lenguaje de marcado XML.

2.2.1 Recursividad en XML

Se podría pensar en un documentos XML como una estructura recursiva. De hecho, los esquemas XML (XML schema) aparte de ser una herramienta para validar los documentos XML, si están bien estructurados y documentados, sirven para comprender en un solo vistazo funcionalidades y/o módulos enteros.

Al definir los esquemas, podemos encontrarnos con la necesidad de definir una recursividad que se puede dar en los documentos XML. Imaginemos, por ejemplo, un departamento de una empresa que, a su vez, está compuesto por varios departamentos y estos a su vez están compuestos por otros departamentos y así sucesivamente.

Sea el ejemplo siguiente que ayuda a entender esta idea:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="empresa">
    <xs:annotation>
      <xs:documentation>Comment describing your root element</xs:documentation>
    </xs:annotation>
    <xs:complexType>
```

```
        <xs:sequence>
          <xs:element name="nombre" type="xs:string"/>
          <xs:element name="departamentos" minOccurs="0">
            <xs:complexType>
              <xs:sequence>
                <xs:element name="departamento" type="departamento" maxOccurs="unbounded"/>
              </xs:sequence>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:complexType name="departamento">
      <xs:sequence>
        <xs:element name="nombre" type="xs:string"/>
        <xs:element name="numeroTrabajadores" type="xs:int"/>
        <xs:element name="subdepartamentos" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="departamento" type="departamento" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

En el esquema anterior, se puede observar que además del nodo raíz del esquema "empresa", esta definido otro elemento aparte denominado "departamento"

```
    <xs:complexType name="departamento">
      <xs:sequence>
        <xs:element name="nombre" type="xs:string"/>
        <xs:element name="numeroTrabajadores" type="xs:int"/>
        <xs:element name="subdepartamentos" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="departamento" type="departamento" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
```

Una vez definido, este elemento puede ser utilizado tantas veces como se quiera dentro de la definición del elemento/nodo principal así como dentro de sí mismo. Lo único que hay que hacer es, al definir un elemento dentro de otro, especificar que es de tipo "departamento":

```
<xs:sequence>
    <xs:element name="departamento "type="departamento" maxOccurs="unbounded"/>
</xs:sequence>
```

Se muestra a continuación este mismo ejemplo de manera gráfica:

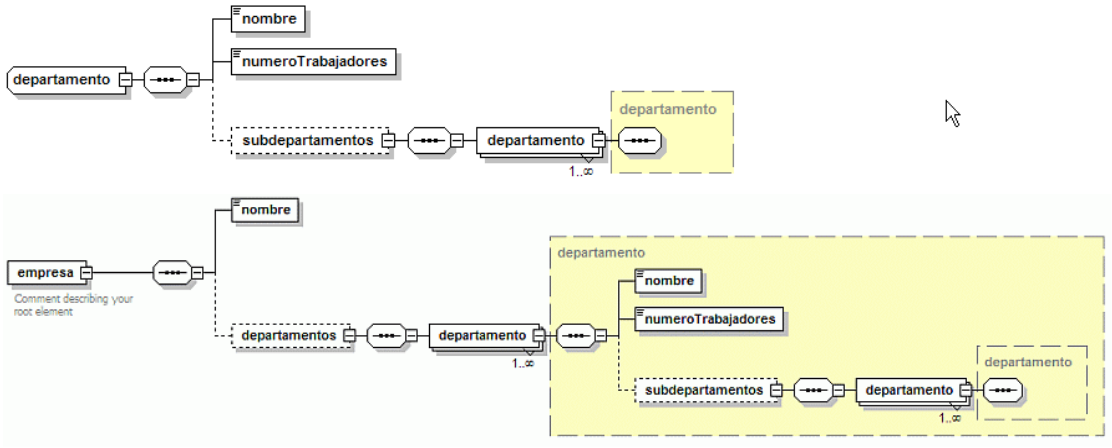


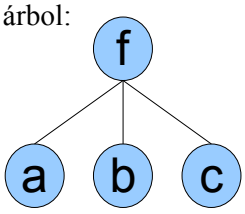
Figura 2.1. Ejemplo recursividad de XML Schema

En general, en cualquier documento XML, encontramos estructuras de la forma <f>abc</f>, en las que hay parejas de etiquetas (<f> , </f>) que recursivamente pueden contener parejas (bien formadas) de etiquetas o texto plano.

En el fondo es equivalente a escribir:

```
<f>
  abc
</f>
```

que se puede representar por un árbol:



Esta idea de recursividad es la que ha llevado a una representación los documentos XML como un tipo de datos recursivo en Haskell. Ya que una de las características más interesantes del lenguaje funcional Haskell incluye el soporte para los tipos de datos y funciones recursivas, listas, guardas y tuplas, el uso de este lenguaje parece ser una solución indicada para representar un documento XML. De hecho, las combinaciones de estas características de Haskell pueden resultar en algunas funciones muy sencillas cuya versión usando lenguajes imperativos, puede llegar a resultar extremadamente tediosa de programar.

2.2.2 Representación

Para que esta sección resulte más clara, se hará uso de un ejemplo de documento XML completo pero sencillo de entender que incluye todos los tipos de elementos importantes. Este mismo ejemplo ha sido el utilizado en el desarrollo del proyecto para llevar a cabo las pruebas y verificaciones necesarias, por lo que se hará uso de este mismo ejemplo de documento XML en el resto de esta memoria.

```
<XML>
  <!-- Test data -->
  <?value="2"?>
  <parent name="data" >
    <child id="1"  name="alpha" >Some Text</child>
    <child id="2"  name="beta" >
      <grandchild id="2.1"  name="beta-alpha" ></grandchild>
      <grandchild id="2.2"  name="beta-beta" ></grandchild>
    </child>
    <pet name="tigger"  type="cat" >
      <data>
        <birthday month="sept"  day="19" ></birthday>
        <food name="Acme Cat Food" ></food>
      </data>
    </pet>
    <pet name="Fido"  type="dog" >
      <description>
        Large dog!
      </description>
      <data>
        <birthday month="feb"  day="3" ></birthday>
        <food name="Acme Dog Food" ></food>
      </data>
    </pet>
    <rogue name="is this real?" >
      <data>
        Hates dogs!
      </data>
    </rogue>
    <child id="3"  name="gamma"  mark="yes" >
      <!-- A comment -->
      <description>
        Likes all animals - especially dogs!
      </description>
      <grandchild id="3.1"  name="gamma-alpha" >
        non-parsable text
      </grandchild>
      <grandchild id="3.2"  name="gamma-beta" ></grandchild>
    </child>
  </parent>
</XML>
```

Como se ha expuesto antes, cualquier documento XML puede ser representado mediante un árbol, por lo cual a continuación se expone la representación arbórea del documento XML anterior. Esta representación permite ver de manera simple la estructura del documento y la relación entre los elementos que la componen.

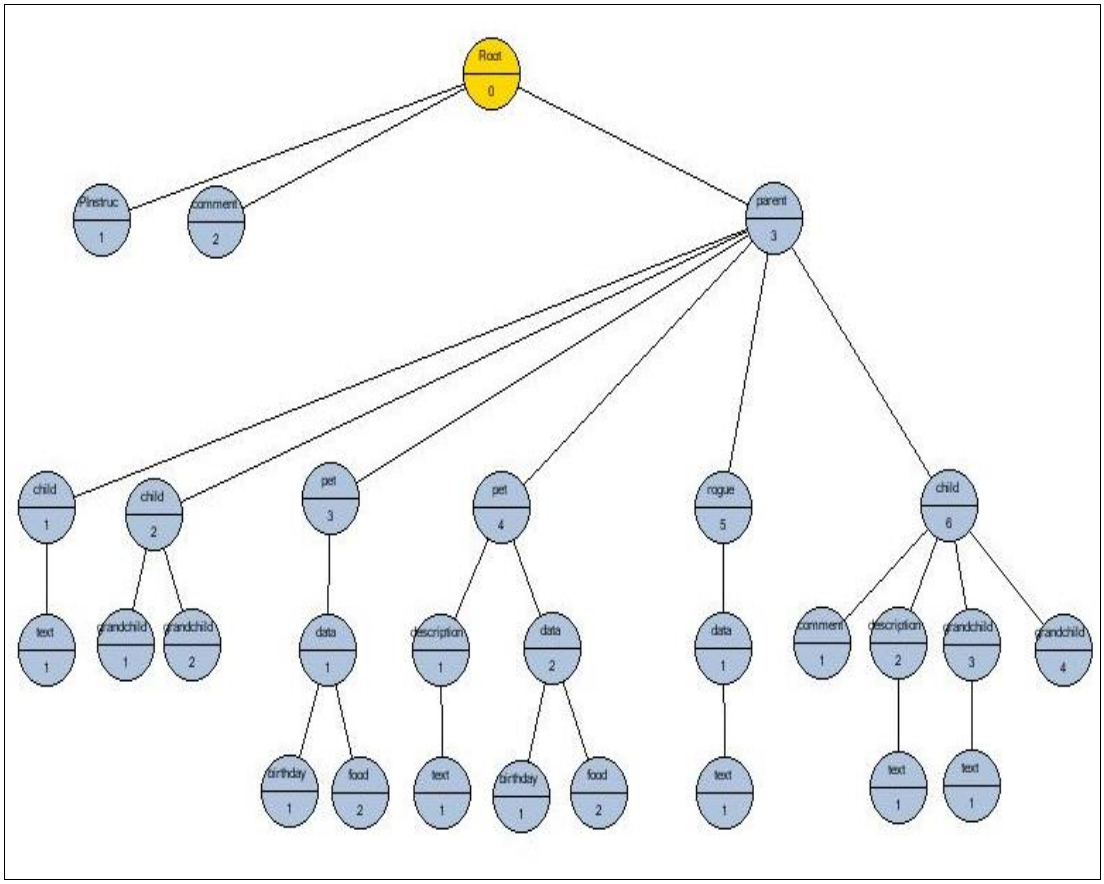


Figura 2.2. Representación arbórea del ejemplo XML

Se podría pensar en un documento XML como un conjunto de elementos, o también, como une lista de estos elementos. Por esto, la representación que se ha utilizado usando el lenguaje funcional Haskell es el de una lista usando la instrucción de renombramiento de la manera siguiente:

```
type XML = [Element]
```

Un elemento XML , como se ha explicado anteriormente puede ser de diferentes tipos. Los que en este proyecto representaremos en concreto son los siguientes:

- Tipo nodo: tipo recursivo que a su vez podrá contener otros elementos.
- Tipo texto
- Tipo comentario
- Tipo instrucción de procesamiento

Estos tipos se han representado usando constructores de Haskell.

El tipo elemento se representa como:

```
data Element = Node OpenLabel  [Element] CloseLabel
                    | XT XText | XC XComment | XPI XProcessingInstruction
type XText = String
type XComment= String
type XProcessingInstruction = String
```

Mostramos los siguientes ejemplos usando esta representación:

- Sea el siguiente elemento de tipo nodo:
  <birthday month="sept" day="19" ></birthday>
  Se representaría usando lo anterior como:
  Node (OLabel "birthday" ["month" :=: VString "sept", "day":=: VInt 19]) [] (CLabel "birthday")
  Este elemento sería una hoja, no tiene hijos por lo que el elemento recursivo es aquí un elemento vacío.
- Sea ahora el siguiente elemento de tipo texto plano:
  Likes all animals - especially dogs!
  Su representación sería:
  XT "Likes all animals - especially dogs!"
- Para un elemento de tipo comentario como:
  <!-- A comment →
  Su representación sería:
  XC "A comment"
- De manera similar, una instrucción de proceso como la del ejemplo siguiente:
  <?value="2"?>
  Sería representada como:
  XPI "value= 2"

En esta representación elegida para los elementos que componen el documento XML, aparecen las parejas de etiquetas de las que hablamos anteriormente. Estas son las llamadas aquí: “OpenLabel” y “CloseLabel” y que están definidas como sigue:

```
data OpenLabel = OLabel String [Attributes]
data Attributes = String :=: Value
data Value = VInt Integer | VFloat Float | VString String
data CloseLabel = CLabel String
```

Se puede ver que en esta representación se incluyen los atributos que pueda tener un elemento en lo que llamamos “OpenLabel” o etiqueta de apertura. Los atributos podrán ser de diferentes tipos; números enteros, números en coma flotante o cadenas de caracteres.

La representación del ejemplo completo de documento XML expuesto anteriormente y con la que se ha desarrollado el proyecto, se muestra a continuación:

```
file :: XML
file = [
XC "Text data",
XPI "value= 2",
Node (OLabel "parent" ["name" :=: VString "data"]) [
    Node (OLabel "child" ["id" :=: VInt 1, "name" :=: VString "alpha"])
        [XT "Some text"]
     (CLabel "child"),
    Node (OLabel "child"["id" :=: VInt 2, "name" :=: VString "beta"])
        [Node (OLabel "grandchild" ["id" :=: VFloat 2.1, "name" :=: VString "beta-alpha"])
            []
         (CLabel "grandchild"),
        Node (OLabel "grandchild" ["id" :=: VFloat 2.2, "name" :=: VString "beta-beta"] )
             []
        (CLabel "grandchild")]
     (CLabel "child"),
    Node (OLabel "pet" ["name" :=: VString "tigger", "type" :=: VString "cat"])
        [Node (OLabel "data" [])
            [Node (OLabel "birthday" ["month" :=: VString "sept", "day":=: VInt 19])
                []
             (CLabel "birthday"),
            Node (OLabel "food" ["name" :=: VString "Acme Cat Food"])
                []
            (CLabel "food")]
        (CLabel "data")]
    (CLabel "pet"),
    Node (OLabel "pet"["name" :=: VString "Fido", "type" :=: VString "dog"])
        [Node (OLabel "description" [])
             [XT "Large dog!"]
         (CLabel "description"),
        Node (OLabel "data" [])
             [Node (OLabel "birthday" ["month" :=: VString "feb", "day":=: VInt 3])
                []
             (CLabel "birthday"),
            Node (OLabel "food" ["name" :=: VString "Acme Dog Food"])
                []
            (CLabel "food")]
        (CLabel "data")]
    (CLabel "pet"),
    Node (OLabel "rogue" ["name" :=: VString "Is this real?"])
        [Node (OLabel "data" [])
            [XT "Hates dog"]
        (CLabel "data")]
    (CLabel "rogue"),
    Node (OLabel "child" ["id" :=: VInt 3, "name" :=: VString "gamma", "mark" :=: VString "yes"])
        [XC "A comment",
         Node (OLabel "description" [])
            [XT "Likes all animals - especially dogs!" ]
        (CLabel "description"),
        Node (OLabel "grandchild" ["id" :=: VFloat 3.1, "name" :=: VString "gamma-alpha"])
            [XT "non parsable text "]
        (CLabel "grandchild"),
         Node (OLabel "grandchild" ["id" :=: VFloat 3.2, "name" :=: VString "gamma-beta"])
            []
         (CLabel "grandchild")]
    (CLabel "child")]
(CLabel "parent")
]
```

Figura 2.3. Representación del ejemplo XML

2.3 Representación de XPath en Haskell

Como hemos visto en las nociones preliminares, un camino de localización o "location path" es el tipo más importante de expresiones XPath. Hemos visto también que este tipo de expresión puede ser de dos tipos:

- Camino de localización absoluto: empieza siempre por "/".
- Camino de localización relativo: no empiezan por "/".

Aunque en la versión final que se utilizó para el desarrollo del proyecto no se contemplan todas las posibilidades, en un primer momento se intentó hacer una representación muy general de los caminos de localización. Se detallarán a continuación los diferentes enfoques y versiones que se hicieron hasta llegar a la representación que finalmente fue elegida para el desarrollo de este proyecto.

2.3.1 Primer enfoque

La primera representación que se hizo para XPath y que se mostrará a continuación, complicaba mucho la representación de cualquier camino de localización por muy simple que éste fuera. Esto fue debido al uso de muchas contructoras Haskell que pretendían facilitar la interpretación de la representación usando nombres completos y cercanos a la sintaxis original de XPath. El resultado fue una representación poco legible que además dificultaba el procesamiento de cada camino de localización por el intérprete, objeto de este proyecto.

Esta representación es:

```
type XPath = LocationPath
data LocationPath = Slash
                  | Absolute RelativeLocationPath
                  | LP RelativeLocationPath
data AbsoluteLocationPath = Slash
                          | ASlash RelativeLocationPath
data RelativeLocationPath = Step Paso
                  | RelativeLocationPath :/ Paso deriving Show
```

Aquí "Paso" se refiere a la forma general de un camino de localización que veíamos antes nombreEje :: nodoComprobación [predicado(s)] y el cual detallaremos más adelante.

Como vemos el camino de localización relativo "Relative Location Path" es representado de como un conjunto de "pasos" usando las contructoras "Step" y ":/". Esto complicaba la representación de un camino de localización compuesto de más de un paso (y que son los caminos de localización más comunes). Por ejemplo, sea el camino de localización relativo siguiente:

Ancestor:: comment()/ child::*

su representación usando lo anterior (aunque aún no hemos visto cómo está definido cada "Paso") sería:

```
LP ((((Step(Ancestor:>Comment)):/(Child :> All))))
```

Se puede observar que la utilización de tantas constructoras en vez de facilitar la lectura de la expresión, la hace ilegible. Por otro lado la necesidad de utilización de paréntesis para cada paso, dificulta la representación y aumenta la posibilidad de cometer errores.

2.3.2 Segundo enfoque

El problema de los paréntesis que hemos visto en el punto anterior, es el que ha llevado a pensar en este segundo enfoque: un camino de localización se podría ver como una lista de pasos. El uso de paréntesis ya no sería necesario y la representación de cualquier camino de localización compuesto por pocos o muchos pasos se haría más fácil y legible.

Parece entonces que pensar en un camino de localización como una lista de pasos a seguir podría resultar en una representación más sencilla para este tipo de expresiones. Es de hecho esta idea de representación aunque un poco simplificada la que se utilizó para el resto del desarrollo.

```
type XPath = LocationPath
type LocationPath = RelativeLocationPath
                  | AbsoluteLocationPath
type AbsoluteLocationPath = Slash
                          | ASlash RelativeLocationPath
type RelativeLocationPath = [ Step ]
```

Como se ha explicado anteriormente, en el desarrollo final del proyecto no se hizo uso de los caminos de localización absolutos sino solamente de los relativos. Es fácil a partir de la implementación del intérprete que se llevó a cabo en este proyecto y que se detallará en el siguiente capítulo y haciendo uso de esta representación, ampliar el interprete para reconocer también los caminos de localización absolutos. Esto se explicará al final de esta memoria.

Para el resto del desarrollo se ha hecho uso de la siguiente versión simplificada de esta última representación:

```
type XPath = LocationPath
type LocationPath = RelativeLocationPath
type RelativeLocationPath = [ Step ]
```

Se puede observar que no sería estrictamente necesario el uso de tantos renombramientos y que bastaría con declarar: type RelativeLocationPath = [Step] ya que es el único tipo de camino de localización que se va a utilizar. Sin embargo, se ha decidido dejar estos renombramientos más generales para su uso en las declaraciones de tipo de las funciones Haskell en vistas de eventuales modificaciones del intérprete para permitir que sea más general procesando no solamente caminos de localización relativos, sino también los absolutos como indicado anteriormente, o cualquier tipo de expresión Xpath.

Los pasos que constituyen los caminos de localización relativos (en adelante: caminos de localización), tienen como hemos visto la forma

nombreEje :: nodoComprobación [predicado(s)]

La manera escogida de representar cada uno de los pasos usando constructoras Haskell es:

```
data Step = AxisName :> NodeTest
          | AxisName :-> NameTest
          | Step ::: Lpredicates
```

ya que un camino de localización podía o no tener predicados.

Lo que en esta representación se llama AxisName, se refiere al "nombreEje", parte que compone todo camino de localización. Como explicado anteriormente, el eje podía tener dirección hacia delante o hacia atrás. En el desarrollo de este proyecto el resultado final solo implementa los ejes, child, descendant, descendant-or-self, attribute y self, aunque se detallará en el capítulo siguiente las pruebas e intentos hechos para la implementación de los ejes con dirección hacia atrás. El nombre de Eje de manera general tendría la representación siguiente:

```
data AxisName = Ancestor | AncestorOrSelf | Attribute | Child | Descendant | DescendantOrSelf
              | Following | FollowingSibling | Parent | Preceding | PrecedingSibling | Self
              | Namespace
```

aunque la simplificación de la cual se hará uso se limita a los ejes:

```
data AxisName = Attribute | Child | Descendant | DescendantOrSelf | Self
```

Recordemos, por otra parte, que el nodo de comprobación podía ser unos de los tipos detallados anteriormente; *, node(), text(), comment(), processing-instruction(), o un nombre válido perteneciente al conjunto de etiquetas del documento XML sobre el que trabaja, por lo que este nodo de comprobación se representa por los dos casos siguientes:

```
data NodeTest = All
              |Comment
              | Text
              | Node
              | PInstruction
type NameTest = String
```

En este proyecto, se trabajará en todo momentos con expresiones XPath válidas, pero si éste no fuera el caso, bastaría con verificar que NameTest pertenece a la lista de cadenas de caracteres compuesta por todas las etiquetas, diferentes a la etiqueta "XML", del documento XML sobre el que se trabaja.

Por último, queda representar los predicados. Estos representan una parte opcional en un camino de localización. Como hemos visto en la representación correspondiente, los caminos de localización pueden contener estos filtros o no.

Los predicados son expresiones que permiten filtrar un conjunto de nodos utilizando un criterio determinado. Filtran los nodos que se obtienen como resultado del camino de localización que se ejecutó hasta el momento, dejando solo aquellos para los cuales se cumple el criterio.

Los predicados pueden ser a su vez una expresión Xpath, y de manera más concreta para este proyecto, un camino de localización relativo o ser una función que devuleve uno de los siguientes tipos:

- node-set: una colección de nodos desordenada sin duplicados
- String: una secuencia de caracteres
- boolean: verdadero o falso
- number: número real

  Ejemplos de funciones cuyas implementaciones se han desarrollado como parte de este proyecto son:

- función: number last()

  La función last() devuelve un número igual al tamaño contextual del contexto de evaluación de la expresión

- función: number position()

  la funcion position() devuelve un número igual a la posición contextual del contexto de evaluación de la expresión

- función: number count(node-set)

  La función count() devuelve el número de nodos en el conjunto de nodos argumento.

Si el predicado consta de una expresión que se evalúa como entero, entonces como resultado se obtiene el nodo que se encuentra en esa posición (es análogo a los indices de un Array, con la diferencia de que comienzan por 1).En cualquier otro caso (es decir cuando el resultado de la expresión no es un entero) el resultado de aplicar el predicado es el conjunto de nodos para los cuales la expresión resulta verdadera. Esto se verá con más claridad en la parte de desarrollo.

La representación elegida para cubrir estos casos de predicado es:

```
data Predicate = RLP RelativeLocationPath
               | Equal RelativeLocationPath  String
               | NEqual RelativeLocationPath String
               | Equ RelativeLocationPath Float
               | NEq RelativeLocationPath Float
               | Gthan RelativeLocationPath Float
               | Lthan RelativeLocationPath Float
               | GEto RelativeLocationPath Float
               | LEto RelativeLocationPath Float
               | FN FunctionName
type Argument[2] = (Int, Int)
data FunctionName = Count (RelativeLocationPath,Int)
                  | Last
                  | Position Argument      deriving (Eq,Show)
```

2La elección de “Arguments” como tupla de enteros no queda clara en este momento, pero se explicará más tarde.

2.4 Resumen de representación

A continuación se resumen las representaciones elegidas tanto para XML como para XPath haciendo uso del lenguaje funcional Haskell:

```
--- Representación de XML en Haskell
type XML = [Element]

data Element = Root | Node OpenLabel  [Element] CloseLabel
          | XT XText | XC XComment | XPI XProcessingInstruction
            deriving Eq
data OpenLabel = OLabel String [Attributes]   deriving Eq

data CloseLabel = CLabel String   deriving Eq

data Attributes = String :=: Value deriving Eq

data Value = VInt Int | VFloat Float | VString String
        deriving Eq

type XText = String

type XComment = String

type XProcessingInstruction = String
```

Figura 2.4. Representación de XML en Haskell

```haskell
---Representación de XPath en Haskell
type LocationPath = RelativeLocationPath


type RelativeLocationPath = [ Step ]


data Step = AxisName :: NodeTest
       | AxisName :-> NameTest
       | Step ::: LPredicates deriving (Eq,Show)

data AxisName = Attribute | Child | Descendant | DescendantOrSelf | Self
            deriving (Eq,Show)

data NodeTest = All
        | Comment
        | Text
        | Nodes
         deriving (Eq,Show)

type NameTest = String

type LPredicates = [Predicate]

data Predicate = RLP RelativeLocationPath
        | Equal RelativeLocationPath  String
        | NEqual RelativeLocationPath String
        | Equ RelativeLocationPath Float
        | NEq RelativeLocationPath Float
        | Gthan RelativeLocationPath Float
        | Lthan RelativeLocationPath Float
        | GEto RelativeLocationPath Float
        | LEto RelativeLocationPath Float
        | FN FunctionName
       deriving (Eq,Show)

type Argument = (Int, Int)

data FunctionName = Count (RelativeLocationPath,Int)
                | Last
                | Position Argument deriving (Eq,Show)
```

Figura 2.5. Representación de XPath en Haskell

Capítulo 3. Desarrollo del intérprete

En este capítulo se explica el desarrollo y pasos de implementación del intérprete de consultas XPath sobre un documento XML. Se detallarán los pasos dados antes de conseguir el resultado final. También se explicarán algunas de las implementaciones que se han probado y luego descartado, explicando las dificultades encontradas en algunas de ellas y las razones por las cuales finalmente no se han utilizado en la implementación final del intérprete.

3.1 Ejes hacia delante

Para poder llevar a cabo la implementación del intérprete e implementar cada una de las consultas XPath, en concreto los caminos de localización, se ha tenido que pensar en cúal es la información que se necesita mantener en cada paso del procesamiento para repetir lo menos posibles las mismas consultas ahorrando así mismo memoria y minimizando el tiempo de ejecución.

Lo primero que se ha implementado es la consulta de camino de localización básica sin filtros. Más adelante se explicará por qué se ha decidido implementar los filtros y cómo se han hecho.

Recordemos que un camino de localización básico sin filtros, o lo que hemos llamado antes “predicados”, tiene la forma:

nombreEje :: nodoComprobación

que se traduce en el tipo de dato “Step” definido anteriormente en Haskell como:

AxisName :> NodeTest
AxisName :-> NameTest

La función principal en este intérprete, mantiene la siguiente información en este orden[3]:

- El documento XML de entrada: Se ha decidido mantener el documento entero para la implementación de los ejes con dirección hacia atrás para no perder información, pues si se procesa el documento hacia delante en los primeros pasos borrando los elementos ya analizados se perdía la información necesaria para ejecutar consultas hacia atrás.
- Parte del documento que estamos analizando en cada momento, que consiste en los contextos de las anteriores consultas y de las cuales se parte para la ejecución de las consultas siguientes. Para la primera consulta este dato es entonces el documento XML completo.
- Los hijos de los nodos que están siendo procesados, para el procesamiento de los caminos de localización con dirección hacia delante. Manteniendo esta información se evita volver a analizar los mismos nodos varias veces. Las consultas son entonces más eficientes.
- El resultado: aunque el único resultado que el usuario necesita es el resultado del último paso del camino de localización de su consulta, se mantienen los resultados parciales después de cada paso por cuestiones de eficiencia. Por ejemplo, en una camino de localización donde el último paso es una consulta sobre atributos como: child::parent/attribute::name, los atributos que se han de devolver son los atributos de los nodos correspondientes al primer paso, con lo que se utiliza el resultado parcial de la primera consulta para extraer los atributos requeridos. De esta manera, y al igual que se ha explicado antes, no es necesario volver a analizar los nodos ya recorridos.

Se puede ver que para la implementación de un intérprete para este tipo de consultas, la recursión es una buena herramienta.

La función principal devuelve siempre el resultado parcial y los hijos de los nodos analizados en cada paso. Por ello, para procesar los pasos siguientes del camino de localización basta pasar los hijos del paso anterior como segundo argumento (como parte del documento que estamos analizando). En casos como consulta sobre descendientes, el resultado parcial de cada paso es también parte del resultado final, ya que se va procesando el árbol de arriba hacia abajo, guardando los descendientes del nodo contexto que cumplan la consulta. El conjunto formado por todos los resultados parciales, constituye, en este caso concreto, el resultado final.

Sea el siguiente ejemplo (manteniendo la sintaxis de XPath y trabajando siempre sobre el documento XML ejemplo de antes):

child::parent/child::pet

este camino de localización esta compuesto por dos pasos:

1. child::parent
2. child::pet

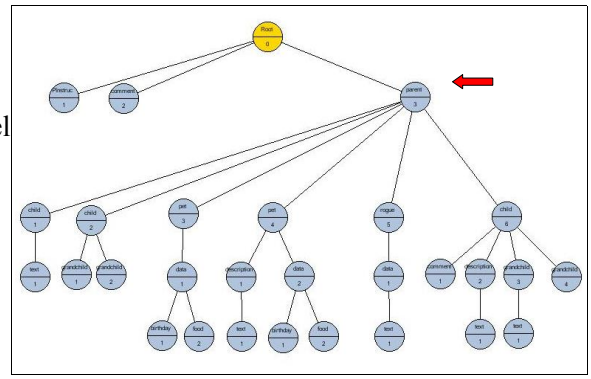
3 Estos datos son suficientes para la implementación de los ejes hacia delante sin perdida de eficiencia. En el punto siguiente se explicará por qué no es el caso para la implementación para los ejes hacia atrás.

Después de la ejecución del primer paso, se obtiene:

Como resultado parcial:

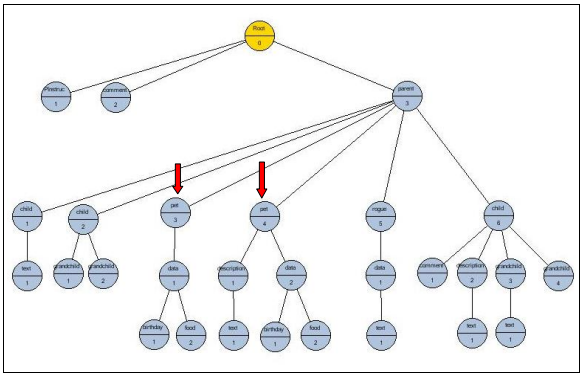
El nodo “parent” indicado por la flecha

Como hijos: Todos los nodos hijos del nodo anterior.



Para la ejecución del siguiente paso:

La parte del documento a analizar será ahora los hijos del paso anterior, y el resultado buscado sera en concreto, de todos estos hijos los que llevan nombre “pet”:



De esta manera, y aplicando recursión se obtiene fácilmente el resultado buscado.

3.2 Back end

Se ha llevado a cabo la implementación de un back end para mostrar el resultado de las consultas del camino de localización sobre el documento XML utilizado.

Este resultado se muestra representado con el tipo de dato de Haskell elegido para los XML. Para ello, se debe introducir el nombre del fichero de texto de salida donde se pretende mostrar la información. Dado que no se ha implementado ningún analizador sintáctico para las expresiones XPath, el uso de esta función requiere la escritura de la consulta XPath directamente en el fichero .hs. Esto no resulta cómodo a la hora de hacer varias consultas y por ello, aunque se muestre aquí un ejemplo, esta función no ha sido utilizada en la etapa de pruebas.

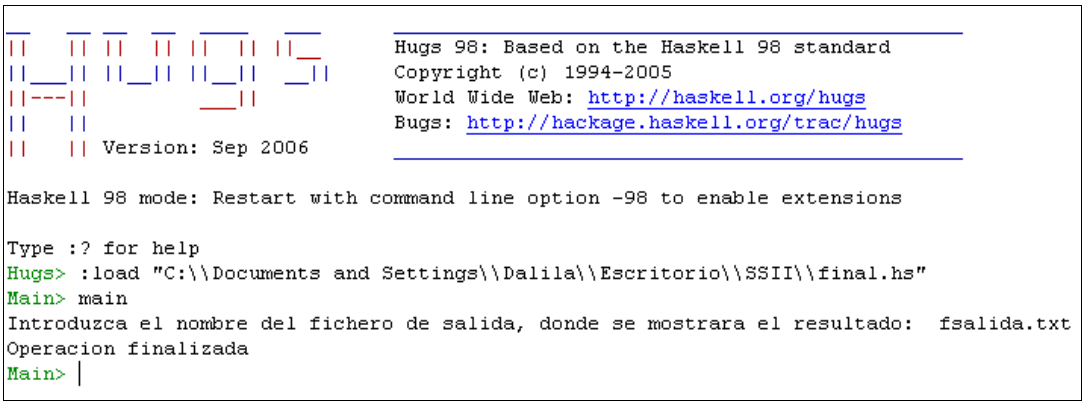


Figura 3.1. Uso de "back end"

3.3 Ejes hacia atrás

Para la implementación de los ejes con dirección hacia atrás, como "parent", "ancestor", etc, la información mantenida en la función principal que se ha detallado en el punto anterior no es suficiente para obtener una implementación eficiente sin necesidad de visitar nodos que hayan sido visitados en pasos anteriores, pues al mantener por ejemplo la parte del documento a analizar, se guardan los hijos de los nodos procesados en el paso anterior. No habría forma pues de consultar los padres o ancestros sin volver a analizar el documento completo empezando por la raíz.

Para solucionar este problema y llegar a tener una implementación eficiente, se han intentado varias implementaciones diferentes que se detallan a continuación.

3.3.1 Implementación del eje "parent"

3.3.1.1 Primera idea

Manteniendo los mismos datos que los utilizados para la implementación de caminos de localización donde los ejes utilizados son los ejes hacia delante, la primera implementación de ejes hacia atrás, en concreto la del eje "parent" visitaba varias veces los mismos nodos para obtener el resultado. Para un camino como: child::*/parent::*, lo que se hacia era lo siguiente:

- Ejecutar el primer paso y obtener como resultado, para este ejemplo en concreto, todos los hijos del nodo contexto.
- A continuación, mediante una función de búsqueda, recorría todos los nodos del resultado anterior, buscando en todo el árbol los nodos que tuvieran como hijos los del resultado antes citado. Esos eran pues los padres buscados.

Esta implementación, aunque resuelva la consulta en la mayoría de los casos, resulta muy poco eficiente. Pero, ¿qué pasaría en un ejemplo como el siguiente?

Sea un documento XML con los siguientes nodos:

---biblioteca

---libro [nombre = Introducción a XPath]

--capítulo1

--capítulo2

---libro [nombre = Introducción a XML]

--capitulo1

---revista

--capítulo1

y suponiendo que la consulta XPath es child::libro/child::capitulo1/parent::*,

el primer paso daría como resultado los dos elementos “libro”, el segundo los hijos de éstos con nombre “capítulo1”, pero aplicando la implementación anterior, al llegar al tercer paso y teniendo en cuenta que ninguno de los elementos “capitulo1” tienen atributos para ser diferenciado, se devolvería entre el resultado además de los dos libros el elemento “revista” ya que también tiene un hijo “capitulo1”. Esto es claramente incorrecto, ya que en el primer paso, el elemento “revista” no fue parte de la consulta.

Este problema se debe al hecho de no guardar la estructura y relación entre los elementos que componen el documento XML.

3.3.1.2 Segunda idea

La solución al problema planteado en primer lugar fue entonces encontrar una manera de mantener los datos en cada paso guardando la estructura del documento a procesar.

Para ello, se ha hecho uso de un nuevo tipo de datos en el que cada elemento visitado se guardaba junto a su padre en una tupla.

Esta estructura nueva de datos es la que se pasaba al paso siguiente.

A nivel de espacio de memoria, y aunque se haga uso de punteros, la información mantenida esta claramente duplicada, para un elemento nodo con 6 nodos hijo, si se visitan todos sus hijos, la información de este nodo padre estará repetida al menos 7 veces, una por cada hijo, más el nodo propio.

Usando esta misma idea de mantener cada elemento con su padre, se han hecho dos implementaciones.

La primera en la que a esa estructura sólo añadían los nodos que hayan sido parte del resultado parcial. Por ejemplo, en una consulta como child::*, solo se guardarían los elementos nodo junto con sus padres, los textos, comentarios e instrucciones de proceso no se guadarían en la estructura nueva. Para una consulta como: child::*/parent::*, bastaría con buscar en el resultado parcial del primer paso, extraer todos los nodos previamente elegidos y devolver los padres (eliminando duplicados). El resultado parcial de antes, pasaría a ser los hijos del segundo paso para pasos posteriores.

El problema en esta implementación, se muestra a continuación:

Si además de los pasos del ejemplo anterior se añade el paso: child::node(), obteniendo así la consulta siguiente: child::*/parent::*/child::node() , el problema está en que ya no tenemos la información sobre los elementos texto, comentarios e instrucciones de proceso que ahora se requieren. Esta información no hace parte de la estructura y se debería volver a procesar todo el árbol de nuevo.

Por ello, se ha añadido a la estructura en la segunda implementación tanto los nodos que hayan sido parte de los resultados parciales como los que no. Esto claramente resuelve el problema anterior, pero:

para una consulta en la que en vez de todos los tipos de elementos se pida solo los nodos: child::*/parent::*/child::*, quedaría en la estructura los elementos texto, comentarios e instrucciones de proceso. Esto es una problema ya que en esa estructura debería solo quedar los padres de los nodos que se procesan a continuación. En cambio, en este caso están los nodos hermano también. La estructura y relación entre los nodos ya no se mantiene. Este último problema se resolvió añadiendo en los resultados parciales el número de nodos visitados de forma que al avanzar, y aunque no se vuelvan a pedir todos los tipos de nodos, usando el numero de hijos procesados, se podía eliminar de la estructura todos los nodos hijos guardando así sólo los padres de los elementos a procesar.

Aunque esta implementación funciona para consultas con “parent”, sólo funciona de manera eficiente para una llamada hacia atrás. Para consultas en las que se pregunta por padres de los padres como por ejemplo: child::*/child::*/child::node()/parent::*/parent::*, el segundo “parent” no devuelve el resultado requerido salvo si se vuelve a recorrer todo el árbol de nuevo, volviendo a dar los pasos explicados para la ejecución del primer “parent”.

Esta implementación no es entonces la más eficiente para los caminos de localización compuestos por varias consultas hacia atrás.

3.3.1.3 Tercera idea

Una solución para el problema anterior es mantener información de todos los nodos que se han visitado para poder hacer todas las consultas hacia atrás requeridas por el camino de localización, sin tener que visitar varias veces los nodos. Para lograr esto, se ha pensado en mantener todos los resultados parciales que se han ido obteniendo a lo largo del recorrido, sin que este sea reemplazado en cada paso por el último resultado parcial. De esta forma cada vez que se cambia de nivel en el árbol, bastaría dar el mismo número de pasos hacia atrás en la nueva estructura para encontrar los resultados deseados.

Para ello, se ha usado una lista para guardar el resultado de cada paso como un elemento de esta lista. El resultado final de la ejecución de la consulta sería entonces el último elemento introducido en esta nueva estructura.

Para aclarar este planteamiento, se propone el siguiente ejemplo de ejecución:

sea la consulta XPath siguiente: /child::node()/child::rogue/parent::*

el resultado compuesto por los resultados parciales de cada uno de los pasos es:

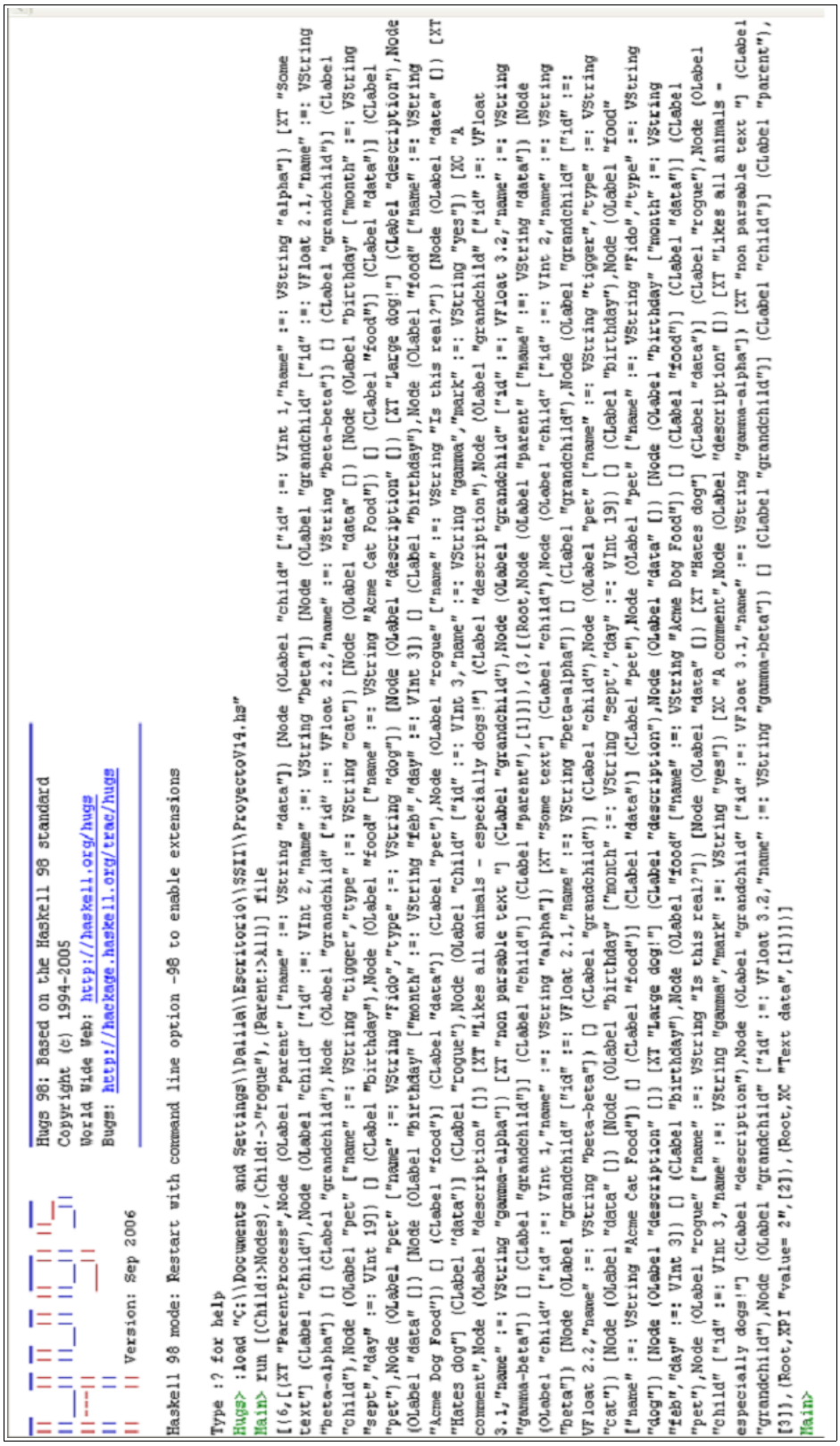


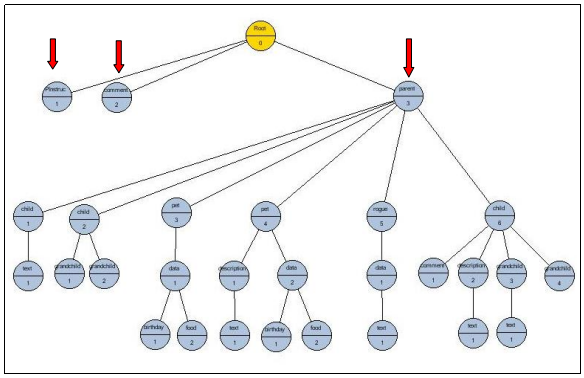
*Figura 3.2. Representación interna de la ejecución: child::node()/child::rogue/parent::**

Esta representación es interna al intérprete y no es visible al usuario final.

Se puede ver que el resultado que ahí aparece, se compone de los siguientes elementos que forman parte de cada uno de los pasos que forman parte del camino de localización del ejemplo.

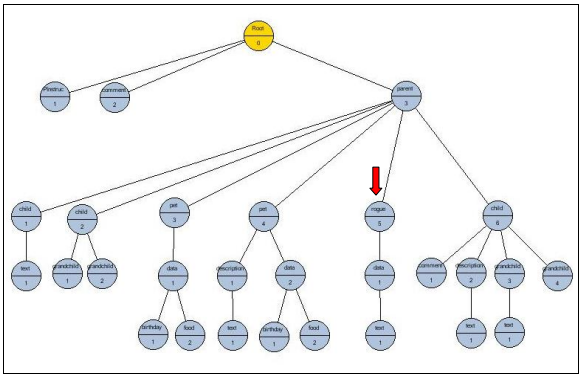
Se puede ver que como primer elemento de la estructura (el primero en ser insertado, es decir el que aparece al final, ya que se inserta por delante de la lista), está la lista formada por el resultado correspondiente al paso: child::node():

- La instrucción de proceso
- El comentario
- El elemento nodo "parent"

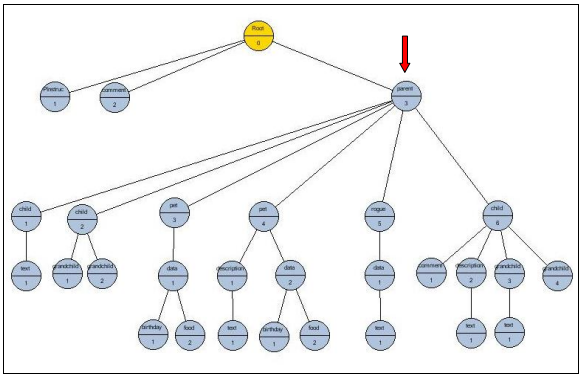


El número 3, que aparece como primer componente de la lista, representa el numero de elementos hijos del nodo contexto inicial (en este caso la raíz).

El siguiente elemento de la estructura es el nodo elemento con nombre "rogue" que corresponde al paso: child::rogue. Un vez más el número que aparece, en este caso 6, son los hijos visitados.



Y finalmente, el último resultado y que representa el resultado final es el padre de este último elemento citado y que corresponde al paso: parent::*



Como se ha indicado antes, esto es un proceso interno al intérprete no visible para el usuario. El resultado final de esta consulta, tal y como la recibiría el usuario, se muestra a continuación:

```
Hugs> run "[(Child:>Nodes),(Child:->"rogue"),(Parent:>All)] file
XmlR [Node (OLabel "parent" ["name" :=: VString "data"]) [Node (OLabel "child" ["id" :=:
VInt 1,"name" :=: VString "alpha"]) [XT "Some text"] (CLabel "child"),Node (OLabel
"child" ["id" :=: VInt 2,"name" :=: VString "beta"]) [Node (OLabel "grandchild" ["id" :
=: VFloat 2.1,"name" :=: VString "beta-alpha"]) [] (CLabel "grandchild"),Node (OLabel
"grandchild" ["id" :=: VFloat 2.2,"name" :=: VString "beta-beta"]) [] (CLabel
"grandchild")] (CLabel "child"),Node (OLabel "pet" ["name" :=: VString "tigger","type" :
=: VString "cat"]) [Node (OLabel "data" []) [Node (OLabel "birthday" ["month" :=:
VString "sept","day" :=: VInt 19]) [] (CLabel "birthday"),Node (OLabel "food" ["name" :
=: VString "Acme Cat Food"]) [] (CLabel "food")] (CLabel "data")] (CLabel "pet"),Node
(OLabel "pet" ["name" :=: VString "Fido","type" :=: VString "dog"]) [Node (OLabel
"description" []) [XT "Large dog!"] (CLabel "description"),Node (OLabel "data" []) [Node
(OLabel "birthday" ["month" :=: VString "feb","day" :=: VInt 3]) [] (CLabel
"birthday"),Node (OLabel "food" ["name" :=: VString "Acme Dog Food"]) [] (CLabel
"food")] (CLabel "data")] (CLabel "pet"),Node (OLabel "rogue" ["name" :=: VString "Is
this real?"]) [Node (OLabel "data" []) [XT "Hates dog"] (CLabel "data")] (CLabel
"rogue"),Node (OLabel "child" ["id" :=: VInt 3,"name" :=: VString "gamma","mark" :=:
VString "yes"]) [XC "A comment",Node (OLabel "description" []) [XT "Likes all animals -
especially dogs!"] (CLabel "description"),Node (OLabel "grandchild" ["id" :=: VFloat
3.1,"name" :=: VString "gamma-alpha"]) [XT "non parsable text "] (CLabel
"grandchild"),Node (OLabel "grandchild" ["id" :=: VFloat 3.2,"name" :=: VString "gamma-
beta"]) [] (CLabel "grandchild")] (CLabel "child")] (CLabel "parent")]
Main>
```

Figura 3.3. Resultado fina de la ejecución: child::node()/child::rogue/parent::

Esta implementación funciona finalmente para el caso de varias consultas hacia atrás, pero surge un nuevo problema.

Ahora, después de este tipo de consultas, cuando se hace una consulta hacia delante, no funciona. Esto es debido a que no se mantiene correctamente la información durante toda la ejecución. En esta opción de implementación se manipulan tipos de datos muy complejos y la depuración de los fallos se hace muy costosa. Por esta razón no se ha llegado a encontrar en qué punto del proceso se pierde la información sobre los hijos y consultas como:

child::*/child::*/child::*/child::*/parent::*/parent::*/parent::*/child::*

se hacen correctamente hasta el último paso donde la información sobre los hijos se ha perdido.

Se muestra un ejemplo de ejecución de consulta con varios pasos hacia atrás:

```
Main> run [(Child:>All),(Child:>All),(Child:>All),(Child:>All),(Parent:>All),(Parent:>
All),(Parent:>All)] file
XmlR [Node (OLabel "parent" ["name" :=: VString "data"]) [Node (OLabel "child" ["id" :=:
VInt 1,"name" :=: VString "alpha"]) [XT "Some text"] (CLabel "child"),Node (OLabel
"child" ["id" :=: VInt 2,"name" :=: VString "beta"]) [Node (OLabel "grandchild" ["id" :
=: VFloat 2.1,"name" :=: VString "beta-alpha"]) [] (CLabel "grandchild"),Node (OLabel
"grandchild" ["id" :=: VFloat 2.2,"name" :=: VString "beta-beta"]) [] (CLabel
"grandchild")] (CLabel "child"),Node (OLabel "pet" ["name" :=: VString "tigger","type" :
=: VString "cat"]) [Node (OLabel "data" []) [Node (OLabel "birthday" ["month" :=:
VString "sept","day" :=: VInt 19]) [] (CLabel "birthday"),Node (OLabel "food" ["name" :
=: VString "Acme Cat Food"]) [] (CLabel "food")] (CLabel "data")] (CLabel "pet"),Node
(OLabel "pet" ["name" :=: VString "Fido","type" :=: VString "dog"]) [Node (OLabel
"description" []) [XT "Large dog!"] (CLabel "description"),Node (OLabel "data" []) [Node
(OLabel "birthday" ["month" :=: VString "feb","day" :=: VInt 3]) [] (CLabel
"birthday"),Node (OLabel "food" ["name" :=: VString "Acme Dog Food"]) [] (CLabel
"food")] (CLabel "data")] (CLabel "pet"),Node (OLabel "rogue" ["name" :=: VString "Is
this real?"]) [Node (OLabel "data" []) [XT "Hates dog"] (CLabel "data")] (CLabel
"rogue"),Node (OLabel "child" ["id" :=: VInt 3,"name" :=: VString "gamma","mark" :=:
VString "yes"]) [XC "A comment",Node (OLabel "description" []) [XT "Likes all animals -
especially dogs!"] (CLabel "description"),Node (OLabel "grandchild" ["id" :=: VFloat
3.1,"name" :=: VString "gamma-alpha"]) [XT "non parsable text "] (CLabel
"grandchild"),Node (OLabel "grandchild" ["id" :=: VFloat 3.2,"name" :=: VString "gamma-
beta"]) [] (CLabel "grandchild")] (CLabel "child")] (CLabel "parent")]
Main>
```

Figura 3.4. Consulta con varios pasos hacia atrás

3.3.2 Buscador "Tracker"

En el punto anterior, se ha visto cómo el hecho de mantener la información necesaria para implementar consultas eficientes complica la depuración de errores y dificulta la corrección de las funciones implementadas.

Esto ha llevado a pensar en una forma más sencilla de llevar la información y poder acceder a cada elemento por separado manteniendo la relación existente entre los elementos y la estructura del documento XML sin tener que almacenar datos complejos.

La forma más sencilla de representar los elementos es mediante la ruta que lleva desde la raíz hasta los elementos. Esta ruta de índices, que indica el camino a seguir en el árbol desde la raíz hasta un elemento, es única y define completamente el elemento, manteniendo la ruta de todos sus ancestros.

La ruta (el camino) se representa con una lista de índices enteros empezando por la raíz con índice 0 (aunque este índice es implícito y no se almacena) . Cada hijo es numerado empezando por 1 de izquierda a derecha añadiendo el nuevo índice a la lista construida hasta llegar al elemento[4]. En cada nivel del árbol y para cada nodo padre, se empieza por el índice 1.

4 La lista de índices usada en la implementación es la inversa a la que se explica en esta sección. Esto es por cuestiones de eficiencia ya que concatener por delante tiene coste O(1) frente al coste lineal O(n) que supone concatenar por detrás para cada índice nuevo insertado.

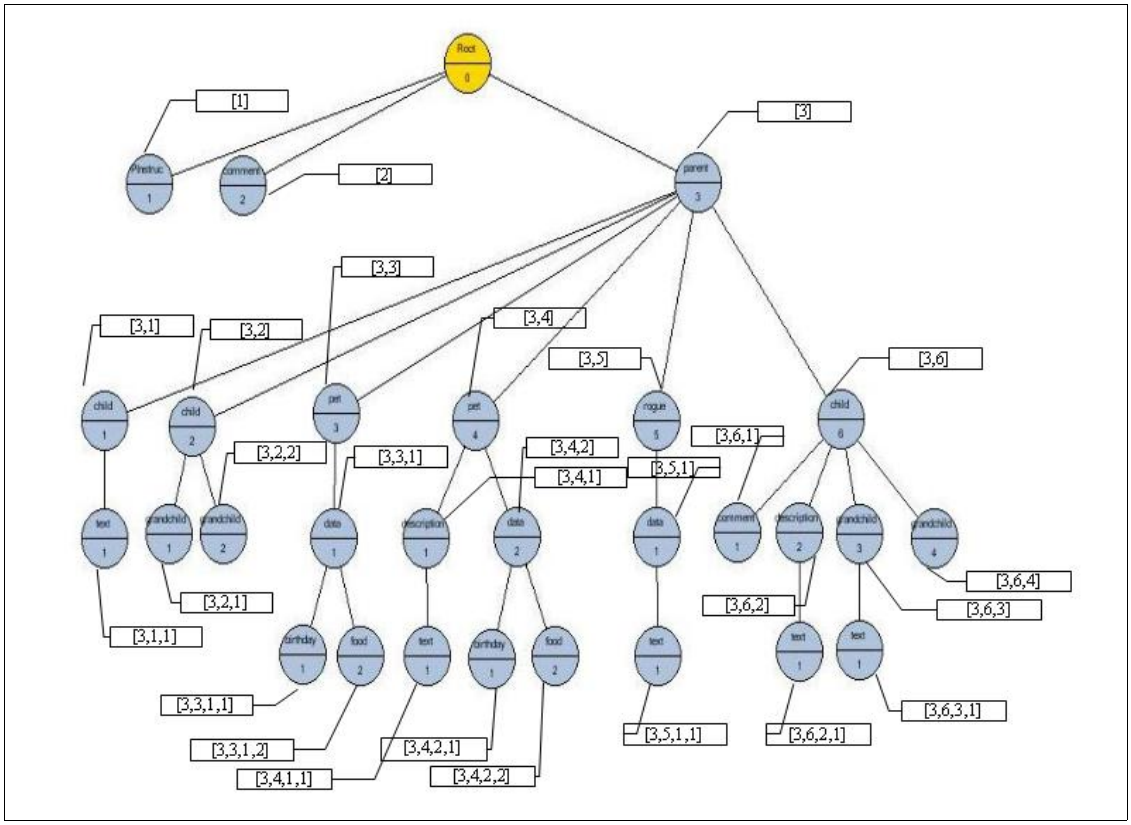


Figura 3.5. Árbol de indices

Se ve en este gráfico que cada elemento tiene asociada una ruta propia. Dos elementos distintos no podrán tener nunca la misma ruta.

Con esta idea, es fácil implementar una función que busque en el documento XML un elemento por su ruta. Se observa además que para cada elemento, basta con con eliminar el último índice para encontrar la ruta del padre. Al igual que se pueden buscar los hermanos de cada elemento sumando o restando (following-sibling y preceding-sibling respectivamente) al último índice el número adecuado.

Ejemplo: nodo texto [3,6,2,1] (child::parent/child::child/child::description/child::text())

```
__   __ __   __  _____   ____
||   || ||   ||  ||  ||  ||__
||___|| ||__|| ||__||  __||
||---||         ___||
||   ||
||   || Version: Sep 2006

Hugs 98: Based on the Haskell 98 standard
Copyright (c) 1994-2005
World Wide Web: http://haskell.org/hugs
Bugs: http://hackage.haskell.org/trac/hugs

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Documents and Settings\\Dalila\\Escritorio\\SSII\\ProyectoV14.hs"
Main> track [1,2,6,3] file
XT "Likes all animals - especially dogs!"
Main>
```

Figura 3.6. Ejemplo de uso del buscador

3.3.3 Funciones XPath

Las funciones XPath que se han implementado en este proyecto son las funciones de conjuntos de nodos de las cuales se ha hablado anteriormente.

La forma de representar estas funciones es como se ha visto antes:

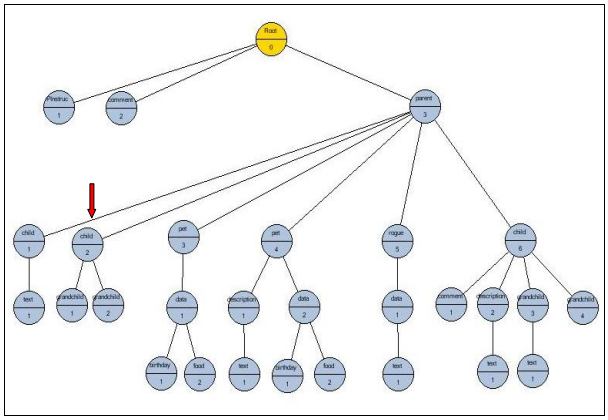
```
data FunctionName = Count (RelativeLocationPath,Int)
                  | Last
                  | Position Argument
type Argument = (Int, Int)
```

Estas funciones son:

- count(conjunto-nodos): Esta función toma un conjunto de nodos y devuelve el número de nodos que componen dicho conjunto.

  Ejemplo: Una consulta de tipo: child::*[count(child::*/child::grandchild) = 2] se escribe usando la representación elegida como:

  (Child::All):::[FN Count([Child:>All, Child:->"grandchild"],2)] y lo que devuelve son todos los elementos hijos del nodo raíz que tienen hijos, que a su vez tienen exactamente dos descendientes con nombre grandchild. En nuestro ejemplo solo existe un elemento que cumple lo requerido y es el elemento child con id = 2.



```
__   __ __  __  ____   ___                 ________________________________________
||   || ||  || ||  || ||__                  Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||                 Copyright (c) 1994-2005
||---||         ___||                       World Wide Web: http://haskell.org/hugs
||   ||                                     Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006                   ________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Documents and Settings\\Dalila\\Escritorio\\SSII\\final.hs"
Main> run [(Child::All):::[FN (Count([Child:>All, Child:->"grandchild"],2))] file
[ child [id = 2,name = beta][ grandchild [id = 2.1,name = beta-alpha][]grandchild,
grandchild [id = 2.2,name = beta-beta][]grandchild]child]
Main>
```

Figura 3.7. Ejemplo de función count()

- last(): devuelve un número igual al tamaño contextual del contesto de evaluación de la expresión. Esta función sólo tiene sentido usarla aquí como argumento de la función Position() que explicaremos a continuación.

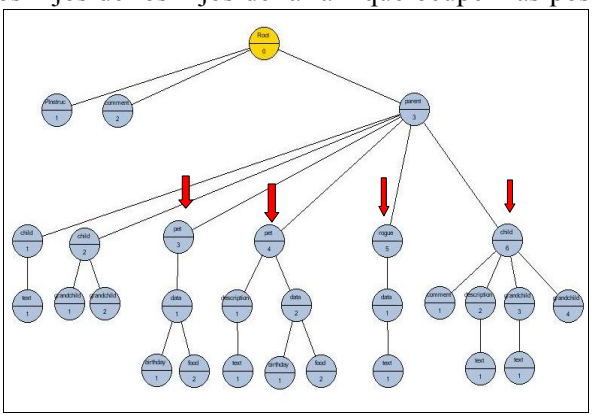
- position(), devuelve un número igual a la posición contextual del contexto de evaluación de la expresión

  Para la implementación de esta función se ha utilizado un parámetro en forma de tupla que recibe un primer entero que codifica qué comparación se tiene que llevar a cabo y un segundo entero, que es con el que se hace la comparación.

  El entero se interpreta internamente como sigue:

  - 1: Mayor que
  - 2: Menor que
  - 3: Mayor o igual a
  - 4: Menor o igual a
  - 5: igual a
  - 6: distinto de

  Ejemplo: una consulta de tipo child::*/child::*[position() >= 3] se escribe usando la representación elegida por [(Child:>All),(Child:>All):::[FN (Position(3,3))]] y que devuelve todos los hijos de los hijos de la raíz que ocupen las posiciones a partir de la tercera.



```
__   __ __  __  ____   ___                     _____________________________________
||   || ||  || ||  || ||__                     Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||                    Copyright (c) 1994-2005
||---||         ___||                          World Wide Web: http://haskell.org/hugs
||   ||                                        Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006                      _____________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Documents and Settings\\Dalila\\Escritorio\\SSII\\final.hs"
Main> run [(Child:>All),(Child:>All):::[FN (Position(3,3))]] file
[ pet [name = tigger,type = cat][ data [][ birthday [month = sept,day = 19][]birthday,
food [name = Acme Cat Food][]food]data]pet, pet [name = Fido,type = dog][ description []
[ Large dog!]description, data [][ birthday [month = feb,day = 3][]birthday, food [name
= Acme Dog Food][]food]data]pet, rogue [name = Is this real?][ data [][ Hates dog]data]
rogue, child [id = 3,name = gamma,mark = yes][ A comment, description [][ Likes all
animals - especially dogs!]description, grandchild [id = 3.1,name = gamma-alpha][ non
parsable text ]grandchild, grandchild [id = 3.2,name = gamma-beta][]grandchild]child]
Main>
```

Figura 3.8. Ejemplo de función position()

3.3.4 Expresiones booleanas

Las expresiones booleanas de comparación para poder filtrar los elementos por el contenido de sus atributos o por el texto que los componen, por ejemplo, se han implementado igualmente con la posibilidad de elegir elementos con contenido igual o distinto a un cierto valor insertado.

Como se ha visto antes, los predicados podían tener la forma siguiente:

data Predicate = RLP RelativeLocationPath
| Equal RelativeLocationPath String
| NEqual RelativeLocationPath String
| Equ RelativeLocationPath Float
| NEq RelativeLocationPath Float
| Gthan RelativeLocationPath Float
| Lthan RelativeLocationPath Float
| GEto RelativeLocationPath Float
| LEto RelativeLocationPath Float
| FN FunctionName

El primer y último caso ya se ha explicado en los puntos anteriores.

Los casos de Equal RelativeLocationPath String y NEqual RelativeLocationPath String se usan para consultas de tipo:

child::parent/child::*[attribute::name != “gamma”] que se traduce a la representación Haskell en: (Child :> All),(Child:>All):::[NEqual [Attribute :-> "name"] "gamma"]

O por ejemplo:
child::*/child::*[child::*/child::text() = “Large dog!”]
(Child:>All),((Child :> All):::: [Equal [(Child :> All),(Child :> Text)] "Large dog!"])

```
__   __ __   __  ____    ___      ________________________________________
||   || ||   || ||  ||  ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||       Copyright (c) 1994-2005
||---||         ___||             World Wide Web: http://haskell.org/hugs
||   ||                           Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006        ________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Documents and Settings\\Dalila\\Escritorio\\SSII\\final.hs"
Main> run [(Child:>All),((Child :> All):::: [Equal [(Child :> All),(Child :> Text)]
"Large dog!"])] file
[ pet [name = Fido,type = dog][ description []][ Large dog!]description, data [][
birthday [month = feb,day = 3][]birthday, food [name = Acme Dog Food][]food]data]pet]
Main>
```

Figura 3.9. Ejemplo de expresión booleana con texto

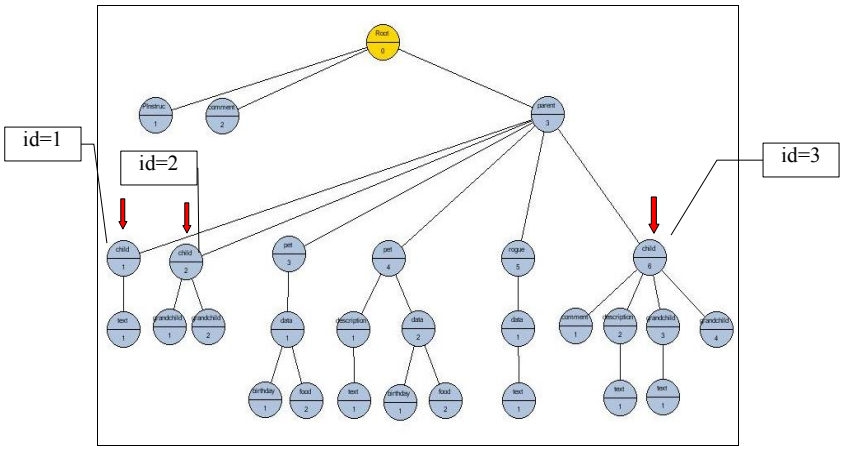
En los demás casos se puede comparar atributos con valor numérico.

Por ejemplo:

child::*/child::*[atribute:: id < 3] que se traduce en la representación Haskell por:

[(Child:>All) , (Child:>All)::: [(Lthan Attribute:->"id" 3)])]

Los únicos elementos con este camino de localización que tienen un atributo "id" son los señalados a continuación:




```
__   __ __  __  ____   ___                     ____________________________________
||   || ||  || ||  || ||__                     Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||                    Copyright (c) 1994-2005
||---||         ___||                          World Wide Web: http://haskell.org/hugs
||   ||                                        Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: Sep 2006                      ____________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs> :load "C:\\Documents and Settings\\Dalila\\Escritorio\\SSII\\final.hs"
Main> run [(Child:>All), (Child:>All)::: [(Lthan [Attribute :-> "id"] 3)]] file
[ child [id = 1,name = alpha][ I am a Text node. Child of child1]child, child [id =
2,name = beta][ grandchild [id = 2.1,name = beta-alpha][]grandchild, grandchild [id =
2.2,name = beta-beta][]grandchild]child]
Main>
```

Figura 3.10. Ejemplo de expresión booleana con valor numérico

Capítulo 4. Conclusiones

Como comentamos al principio de este documento, el objetivo principal del proyecto era desarrollar un intérprete del lenguaje de consultas para XML XPath utilizando el lenguaje funcional Haskell. Este objetivo se ha cumplido en su mayor parte, consistiendo en un herramienta capaz de evaluar caminos de localización utilizando sintaxis no abreviada de XPath. Este intérprete evalúa cualquier camino de localización XPath cuyos ejes forman parte del grupo de ejes con dirección hacia delante siguiente:

- child
- descendant
- descendant-or-self
- attribute

Por otro lado, este intérprete también evalúa caminos de localización con predicados con expresiones booleanas que permiten hacer búsquedas más precisas filtrando los resultados por valores de atributos por ejemplo, o textos. Se ha llevado a cabo la implementación de funciones XPath pertenecientes a las funciones de conjuntos de nodos como: position(), last() y count() que forman parte de las posibilidades de predicados que ofrece el lenguaje XPath.

Se puede igualmente utilizar esta herramienta de evaluación para obtener el resultado en un fichero en formato texto cuya ruta se debe indicar al iniciar la evaluación. El resultado mostrado en este fichero está representado utilizando la representación elegida e implementada en Haskell para los documentos XML ya que no se ha llevado a cabo la implementación del analizador sintáctico.

Aunque se ha intentado implementar las funciones adecuadas para poder evaluar consultas con ejes hacia atrás, este objetivo no ha sido alcanzado debido a numerosos problemas de eficiencia que ello planteaba.

Capítulo 5. Posibilidades para futuros desarrollos

Durante el desarrollo del proyecto surgieron varias ideas que podían aprovechar lo implementado hasta el momento para conseguir un intérprete de XPath para documentos XML más completo.

Una de las ideas y, basándose en las equivalencias existentes entre consultas XPath usando la sintaxis abreviada y aquellas usando la sintaxis no abreviada, sería la ampliación del intérprete para reconocer consultas XPath escritas con sintaxis no abreviada.

No haría falta volver a implementar todas las funciones y consultas para cada uno de los posibles ejes, sino que bastaría con implementar un traductor que, para cada camino de localización por ejemplo, escrito usando sintaxis abreviada, lo tradujera a su equivalente en sintaxis no abreviada y usara este mismo intérprete para hacer la consulta.

Se muestran las equivalencias que se podrían utilizar en parte para este traductor:

Abreviación	Equivalente
Eje no especificado	Se da por supuesto el eje child Ejemplo: capítulo/párrafo equivale a : child::capítulo/child::párrafo
@	Es el eje attribute Ejemplo: párrafo[@tipo = “Implementación”] equivale a: child::párrafo[attribute::tipo = “Implementación”]
//	Es el eje descendant-or-self Ejemplo: //párrafo es equivalente a: /descendant-or-self::node()/child::párrafo
.	Equivale a self::node()
..	Equivale a parent::node()

Tabla 5.1. Equivalencias entre sintaxis abreviada y no abreviada en XPath

Por otro lado, después de ver las dificultades para la implementación de los ejes hacia atrás, también se podría implementar una función para reescribir las consultas donde se utilicen ejes hacia atrás en consultas hacia delante. Una vez la función estuviera definida, bastaría con usar el intérprete implementado en este proyecto para dar con los resultados deseados.

Los ejes:

- ancestor
- ancestor-or-self
- preceding
- preceding-sibling
- parent

se pueden reescribir utilizando los ejes hacia delante detallados e implementados en este proyecto.

Un ejemplo muy sencillo de esto es, es una consulta como:

child::*/child::*/child::*/parent::*

es equivalente a escribir: child::*/child::*

Bajar un nivel del árbol XPath para volver a subir un nivel, es equivalente a no dar ninguno de esos dos pasos.

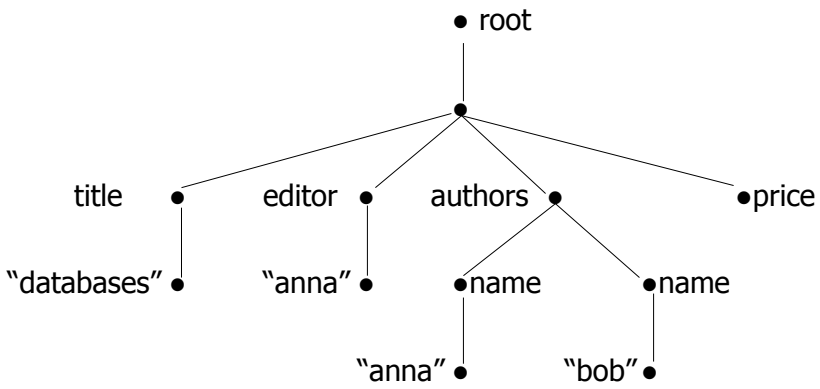
El uso de predicados, como los implementados en el proyecto, constituyen una herramienta útil que facilita esta tarea de "traducción".

Para explicar esto, se propone el siguiente ejemplo:

Sea el siguiente documento:

```
<journal>
      <title>databases</title>
      <editor>anna</editor>
      <authors>
            <name>anna</name>
            <name>bob</name>
      </authors>
      <price />
</journal>
```

Con su representación arbórea mostrada a continuación:



Una consulta donde se piden todos los nombres que aparecen antes del nodo de nombre "price" se escribe usando ejes hacia atrás de la forma:

/descendant::price/preceding::name

Usando predicados, esta consulta se puede reescribir sin necesidad del uso de ejes hacia atrás de la siguiente manera:

/descendant::name[following::price = = /descendant :: price]

Mientras en la primera consulta se seleccionan los nodos "name" anteriores al nodo "price", el camino de localización equivalente seleciona todos los nodos "name" que tienen un nodo "price" como siguiente, si estos nodos resultan ser también nodos descendientes de la raíz. Existe claramente un camino de localización igualmente equivalente y más sencillo:

/descendant::name[following::price]

Este ejemplo está sacado de un estudio hecho por el instituto de informática y centro de información y procesamiento de lenguaje de la Universidad de Munich, Alemania, donde se explica y demuestra esta idea con mucho detalle.[5]

5 Ver bilbiografía [8].
También se puede descargar este trabajo en: http://www2.pms.ifi.lmu.de/publikationen/PMS-FB/PMS-FB-2002-4.pdf

Bibliografía

[1] Sitio oficial de la World Wide Web consortium : *XML Path Language (XPath) 2.0 (www.w3.org/TR/xpath20)*

[2] Sitio oficial de la World Wide Web consortium : *Extensible Markup Language (XML) (www.w3.org/XML)*

[3] G.Martín, I.Martín Benitez: *Curso de XML introducción al lenguaje web,* 2005

[4] J.Brochard: *XML conceptos e implementación,* 2001

[5] A.Watt, D.Marcus, E.Lenz, J.Duckett, J.Pinnock, K.Visco, K.Williams M.Birbeck, N.Ozu, O.G.Gudmundsson, P.Kobak, S.Mohr, S.Livingstone, Z. Zaev : *Professional XML,2nd Edition ,* 2001

[6] P.Walmsley: *Xquery,* 2007

[7] B.C. Ruiz Jimenes... [et al.]: *Razonando con Haskell: un curso sobre programación funcional ,* 2004

[8] D.Olteanu, H.Meuss, T.Furche, F.Bry: *XPath: looking Forward*