Recibido: 17 de agosto 2012; Aceptado: 8 septiembre 2012.

Interoperabilidad de Haskell con otros lenguajes de programación mediante XML

Haskell interoperability with other programming languages using XML

Alvaro Israel Sejas

Universidad Mayor de San Simón, Cochabamba - Bolivia

isoujiro@gmail.com

Resumen: En este artículo se propone un enfoque para la integración de Haskell con lenguajes foráneos, usando XML como mecanismo para expresar llamadas remotas a servicios implementados en Haskell. HermesPC es la solución planteada (e implementada) que consiste de un protocolo y un servidor que permite el acceso remoto a las funciones Haskell. Además se presenta un ejemplo del mundo real (HermesRss) en el que se utiliza la solución implementada.

Palabras clave: Programación funcional, Haskell, .Net, XML, servicio remoto.

Abstract: This paper proposes an approach for the integration of Haskell with foreign programming languages, using XML as a mechanism to compose remote calls to services implemented in Haskell. HermesPC is the proposed (and implemented) solution, it is composed of a protocol and a server which allows remote access to Haskell functions. Moreover, a real world example (HermesRss) that uses the solution is presented.

Keywords: Functional programming, Haskell, .Net, XML, remote service.

1. Introducción

La integración de Haskell con otros lenguajes de programación es un tema que ha sido abordado por varias iniciativas desde diferentes enfoques, cada uno con sus ventajas e inconvenientes.

Varias de las tecnologías Haskell implementadas actualmente tienen su propio enfoque para tratar con el problema de interoperabilidad, pero como se verá más adelante éstas presentan ciertas desventajas, lo cual nos motiva a proponer una solución desde otro punto de vista.

En este artículo se pretende mostrar una propuesta de solución que se apoya en el uso de XML, algo similar a los Servicios Web pero por ahora más simple. XML es la tecnología que se utiliza para el intercambio de información estructurada porque permite lograr independencia de la plataforma con la que se quiere la interoperabilidad.

La idea es trabajar en busca de un protocolo que permita hacer invocaciones remotas a funciones de Haskell y un servidor que provea éstas funciones, lo cual es natural implementar usando las tecnologías de red que provee Haskell en su intérprete Ghc.

El artículo se ha organizado de la siguiente forma: en la sección dos se presenta un resumen de algunas tecnologías que abordan el dominio del problema y presentan diferentes enfoques de solución; en el apartado tres se hace una introducción a HaXml que es una librería para procesamiento de XML cuya funcionalidad es indispensable en este caso; en la sección 4 se presenta una explicación de la solución planteada; y finalmente en la parte 5 se muestra un ejemplo de uso de la solución propuesta en una aplicación de la vida real.

2. Tecnologías Haskell para interoperabilidad con lenguajes foráneos

Debido a la naturaleza actual de algunas aplicaciones, un lenguaje de programación debe poseer algunos mecanismos que le permitan interactuar con otros lenguajes, para así ampliar sus posibles aplicaciones. Haskell posee varias tecnologías que le permiten interactuar con lenguajes foráneos, a continuación se presentan algunas de ellas.

2.1. FFI

La Interfaz de Lenguajes Foráneos de Haskell (FFI o Foreign Function Interface [1]) es una especificación que añade a Haskell la funcionalidad de interactuar con otros lenguajes de programación, es decir que permite llamar desde Haskell a código implementado en un lenguaje diferente y viceversa.

Antes de presentar el objetivo principal de FFI se debe definir dos conceptos: el contexto Haskell es el contexto de ejecución de la máquina abstracta en la cual Haskell está basado; mientras que un contexto externo es otro contexto diferente a este último.

El objetivo principal de FFI es permitir el acceso desde un contexto Haskell a un contexto externo mediante una interfaz programable, es decir el poder acceder a datos y funciones de un contexto externo desde Haskell y lo inverso.

La extensión a la especificación de Haskell para soportar FFI solo agrega las siguientes producciones:

```
reservedid -> foreign specialid -> export | safe | unsafe |
         ccall| cplusplus | dotnet | jvm | stdcall| system-
         specific calling conventions
```

Mediante estas producciones se puede declarar funciones que permiten invocar entidades del contexto externo así como también declarar funciones del contexto Haskell para ser invocadas por el contexto externo.

Por ejemplo, para llamar a la función seno de la librería math.h de C, se debe hacer una declaración `import`, de la siguiente forma:

```
foreign import ccall "math.h sin" seno::CDouble -> CDouble
```

En la declaración: `'ccall'` es la convención de llamada, '`math.h sin`' es el identificador de la entidad externa, `'seno'` es el identificador de la función y `'CDouble'` el tipo utilizado en la función.

En el caso opuesto, para poder hacer que una función implementada en Haskell esté disponible para ser llamada desde otro lenguaje, se debe hacer una declaración `export`, de la siguiente forma:

```
foreign export ccall "restarInt" (-):: Int -> Int -> Int
```

En la declaración, `'ccall'` es la convención de llamada, `'restarInt'` es el identificador de la función, '`(-)`' es lo que será evaluado e `'Int -> Int -> Int'` es la instancia del tipo de la función exportada.

Esto es básicamente lo que provee FFI para que Haskell pueda interactuar con otros lenguajes de programación, el intercambio de información se puede hacer mediante los tipos de datos básicos soportados por FFI.

2.2. Hugs98 para .Net

Hugs98 para .Net [2] es una extensión de Hugs98 que permite la interoperabilidad entre la plataforma .Net y Haskell.

Esta extensión es provista como una implementación del intérprete Hugs98 que agrega algunas producciones al lenguaje, mediante las cuales se puede instanciar y usar objetos .Net desde Haskell y también permite llamar a funciones Haskell desde cualquier lenguaje de la plataforma .Net.

La motivación de esta implementación es completamente pragmática, el intérprete Hugs98 opera lado a lado con el .Net runtime, lo que permite que segmentos de código de un lenguaje hagan invocaciones a segmentos de código del otro lenguaje, es decir que no es un compilador que transforma de código Haskell a código intermedio de .Net (MSIL[1])

De manera similar a FFI, Hugs98 para .Net accede a un contexto externo mediante producciones que agrega a la especificación de Haskell, las cuales son:

```
ffidecl : ...
         | 'foreign' 'import' 'dotnet'
         "spec-string" varName '::' ffiType
```

[1] Microsoft Intermediate Language, lenguaje intermedio de la plataforma .Net

```
spec-string : ('static')?
         ('field'|'ctor'|'method')? ('[' assemblyName ']')?
         .NETName

ffiType : PrimType -> ffiType
         | IO PrimType
         | PrimType

PrimType = standard FFI types + Object a + String
```

Las producciones `'foreign'` e `'import'` tienen el mismo objetivo que en la especificación FFI, `'export'` no está implementada en Hugs98.Net, y `'dotnet'` es el identificador de la convención de llamada.

Las palabras reservadas listadas en `'spec-string'` hacen referencia a la entidad que se quiere llamar, es decir: `'static'` para una entidad estática, `'field'` para un atributo de un objeto, `'ctor'` es el constructor de objetos, `'method'` para un método; `'assemblyName'` y `'.NETName'` para hacer referencia a una clase de .Net.

Por ejemplo una llamada a un método estático tendría la siguiente forma:

```
foreign import dotnet "static System.Console.Write"

saludar :: String -> IO String
```

El tipo de la función que se quiere importar tiene que ser compatible con el tipo de .Net correspondiente, de lo contrario ocurre una excepción IO.

Las llamadas a funciones Haskell se realizan mediante una librería (Hugs Server API) que ofrece un API para acceder a algunas funcionalidades del intérprete Hugs98 para .Net, pero esto es sólo posible partiendo desde el lado Haskell (enviando un referencia de una instancia del intérprete al programa .Net) lo cual es un serio inconveniente.

Finalmente podemos mencionar que se puede también emular herencia y hacer llamadas con estilo orientado a objetos; además que Hugs98.Net contiene una herramienta que permite la generación automática de declaraciones FFI y tipos de objetos para clases .Net llamada hswrapgen.

2.3. H/Direct

H/Direct[3] es una interfaz de lenguaje foráneo para Haskell, que en lugar de confiar en tipos específicos a lenguajes de programación, compila IDL (Interface Definition Language) a código que transforma los datos a través de la interfaz. Permite así a Haskell llamar a código implementado en C o empaquetado como COM (Component Object Model); y también a los componentes Haskell ser envueltos en C o COM para ser llamados desde otros lenguajes.

Para el caso de interoperabilidad entre dos lenguajes, definir el tipo de un procedimiento en una notación formal es lo más apropiado. Basándose en esa notación H/Direct genera código de tubería que convierte los parámetros para que puedan ser comprendidos por el otro lenguaje, entonces llama al procedimiento extranjero usando la convención de llamada del mencionado lenguaje y después vuelve a convertir los resultados para que puedan ser entendidos nuevamente.

Para resolver el problema de la conversión de tipos, H/Direct usa IDL, porque no es un lenguaje de programación sino un lenguaje de especificación, este le permite definir tanto los tipos de las funciones como también los datos que toma como argumentos, todo de una manera independiente de la arquitectura y del lenguaje.

H/Direct permite las siguientes operaciones:

- Acceder a librerías externas
- Exportar funciones Haskell al mundo exterior
- Usar objetos COM desde Haskell
- Implementar componentes COM con Haskell

Para implementar un componente COM con Haskell y H/Direct se requiere lo siguiente:

- La aplicación implementada en Haskell
- La especificación IDL para las funciones que se quiere acceder desde el exterior
- El código de tubería que H/Direct genera automáticamente
- La librería COM que exporta las funciones de Haskell, y una librería en C que provee algún código de soporte de tiempo de corrida

Para comprender mejor lo que es H/Direct se presenta un ejemplo [4][5]. En el que se empaqueta una función de Haskell como un componente COM, en la forma de una librería DLL, para luego poder ser usada desde otros programas.

Se parte definiendo la estructura de la función en IDL (archivo Hola.idl) e implementando algunas otras definiciones requeridas:

```
...
[ object, uuid(2cf00d46-4f67-11dc-8314-0800200c9a66)]
interface Ihola : IUnknown {HRESULT saludar([out,retval] BSTR
         *out); };
[ object, uuid(4df11530-4f67-11dc-8314-0800200c9a66)]
coclass Hola { [default]interface Ihola; };
...
```

El anterior listado constituye la definición de la función `'saludar'`, que lo único que hace es retornar una cadena.

A continuación, usando una de las herramientas de H/Direct se generan dos archivos, el proxy que implementará una interfaz COM mediante la cual se podrá acceder al componente y también se genera el esqueleto de la función:

```
$ ihc -fcom Hola.idl -s --skeleton
```

Se debe completar código en el archivo con el esqueleto de la función y después compilar los dos archivos generados:

```
$ ghc -c -ffi -package com Hola.hs HolaProxy.hs
```

Para generar una librería DLL se debe crear un módulo Main, copiar los archivos ComDllMain.lhs y dllStub.c al directorio del ejemplo y compilarlos junto con el módulo Main:

```
$ ghc -c -ffi -package com Main.hs ComDllMain.lhs dllStub.c
```

Finalmente se copia el archivo ComServer.def y se procede a generar la librería DLL:

```
$ ghc --mk-dll -o comdll.dll -optdll-def -optdllComServer.def
         Hola.o HolaProxy. o Main.o ComDllMain.o dllStub.o -
         fglasgow-exts -syslib com
```

La librería DLL generada puede ser usada desde cualquier lenguaje que soporte esta tecnología.

2.4 Comparativa

Cada una de las soluciones expuestas presenta ventajas y desventajas en cuanto a su forma de tratar con el problema de interacción de Haskell con lenguajes foráneos.

FFI está implementada tanto por Ghc como por Hugs, lo cual la hace bastante accesible, pero la implementación actual en Ghc solo soporta llamadas foráneas a lenguajes con la convención de llamada de C. La implementación de FFI (en Ghc) requiere que además se deba implementar la convención de llamada del lenguaje con el que se quiere usar (cuando este es diferente de C), lo cual hace de esta interfaz muy dependiente de la plataforma, esto es una debilidad para el desarrollo con esta tecnología.

Hug98 para .Net implementa la mitad de la funcionalidad de FFI, y si bien posee de un mecanismo para llamar a funciones Haskell desde .Net, todo este mecanismo fue implementado solo para probar conceptos, no está orientado a ser un producto que pueda usarse en producción, no se cuenta con soporte en caso de encontrar algún problema, lo cual es un gran inconveniente al momento de implementar aplicaciones del mundo real y querer usarlo.

H/Direct tiene uno de los mejores enfoques, el uso de IDL lo hace una solución bastante completa, pero una de sus debilidades es el alto grado de conocimiento de la tecnología COM y lenguaje C que se necesita para poder utilizarlo, además de su escasa documentación y soporte. Por otro lado, al encapsular un programa de Haskell en una librería DLL se le quita cierto grado de independencia al resultado de H/Direct, puesto que no todas las implementaciones de los lenguajes de programación soportan el uso de librerías DLL.

Para tener una visión más clara al respecto, se presenta la siguiente tabla comparativa:

Tabla 1: Comparativa de herramientas para interacción de Haskell con lenguajes foráneos.

Criterio	FFI	Hug98.Net	H/Direct
Característica	Extensión a Haskell	Extensión a Haskell	IDL
Es estándar?	SI	NO	SI
Interactúa con	C (Se puede extender)	.NET	C, COM
Complejidad de uso	Medio	Medio	Alto
Soporte	Alto	Muy Bajo	Bajo

3. HaXml: Tecnología Haskell para XML

HaXml [8] es una librería implementada en Haskell que presenta dos formas de procesar XML en un lenguaje funcional.

La primera aprovecha la estructura de árbol genérico de XML y la usa como base para crear un conjunto de filtros y combinadores que permiten el procesamiento de su contenido. La segunda consiste en transformar el contenido XML en un dato de Haskell y trabajar sobre este tipo de dato algebraico.

3.1. Los combinadores genéricos

Un documento XML es esencialmente un árbol, tiene dos tipos: un elemento (encerrado en tags), y texto plano. El elemento tiene un tag de inicio y un tag de terminación, además el tag de inicio puede contener atributos en forma de pares nombre y valor. Entonces se puede representar un documento XML usando el siguiente tipo de dato:

```
data Element = Elem Name [Attribute] [Content]

data Content = CElem Element
             | CText String
```

Los lenguajes funcionales son muy buenos para procesar XML debido a su naturalidad para procesar tipos de datos en forma de árbol.

El tipo `'content filter'` toma un fragmento del contenido XML y retorna una secuencia de contenido.

```
type CFilter = Content -> [Content]
```

En base a este tipo se puede definir filtros básicos como filtros para selección, para predicados, y para construcción de XML. por ejemplo podemos mencionar `'elm'` y `'txt'`, el primero que sirve para filtrar el contenido XML siempre que sea un elemento, y el segundo que filtra un texto plano como se observa en la siguiente definición:

```
elm, es un elemento encerrado en tags?

txt, es texto?
        :: CFilter
```

Como pegamento para utilizar los filtros se definen combinadores, éstos son operadores de alto orden que sirven para componer funciones y construir a su vez combinadores más poderosos, por ejemplo tenemos `'o'` que es la composición irlandesa que sirve para conectar dos filtros entre si, el filtro de la izquierda es aplicado a los resultados del filtro de la derecha, a continuación se presenta éste combinador y otro de búsqueda recursiva:

`o`, composición irlandesa

```
:: CFilter -> CFilter -> CFilter
```

`deep`, búsqueda recursiva (el de más arriba)

```
:: CFilter -> CFilter
```

Usando los combinadores se obtiene una forma de expresión más natural para el dominio de un problema, en este caso los combinadores de HaXml dotan a Haskell de mayor expresividad para el procesamiento de contenido XML.

3.2. Transformación a tipos de datos Haskell

Existe una correspondencia entre el lenguaje que usado por DTDs para definir las reglas de XML y los tipos de datos de Haskell.

Para comprender mejor la forma en que trabaja la transformación se presenta un ejemplo de DTD utilizado para definir las reglas que debe cumplir un contenido XML que define una estructura 'persona' con sus 'roles':

```
<!DOCTYPE Persona
         [<!ELEMENT Persona (nombre, (rol*)) >
         <!ELEMENT nombre (#PCDATA) >
         <!ELEMENT rol (#PCDATA) >
              <!ATTLIST parameter nombrerol CDATA #REQUIRED>]>
```

Este DTD se transforma en el siguiente tipo de dato algebraico de Haskell mediante la herramienta DtdToHaskell de HaXml:

```
...
data Persona = Persona Nombre ([Rol])
         deriving (Eq,Show)

newtype Nombre = Nombre String
         deriving (Eq,Show)

newtype Rol = Rol String
         deriving (Eq,Show)

data Parameter = Parameter { parameterNombrerol :: String }
         deriving (Eq,Show)

...
```

Se puede hacer la lectura del contenido XML y convertir el mismo en el tipo de dato Haskell usando la función 'readXml' de HaXml.

```
leerPersona:: String -> Maybe Persona

leerPersona xmlMessage= readXml xmlMessage
```

Este enfoque de HaXml va más allá de la simple comprobación de contenido XML bien formado, da lugar al concepto de procesamiento válido de documentos, es decir un script que dado un documento válido como entrada devuelve otro documento válido como resultado. Como la correspondencia entre el DTD y el tipo de dato algebraico Haskell (generado a partir de éste) es directa, el verificar que el contenido de un documento es válido como también el script que lo procesa, se reduce a inferencia de tipos de Haskell.

4. HermesPC

Si bien las tecnologías revisadas con anterioridad en este artículo tienen como objetivo la interacción de Haskell con otros lenguajes de programación, algunas de ellas tienen ciertos problemas y otras ciertas debilidades.

La solución propuesta a este problema de interoperabilidad es dividida en dos partes:

1. Un protocolo basado en XML para la invocación remota de funciones implementadas en Haskell (HermesProtocol).
2. Un servidor de funciones Haskell que provea la ejecución de las funciones procesando/respondiendo a las peticiones hechas mediante HermesProtocol (HermesService).

4.1. Protocolo para invocación de funciones Haskell: HermesProtocol

Es necesario proveernos de un mecanismo que nos permita invocar a las funciones implementadas en Haskell, para esto definimos HermesProtocol, un protocolo basado en XML.

El comportamiento que se quiere soportar es hacer peticiones de funciones sin saber acerca de la operación requerida o del que hace el requerimiento. Ésta es la motivación del patrón de diseño 'comando'[7] del mundo Orientado a Objetos. Este patrón permite encapsular un requerimiento como un objeto, permitiendo de esta manera tener clientes que expresen sus diferentes requerimientos mediante parámetros.

Apoyándose en HaXML es muy fácil implementar el protocolo, es tan simple como definir el DTD con la estructura y generar el tipo de dato Haskell que soportará el contenido XML.

El primer paso es definir la estructura de nuestro HermesProtocol:

```
<!DOCTYPE HermesMessage
         [<!ELEMENT HermesMessage (key, operation,
         (parameter*|result*)) >
         <!ELEMENT key (#PCDATA) >
         <!ELEMENT operation (#PCDATA) >
         <!ELEMENT parameter (#PCDATA) >
              <!ATTLIST parameter name CDATA #REQUIRED>
         <!ELEMENT result (#PCDATA) >
              <!ATTLIST result name CDATA #REQUIRED>]>
```

`HermesMessage` representa un mensaje que expresa la invocación de una función Haskell o el resultado de dicha invocación. El elemento `'key'` provee un mecanismo de seguridad (una llave encriptada en MD5 y codificada en Base64) que permite validar el origen del mensaje, `'operation'` es el identificador de la funcionalidad requerida, y dependiendo de si es un requerimiento o un resultado tenemos los parámetros (`'parameter'`) o los resultados (`'result'`).

Seguidamente debemos generar la estructura de nuestro protocolo como un tipo de dato algebraico de Haskell:

```
...
data HermesMessage = HermesMessage Key Operation
         (OneOf2 [Parameter] [Result])
         deriving (Eq,Show)

newtype Key = Key String
         deriving (Eq,Show)

newtype Operation= Operation String
         deriving (Eq,Show)

data Parameter =Parameter Parameter_Attrs String
         deriving (Eq,Show)

data Parameter_Attrs = Parameter_Attrs{ parameterName ::
         String }
         deriving (Eq,Show)

data Result = Result Result_Attrs String
         deriving (Eq,Show)

data Result_Attrs = Result_Attrs { resultName :: String}
         deriving (Eq,Show)

...
```

Se eligió este enfoque de procesamiento de XML de HaXml (en lugar de usar los combinadores genéricos), debido a que un HermesMessage siempre debe llegar completo (caso contrario no sería un mensaje correcto), lo cual nos obliga a obtener un documento válido como entrada para poder generar otro documento válido como salida (procesamiento válido de documentos).

Un ejemplo de un mensaje de HermesProtocol se muestra en el siguiente listado:

```
<HermesMessage>

<key>2sxdCynQpn0+YshcbE8XnQ==</key>

<operation>VERIFY_RSS</operation>

<parameter name="FEED">PD94bWwgdmVyc2lvbj0i...</parameter>

</HermesMessage>
```

4.2. Servicios en Haskell: HermesService

Un enfoque muy apropiado para el servidor de funciones Haskell es el uso de programación de redes, la comunicación mediante protocolos de red es

independiente del lenguaje de programación, solo se necesita dos puntos que puedan comunicarse entre si usando un protocolo de red para poder interactuar. Pero los protocolos de red (como TCP o UDP) están orientados simplemente a la transmisión correcta de datos planos, razón por la cual se tiene que apalancar este mecanismo de manera que se pueda transmitir requerimientos de invocación de funciones, para eso usamos el HermesProtocol definido en la anterior sección.

Se puede observar el comportamiento de HermesService en la Figura 1.

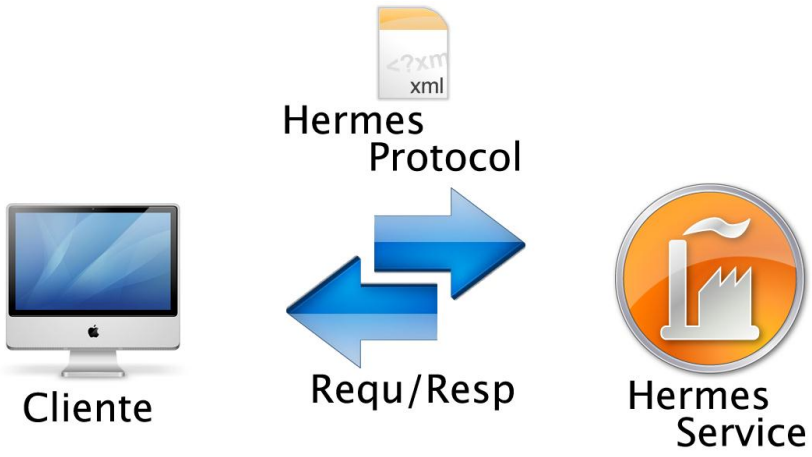

Figura 1:HermesService

Dado que ya tenemos el tipo de dato algebraico de Haskell que define la estructura de HermesProtocol, convertir el contenido XML al tipo de dato se hace en unas pocas líneas de código:

```
import Text.XML.HaXml.XML2Haskell import Maybe

...

parseHermesMessage:: String -> Maybe HermesMessage

parseHermesMessage xmlMessage= readXml xmlMessage

...
```

El mensaje se lo recibe mediante una conexión de red, esto creando un socket usando la librería Network de Ghc.

```
startHermesService=forever (startService) startService=
        do{(msg, socket, handle) <- listenMessage portNumber;
        forkIO(processNetworkMessage msg socket handle);}

where portNumber=5760
```

La función '`forkIO`' del módulo `Concurrent` de Ghc permite iniciar un nuevo hilo que procesa el requerimiento, retorna el resultado (o un error si se diera el caso) y termina. Además que '`forever`' proporciona un ciclo infinito que constantemente acepta los requerimientos entrantes.

```
result <- catch (processMessage (parseHermesMessage message))
         (return(createError "ERROR"));
```

El resultado se lo retorna también empaquetado en HermesProtocol (en la sección `'result'`), lo mismo que los posibles errores.

Toda esta lógica es muy similar a la de los Servicios Web con SOAP, claro que mucho más simple y liviana.

La implementación realizada, permite recibir requerimientos de funciones Haskell a través de un Socket de red, seguidamente buscar la función referida por un identificador (`operation`), ejecutar esta función pasándole como parámetros los contenidos en el mensaje y finalmente retornar un resultado también empaquetado en el protocolo.

La definición y consistencia de tipos de datos de los parámetros son responsabilidad del desarrollador, aunque el mecanismo debería extenderse en un futuro para poder hacer este manejo de tipos de manera automática.

5. Ejemplo: usando HermesPC desde .Net

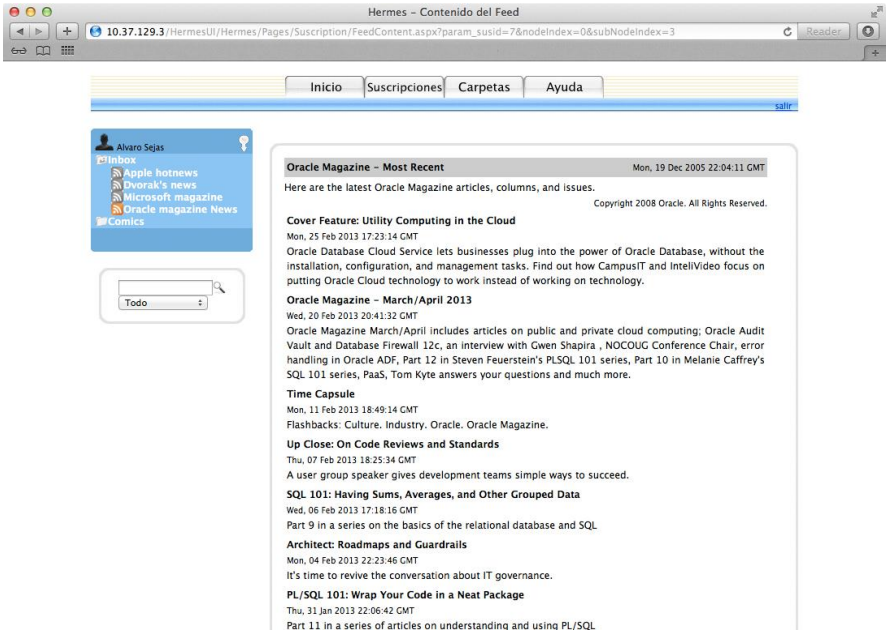

Figura 2: Vista de HermesRSS

Para experimentar con la solución propuesta (e implementada), se desarrolló una aplicación de complejidad promedio con requerimientos de usuarios reales.

HermesRss (nombre código) es una aplicación web implementada con las tecnologías ASP.NET (en C#) y NHibernate, que tiene como objetivo

proporcionar las funciones de un agregador de formatos de sindicación, es decir que permite a un usuario monitorear fuentes de redifusión (en formatos rss y atom) de su interés.

La aplicación implementa componentes que presentan funcionalidades requeridas por un sistema de este tipo, tales como administración de usuarios, manejo de fuentes de redifusión, administración de carpetas, manejo de suscripciones y búsquedas en las mismas. En la Figura 3 se puede apreciar los componentes de la lógica del negocio de la aplicación.

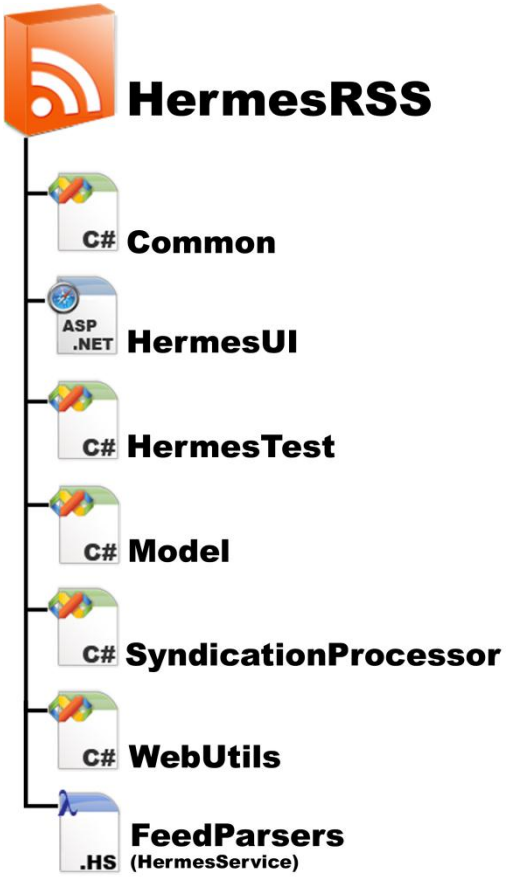

Figura 3: Diagrama de componentes de HermesRss

La solución de servicios Haskell, simplemente provee los servicios de traducción de contenido XML en formatos rss y atom a formato HTML. Esta funcionalidad es implementada con la ayuda de HaXml que permite esta transformación de XML a HTML de manera liviana usando los filtros y combinadores genéricos y los combinadores para generación de HTML que tiene incluidos.

HermesRss está implementado con un enfoque orientado a componentes, debido a lo cual se implementó un componente Procesador de fuentes de redifusión, el cual contiene un cliente de HermesService. Este cliente se encarga de componer los mensajes que invocan las funciones Haskell (funciones de traducción de formatos de sindicación); luego envía los mensajes y procesa el resultado.

Todo esto se hace tomando consideraciones de seguridad pertinentes, es decir se genera una llave encriptada que luego debe ser verificada en el lado Haskell y viceversa, se puede ver esto en el siguiente listado que contiene código C#:

```
string encryptedKey = generateKey("..."); //genera llave

Communications comm=new Communications();

comm.startConnection(server,portNumber);

string message=createHermesServiceMessage(encryptedKey,
         operation, parameters); //compone el mensaje

res=comm.transmitMessage(message+"\n"); //invoca función
         Haskell

...

res = comm.readMessage(); //espera resultados

...

comm.closeConnection();
```

El procedimiento para consumir un servicio Haskell es bastante sencillo, primero se debe generar la llave encriptada, luego se inicia la comunicación con HermesService (mediante un Socket sobre TCP), se compone el mensaje (con la operación requerida y los parámetros necesarios) y se envía el mensaje. A continuación se espera, se procesa los resultados y se cierra la comunicación.

Este comportamiento es posible desde cualquier lenguaje de programación que soporte comunicación de redes y procesamiento de contenido XML, este es uno de los mayores beneficios de HermesPC, de hecho el HermesService jamás se entera con qué lenguaje o plataforma está interactuando.

6. Conclusión

XML es un lenguaje muy potente para representar información estructurada, además que es una especificación, es decir que no está ligada a ninguna plataforma en particular, sino que puede ser procesado desde cualquier lenguaje de programación que tenga capacidades básicas (es prácticamente una funcionalidad por defecto en los lenguajes de programación actuales).

El uso de XML también permite aprovechar las capacidades naturales de Haskell para procesar información estructurada en forma de árboles con HaXml.

Lo anterior hace que se opte por utilizarlo para diseñar un protocolo que sirva para transmitir requerimientos de invocación de funciones y procesar sus resultados, esto a través de tecnología de red, lo cual mantiene consistencia con la

intención de no ligar la solución a algún lenguaje de programación o plataforma en particular.

HermesProtocol, si bien es un protocolo bastante simple, cuenta con todo el potencial para poder realizar las invocaciones remotas a funciones de Haskell de una manera bastante transparente a la plataforma, además que es bastante simple de usar al momento de desarrollar y no crea complejidad innecesaria.

HermesService por su parte, hace un procesamiento de HermesProtocol bastante natural, debido al gran soporte que presta HaXml y todas las funcionalidades provistas por los módulos de Ghc.

A continuación se presenta una tabla comparativa que muestras algunas diferencias con las soluciones investigadas con anterioridad:

Tabla 2: Comparativa de las soluciones estudiadas con HermesPC

Criterio	FFI	Hug98.Net	H/Direct	HermesPC
Es estándar?	SI	NO	SI	NO, pero usa XML
Interactúa con	C (se puede extender)	.NET	C, COM	Cualquier lenguaje que soporte XML y TCP
Complejidad de uso	Medio	Medio	Alto	Medio
Diseñado para llamadas remotas?	NO	NO	NO	SI

El software en el que se puso a prueba la solución propuesta es una aplicación web de la vida real, un sistema pensado para usuarios reales que requieren de soluciones a sus requerimientos, este caso de uso pone a prueba el comportamiento de la solución planteada, y da resultados satisfactorios (se pudieron implementar las funcionalidades requeridas mediante HermesPC).

Finalmente esto es un ejemplo del gran potencial que tiene Haskell dentro del desarrollo de aplicaciones del mundo real, y desmitifica de cierta manera su alcance, demostrando que puede ser empleado para resolver problemas que se hallan dentro un amplio dominio de aplicaciones, además que de esta manera se quiere animar a más desarrolladores a poder probar los beneficios de la Programación Funcional.

Referencias

[1] Chakravarty M. et al The Haskell 98 Foreign Function Interface 1.0 An Addendum to the Haskell 98 Report. 2002..2003.

[2] Finne S. Hugs98 for .Net. 2002-2003.

[3] Finne S. et al H/Direct: A Binary Foreign Language Interface for Haskell. 1998.

[4] Finne S. HaskellDirect User's Manual. 1999.

[5] Alarcón B., Lucas S. Integración de componentes Haskell en .NET. 2005.

[6] Alarcón B., Lucas S. Building .NET GUIs for Haskell applications. 2006.

[7] Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns - Elements of reusable Object-Oriented Software. Addison-Wesley, 1995.

[8] Wallace M., Runciman C. Haskell and XML: Generic Combinators or Type-Based Translation?. 1999.

[9] Peyton Jones S. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. 2005.