A Web Services Proxy Generator for Haskell

André W. B. Furtado, Adeline de S. Silva, Carla M. P. do Nascimento, Gustavo A. dos Santos, Carlos A. G. Ferraz, André L. de M. Santos

Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)

Av. Professor Luís Freire, s/n, Cidade Universitária, CEP 50740-540, Recife/PE/Brazil
+55 (81) 21268430

{awbf,adss,cmpn,gas,cagf,alms}@cin.ufpe.br

ABSTRACT

This paper presents the implementation of a web services proxy generator for the Haskell functional language. Its purpose is to free Haskell programmers from low-level implementation details when building applications that consume services offered through the Web. The generator architecture, as well as implementation experiences resulting from the specific web services access/proffer model, is detailed. The final purpose is to show that Haskell and functional languages in general can be effectively used to implement distributed components and applications, interacting with services implemented in different languages and/or platforms.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability.

General Terms

Standardization, Languages.

Keywords

web services, proxy, Haskell, SOA (service-oriented architecture), interoperability

1. INTRODUCTION

Although relatively recent, the web services technology is considered to be one of the most important topics regarding distributed systems and information technology in general. By allowing the integration and interoperability of systems developed under different platforms and languages, web services became a concrete alternative to the development of distributed and connected applications. As a consequence, software industry is giving special attention to the subject, as it can be verified by the implementation and maintenance of web services support in major software development platforms, such as Microsoft .NET [10] and J2EE [15].

On the other hand, it can be noticed that the opportunities made available by web services are still under-explored in relation to functional languages. Although this type of programming languages incorporates some of the benefits of object-oriented programming [17], today's most popular programming paradigm [12], as well as presents some advantages in relation to other paradigms (conciseness, elegance, higher abstraction and lazy evaluation, among others [17]), it is recognized that functional languages also present some limitations. The isolation of functional languages regarding integration with other languages and technologies is certainly one of the factors that limit their adoption as a solution to problems belonging to various application domains [21]. Although some efforts are being taken in order to overcome this deficiency, such as the integration of functional languages with COM [14], Java [9], .NET [7] and OpenGL [13], it is noticeable the absence of projects intending to integrate web services with functional languages.

We believe, therefore, that the ubiquitous presence of recent internet technologies aimed at interoperability, such as XML, SOAP and web services, are major keys to situate functional languages, more specifically Haskell [5], as a viable alterative to the implementation of distributed applications and components. The approach presented in this paper consists of the creation of a web services proxy generator for Haskell, allowing applications implemented in this language to use services implemented in other languages/platforms and which are available in the internet through the web services technology. By detailing the implementation experience gathered during the development of the proxy generator, this paper also represents a contribution to other projects with focus on simplifying web services consumption for other languages[1].

The next sections of this paper are organized as follows. Section 2 presents an introduction to key web services concepts. Section 3 explains how to use the proxy generator from a Haskell programmer point of view. Section 4 details our implementation approach. Section 5, finally, presents some conclusions about the work and points out future directions.

2. UNDERSTANDING WEB SERVICES

Web services incorporate advantages of two previous technologies: web communication standards and application servers, which use the concepts of service-oriented computing. Evolving from these principles, web services technology uses a set of open standards as its foundation, such as XML, SOAP, WSDL and UDDI, which are maintained by the World Wide Web Consortium (W3C) [20]. These four key concepts are briefly described below:

- XML (*eXtensible Markup Language*) is a language used to structure data and consists of the main web services foundation, since it allows the organization of data used in message exchanges between applications;

[1] The implemented proxy generator, as well as project details, can be found in http://www.cin.ufpe.br/~haskell/hwsproxygen.

- WSDL (*Web Services Description Language*) is an XML-based language which describes the services proffered by a web service;
- SOAP (*Simple Object Access Protocol*) is the transport protocol used to exchange data between the web service provider and the web service client;
- UDDI (*Universal Description and Discovery of Information*) is a catalogue through which web services can be published and searched for.

Figure 1 shows how each one of these technologies relates to each other: when a provider wishes to make a web service available to clients, it builds the web service description (through WSDL) and, optionally, registers the service in a UDDI catalogue. The catalogue, hence, maintains references to the WSDL description and to the service. When a client wants to use a service, it queries the UDDI catalogue by using different filters in order to find a web service compliant to its necessities, obtaining the service description in WSDL and the service access point. A proxy to interact with the web service can also be built in the client at this moment. The proxy uses the WSDL description, finally, to build a SOAP message through which it will communicate with the service provider and effectively consume the service. If the client already knows the URL of the web service WSDL description, which is always the case assumed in this paper, the UDDI catalogue query stage can be ignored.

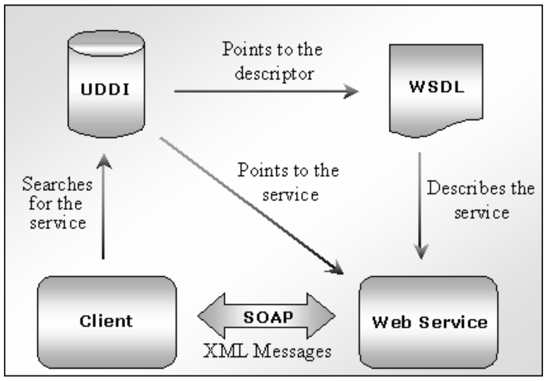

Figure 1. Key web service components.

According to the above description, it can be noticed that, from WSDL specifications and SOAP-enabled communication processes, in theory it is possible to make the web services technology available to any programming language, with the aid of two important elements: a proxy, that hides low-level web service usage details; and a SOAP communication module, which allows message exchange using this protocol.

3. USING THE GENERATOR

The implemented web services proxy generator consists of a command line executable, named *hwsproxygen.exe*. The Haskell programmer must invoke it passing as parameters the URL of the web service description (WSDL file) and the output folder in which the proxy will be generated. For example, in order to generate, in the current folder ("."), a proxy for a web service named *Calc*, which is hosted in a machine named *MySrv*, something similar to the following command line would be used:

```
hwsproxygen.exe http://MySrv/Calc/Calc.asmx?WSDL .
```

As output, the proxy generator creates a series of Haskell modules. One of these modules exports the proxy interface and must be imported by Haskell programs that will use (consume) the web service. In the above example, this important module would be named `Calc` and would be defined in the file *Calc.hs*. The other generated Haskell modules implement the HTTP communication with the web service server and necessary marshalling/unmarshalling [2] operations. The Haskell programmer will not need to use them directly; they will be used by the proxy.

Figure 2 illustrates the relationship between the Haskell modules of a Haskell application under development which uses the proxy generator. In this scenario, the proxy generator was invoked three times, for hypothetical web services named *Bank*, *Timer* and *Calculator*. After the proxies generation, functions exported by the generated modules *Bank.hs*, *Timer.hs* and *Calculator.hs* can be used by the Haskell programmer in order to consume the services (web methods) proffered by those web services.

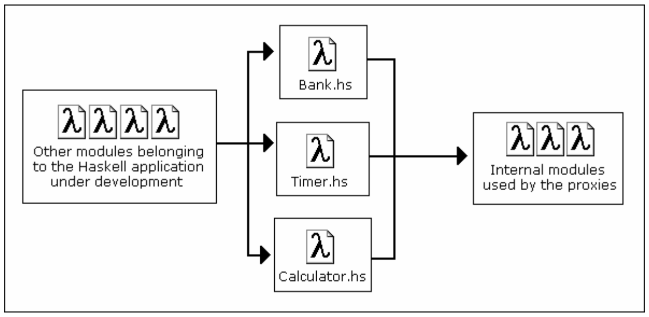
λλλλ
Other modules belonging to the Haskell application under development
λ
Bank.hs
λ
Timer.hs
λ
Calculator.hs
λλλ
Internal modules used by the proxies

Figure 2. Haskell modules after proxy generation

Suppose the *Bank* web service proffers a service named `getAccount`, which receives a *string* value as input (corresponding to the identifier of the bank account) and returns a bank account as output. In order for the Haskell programmer to consume the `getAccount` service, she only needs to import the generated Haskell module `Bank` and call the function `getAccount`, as shown in Figure 3. Although this function invokes an Internet hosted service, the Haskell programmer consumes it as if it were defined locally and does not have to implement any network communication code.

```
import Bank

main :: IO()
main  =  do
  z <- getAcount "XYZ03"
  putStrLn ("Account XYZ03: " ++ show z)
```

Figure 3. Example of web service consumption code, implemented by the Haskell programmer.

4. IMPLEMENTATION APPROACH

Once understood what happens after the invocation of the proxy generator and how to consume its generated modules, in this section we present its internal processing logic and architecture. Basically, the execution flow of a generator session can be divided in two distinct phases:

- An intermediate code structure is built from the web service WSDL description;

- This intermediate code structure is converted into Haskell code that will be finally consumed by Haskell programmers.

The generator was implemented using Microsoft Visual C# [11], one of the .NET Platform languages. This choice was done because .NET already contains an API for dealing with WSDL and other web service concepts. However, the implementation stage of the proxy generator was not restricted to this language, since it was necessary to code, in Haskell, the generated modules to be used by the programmer and internally by the generated proxy.

The intermediate structure, implemented by a class named `ProxyModel`, provides a mapping between the web service WSDL description into programming language elements (such as functions, data types, etc.). This class, as well as other classes of the generator architecture, is presented in the generator Architecture Diagram (Figure 4). The `Orchestration` class is responsible for obtaining, from the network, the XML structure corresponding to the WSDL description. This class then builds a `ProxyModel` and invokes the method `CreateModule` of the `Generator` class. This method will map the intermediate code structure into Haskell code, as well as create the output files. The type hierarchy which has `WSType` as its root, finally, is used by the intermediate code structure to represent the different data types exposed by web services (such as enumerations, lists or primitive types, for example).

The following subsections present specific details about the proxy generator implementation.

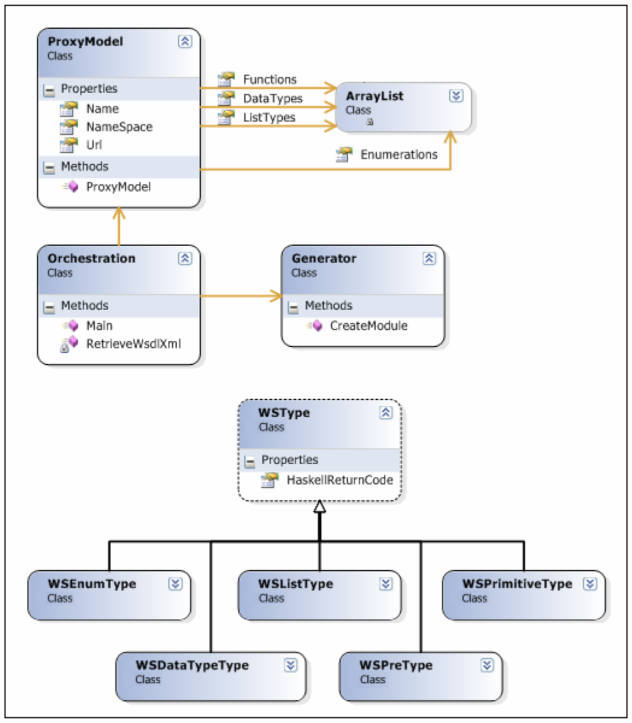

Figure 4. Architecture Diagram of the proxy generator.

4.1 HTTP Communication through Haskell

The implementation of the communication between a Haskell client application and a web server that hosts web services was based on the `Browse` and `HTTP` Haskell modules, implemented by Bringert & Gray [4]. These modules allow the creation of HTTP packages and their dispatch to a web server, making available to the clients the response. The communication is implemented using sockets, accessible in Haskell by a standard module named `Network`.

Since the `Browse` and `HTTP` modules do not support the SOAP protocol, it was included in the scope of this work the extension of these modules in order to enhance them with SOAP support. Fortunately, this extension was very intuitive and localized, not being a major challenge. It was only necessary to add a new HTTP header type, named *SOAPAction*, to the HTTP package definition, as well as to implement a `buildSoapRequest` function that builds parameterized SOAP requests:

```
buildSoapRequest :: URI -> String -> String -> Request
```

Both the first parameter (`URI`) of the function as well as its return type (`Request`) are Haskell data types [18]. The former is defined in the `Network` module, and the latter in the `HTTP` Module. The other parameters are *strings* that correspond, respectively, to the value to be assigned to the *SOAPAction* header of the HTTP package and to the value of the HTTP package body, which must comply with a specific XML/SOAP format.

A Haskell module named `SoapHttpClientProtocol`, one of the most important generated modules, uses the `Browse` and `HTTP` modules. It implements the HTTP package creation logic, according to the SOAP specification, and performs requests to the web server, properly extracting the response. The main function of `SoapHttpClientProtocol` is `invokeWS`:

```
invokeWS
    :: String       -- web service address
    -> String       -- web method to be invoked
    -> String       -- SOAPAction header content
    -> String       -- web service namespace
    -> [Parameter]  -- web service parameters
    -> String       -- xml node containing response
    -> IO String    -- response
```

The `Parameter` type is a synonym type, corresponding to a *string* tuple containing the name and the value of a web method parameter. The name of the parameter is necessary because it is used to build the XML node responsible for encapsulating the parameter value.

The parameters of `invokeWS` were defined as a consequence of what is expected to the creation of a HTTP package which uses the SOAP protocol and invokes a web service. It is worth noticing that `invokeWS` is an internal method, used only by the generated proxies. More high-level methods are available to the Haskell programmer, as discussed in subsection 4.4.

4.2 WSDL/Haskell Mapping

WSDL documents, which describe the services proffered by a web service, are created in XML. Therefore, it was necessary to develop a mapping between the possible contents of the WSDL XML and the target language in which the generated proxy is implemented. This subsection presents some of the mappings between the XML format of WSDL descriptions into Haskell. This mapping is facilitated by the `ProxyModel` intermediate structure, which was previously presented and will be ignored in this section for simplification purposes.

Mapping the W3C specified primitive types used by web services into Haskell did not present much complexity, since Haskell contains the definition of the most common primitive types used by programming languages in general. Therefore, W3C's `int` type was mapped into Haskell's `Int` type and so on. An exception was W3C's `long` type: since Haskell does not contain a definition for this type, the Haskell type `Integer` was used instead.

Besides primitive types, web services also support user-defined types, receiving them as parameters or returning them as result. Enumerations, exemplified by Figure 5, are one example. The value of a variable belonging to an enumeration type can assume only one of the enumeration values in a given moment.

```
<s:simpleType name="Orientation">
    <s:restriction base="s:string">
        <s:enumeration value="North"/>
        <s:enumeration value="South"/>
        <s:enumeration value="East"/>
        <s:enumeration value="West"/>
    </s:restriction>
</s:simpleType>
```

Figure 5. Enumeration sample exposed in WSDL.

The chosen approach for mapping enumerations defined in WSDL into Haskell consists of Haskell data types with multiple constructors, as shown in Figure 6. It is worth noticing that the created type derives from Haskell `Read` and `Show` classes. This is necessary in order to easily convert values of this type to/from *strings*, which is needed when generating and reading the XML body of requests and responses in HTTP (SOAP) packages.

```
data Orientation  = North
                  | South
                  | East
                  | West
                  deriving (Read, Show)
```

Figure 6. Converting the WSDL enumeration to Haskell.

A WSDL description can also specify complex data types, i.e., types composed by the combination of other types. For example, a C# class named `Client`, containing as accessible fields its name (*string*) and age (`int`) would be typically exposed in WSDL as shown in Figure 7.

```
<s:complexType name="Client">
 <s:sequence>
   <s:element minOccurs="0" maxOccurs="1"
      name="name" type="s:string"/>
   <s:element minOccurs="1" maxOccurs="1"
      name="age" type="s:int"/>
    </s:sequence>
</s:complexType>
```

Figure 7. Complex type sample exposed in WSDL.

The chosen approach for mapping complex types exposed this way into Haskell consists of the definition of new Haskell data types with only one constructor, but with many labeled fields, as shown in Figure 8.

```
data Client = Client {
   name :: String,
   age :: Int
} deriving (Show)
```

Figure 8. Converting the WSDL complex type to Haskell.

If the complex type does not contain child elements (`<complextype/>`, in the WSDL), it is still converted into a Haskell data type containing only one constructor, but now with zero fields. For example, a C# class named `XYZ` that does not contain visible attributes would be converted into Haskell as the following data type:

```
data XYZ = XYZ
```

A complex type, additionally, may contain as a child element another complex type. For example, a C# class exposed in WSDL, named `Account`, could contain an object of type `Client` as its child element. For this case, Haskell data types containing other Haskell data types as child fields were used. Nevertheless, these possibilities caused a considerable increase of complexity in relation to the conversion of a Haskell data type to/from XML.

Lists are also supported by WSDL. For example, Figure 9 presents a complex type named `Bank`, which contain as child elements a list of `Accounts`.

```
<s:complexType name="Bank">
 <s:sequence>
  <s:element minOccurs="0" maxOccurs="1"
   name="accounts" type="s0:ArrayOfAccount"/>
 </s:sequence>
</s:complexType>
<s:complexType name="ArrayOfAccount">
 <s:sequence>
  <s:element minOccurs="0"
     maxOccurs="unbounded" name="Account"
     nillable="true" type="s0:Account"/>
 </s:sequence>
</s:complexType>
```

Figure 9. List sample exposed in WSDL.

Since lists or arrays are supported by the majority of today's programming languages, such as Haskell, this mapping presented no difficulty at all, as shown in Figure 10. Nevertheless, some list specifications encountered in WSDL files, such as the one presented in Figure 9, caused an extra indirection level, since an intermediary type representing the list (`ArrayOfAccount`) was now introduced and had to be dealt with by the generator.

```
data Bank = Bank {
  accounts :: [Account]
} deriving (Show)
```

Figure 10. Converting the WSDL list to Haskell.

Finally, once the mappings between the various data types specified in WSDL into Haskell are defined, the generation of a Haskell **module** structure deserves some explanation.

For each web service described in a WSDL document, a unique Haskell module is generated. The name of the Haskell module is defined by the *name* attribute of a WSDL XML node named *service*. Mapping from web methods into Haskell is also straightforward and intuitive. Each operation presented in the WSDL document, which corresponds to a web method, is converted into a Haskell function of the module under generation. An important observation is that the first character of the operation name must be converted to lower case, in order to comply with the Haskell syntax. This observation also applies to the field names of the generated Haskell data types (Figure 8).

It is worth noticing, however, that the names of methods and attributes are used without any changes when the XML received/sent by the web server is built. Therefore, besides the proxy generator makes available to Haskell programmers names with first letters in lower case, the SOAP XML nodes use the same original names defined in WSDL.

4.3 Converting to/from XML

Since the low-level details of a web service usage include the creation and interpretation of an XML document, the proxy generator should, in runtime, convert the values of web service parameters into *strings* and encapsulate them in XML tags, in order to send them through HTTP (SOAP) packages. On the other hand, the generator should also be able to extract, from an XML file, the *string* corresponding to a web method invocation result, as well as to convert this *string* into the expected return type.

The implementation of the Haskell standard functions `read` and `show` for primitive types and enumerations, available through the Haskell `Prelude` module, made this work easier. Whilst the `read` function converts a *string* into another data type, the `show` function does the inverse operation. Nevertheless, three issues are worth noticing in relation to this subject:

- If the `show` function is applied to a value that is already a *string*, the resulting *string* will be enclosed by double-quote characters (“). In the same way, if `show` is applied to a `Char` value, the resulting *string* will be enclosed by single-quote characters (‘). Since enclosing quote characters are not expected in values of a web service XML message, even if the value belongs to `Char` or `String` types, the proxy generator should identify when these characters are undesirable and need to be removed.

- When dealing with `Char` values, web services use the Unicode [19] standard. This way, the proxy generator should generate Haskell code that converts Haskell `Char` values into Unicode and vice-versa. The first version of the proxy generator uses the `chr` and `ord` standard Haskell functions, respectively. Therefore, it is actually dependent on the implementation of these two functions to provide character encoding support.

- If the function `read` is applied to a *string* representing a boolean value whose first letter is lower cased (such as `"true"` or `"false"`), it fails. This way, the generator should convert *strings* of such type to *strings* that start with upper cased letters (`"True"` and `"False"`).

Although the usage of the `read` and `show` functions was appropriate for primitive types and enumerations, complex types, on the other hand, demanded a more elaborated approach. A new Haskell interface named `XmlSerializable` had its definition added to the `SoapHttpClientProtocol` module. For each Haskell data type generated from a WSDL complex type (except for enumerations), an implementation of this new interface is also generated. The interface defines only two functions: `toXML`, which should convert the Haskell data type into an XML *string*, and `fromXML`, which should do the opposite operation. The proxy generator, therefore, should identify, according to the type of the value being converted to/from XML, if an implementation of `XmlSerializable` interface should be generated and invoked by the generated Haskell code.

In order to illustrate these concepts, suppose `Account` is a Haskell data type defined as shown in Figure 11. It contains as fields its `balance` (`Double`) and `client` (which is also a Haskell data type).

```
data Account = Account {
      client  :: Client,
      balance :: Double
} deriving (Show)
```

Figure 11. Definition of the `Account` Haskell data type.

The generated implementation for the `toXML` function is shown in Figure 12. The functions `xmlTagStart` and `xmlTagEnd`, defined in the `SoapHttpClientProtocol` module, are auxiliary functions for the generation of XML nodes. The contents of the `Account` data type child nodes are obtained in different ways for the `balance` field and for the `client` field: since the `client` field is a Haskell data type, the function `toXML` is invoked. On the other hand, the `balance` field belongs to a primitive type and, therefore, only the `show` function needs to be called[2].

```
toXml account  =
     (xmlTagStart "client")
  ++ (toXml       (client account))
  ++ (xmlTagEnd   "client")
  ++ (xmlTagStart "balance")
  ++ (show        (balance account))
  ++ (xmlTagEnd   "balance")
```

Figure 12. Generation of the `toXML` implementation for the `Account` data type.

The generation of the `fromXML` function implementation, on the other hand, is a little more complex, since the proxy generator should build the resulting Haskell data type (which can be composed by other Haskell data types). The generation of the `fromXML` implementation for the `Account` data type is presented in Figure 13.

Each resulting Haskell data type field is extracted individually from the response XML. The function `getNodeValues`, exported by `SoapHttpClientProtocol`, is an auxiliary function that extracts the contents of an XML node, receiving the desired node

[2] Actually, the `show` function is not really called, but another function which invokes `show` and verifies undesirable enclosing quotes (check subsection 4.3). The `show` function was kept in Figure 12 for simplicity purposes.

name as parameter. Since more than one node can have the same name in an XML file, `getNodeValues` returns a list, not a single value. Nevertheless, since it is expected that the XML response has only one value regarding the Haskell data type field, the `head` function is used to extract the first (and only) element of the list returned by `getNodeValues`.

Finally, a specific function is called according to the type of the field whose value will be extracted from the response XML. For the `balance` field, which belongs to a primitive (`Double`) type, the function `read` is invoked. For the `client` field, on the other hand, function `fromXML` is invoked, since `client` is a Haskell data type.

```
fromXml str = account
  where
    account = Account {
       client  = accountClient,
       balance = accountBalance
    }
    accountClient =
       fromXml $ head $
       getNodeValues str "client"
    accountBalance =
       read $ head $
       getNodeValues str "balance"
```

Figure 13. Generation of the `fromXML` implementation for the `Account` data type.

4.4 Generating the Final Exported Function

Suppose `getAccount` is a web method implemented in C# and defined in a C# class named `Bank`. Suppose that its signature is[3]:

```
public Account GetAccount (double balance)
```

According to what was previously explained, after the proxy generator is invoked, a Haskell module named `Bank` will be generated, containing the definition of a Haskell data type named `Account`, its `XmlSerializable` implementation and, finally, a function named `getAccount`, which can be used by the Haskell programmer. The implementation of this function is presented in Figure 14.

The first important point to be observed in the generated implementation refers to the function signature: since the web service invocation involves IO operations, the final function exported to the Haskell programmer returns an IO type, as expected [6].

The function body, on the other hand, has two distinct execution phases: first, the parameters expected by the `invokeWS` function are processed and `invokeWS` is called. The values defined in the `where` clause are generated from the web service WSDL description. In the generated code, we use zip to pair the elements of the name list and the value list of the parameters expected by the web method.

[3] In this example, an account balance, not a *string* account identifier, is received as parameter, in order to better illustrate the proxy generator capabilities.

Next, a function call that is applied to the obtained `invokeWS` result, which is still a *string*, is generated. This generated function depends on the web method return type. In Figure 14, for example, `fromXML` is invoked, since the return type (`Account`) is a Haskell data type. If the return type were `IO Double` or `IO Int`, `read` would be the generated function call. If the return type were a list, on the other hand, a more complex function would be applied to the resulting *string*. For example, if the web method returned a list of `Accounts`, the generated code would be:

```
return $ (\xml -> map fromXml
(getNodeValues xml "Account")) strResult
```

```
getAccount :: Double -> IO Account
getAccount balance = do
  strResult <- invokeWS uriStr name
                        action namespace
                        parameters response
  return $ fromXml strResult
  where
    uriStr =
       "http://localhost/TestWS/Bank.asmx"
    name = "GetAccount"
    action = "http://tempuri.org/GetAccount"
    namespace = "http://tempuri.org/"
    parametersValue = [(show balance)]
    parametersName = ["balance"]
    parameters = zip  parametersName
                      parametersValue
    response = "GetAccountResult"
```

Figure 14. Generated implementation of `getAccount`, which would be exported to the Haskell programmer[4].

4.5 WSDL Flexibility Problem Turnaround

Since web services were designed with multi-platform and multi-language concepts in mind, the XML schema used to specify and constrain WSDL descriptions is extremely flexible, allowing different extensions and interpretations. This way, the contents of two distinct WSDL descriptions, although being different, can represent exactly the same web service. For example, the same web service created through different tools (such as Microsoft Visual Studio .NET [22] and Borland JBuilder [3]) can have very different WSDL descriptions. This nuance makes the proxy generator job more difficult, since it should provide a generic WSDL interpretation, unattached to its origin. Taking this difficulty into account, the first version of the proxy generator was designed to work with web services generated by Visual Studio .NET. Web services created using other tools are not granted to work with the generator, but this limitation should be removed in future versions.

[4] The URL http://tempuri.org is the default web service namespace.

5. CONCLUSIONS

In spite of the difficulty presented in subsection 4.5, it could be observed that the generator implementation occurred very intuitively, since the service access/proffer model used by web services could be naturally mapped into Haskell types and functions. Nevertheless, peculiarities of having Haskell as a target language, such as syntax issues presented in Section 4, had to be dealt with.

The proxy generator architecture was developed in a reusable manner. Only its back-end needs to be changed in order to allow code generation for other target languages. This way, a natural evolution of this work would be an extension to other target languages, possibly belonging to other programming paradigms. We believe this extension will also occur intuitively, since the generator does not depend on its target language (Haskell), but uses concepts that are common to many other programming languages, such as primitive types, lists, data structures containing nested fields, functions, etc.

An equally interesting work is the implementation of a web services **provider** for Haskell, which could be based on an existing Haskell web server [8] or on a server that supports ASP.NET [1], which would host code resultant from the compilation of Haskell to the .NET Platform [16]. Extensions like these would contribute to the final purpose of situating Haskell as a concrete alternative to the implementation of distributed applications and components.

6. REFERENCES

[1] ASP.NET: The Official Microsoft ASP.NET Site, http://www.asp.net.

[2] Birrell A. D., Nelson B. J., Implementing Remote Procedure Calls, ACM Transactions on Computer Systems 2, 1 (1984).

[3] Developer.com, Developing Web Services with Borland JBuilder Enterprise and BEA WebLogic Server, http://www.developer.com/java/ent/article.php/3485321.

[4] Gray W., Bringert B., Haskell HTTP Module, www.bringert.net/haskell-xml-rpc/http.html.

[5] Haskell.org., About Haskell, http://www.haskell.org/aboutHaskell.html.

[6] Hudak P., Peterson J., Fasel J., A Gentle Introduction to Haskell, Version 98: Input/Output, http://www.haskell.org/tutorial/io.html.

[7] Hugs98 for .NET, http://galois.com/~sof/hugs98.net/.

[8] Marlow S., Writing High-Performance Server Applications in Haskell, Case Study: A Haskell Web Server, Haskell Workshop, Montreal, Canada, September 2000.

[9] Meijer E., Finne S., Lambada, Haskell as a better Java, Electronic Notes in Theoretical Computer Science 41, no. 1 (2001).

[10] Microsoft.com, Microsoft .NET Homepage, http://www.microsoft.com/net/.

[11] Microsoft Developers Network, Visual C# Developer Center, http://msdn.microsoft.com/vcsharp/.

[12] Network Integrated Access (Netia.com), The Next Major Platform Shift: Java Computing Changes Everything, http://www.netia.com/Sun_Java_Sea_Change.pdf.

[13] Panne S., HOpenGL: An OpenGL/GLUT binding for Haskell, http://www.haskell.org/HOpenGL/.

[14] Peyton-Jones S., Meijer E., Leijen D., Scripting COM components in Haskell, IEEE Fifth Inter-national Conference on Software Reuse, Vancouver, BC, 1998

[15] Sun Microsystems, Java 2 Platform, Enterprise Edition, http://java.sun.com/j2ee/.

[16] The Haskell .NET Project, http://www.cin.ufpe.br/~haskell/haskelldotnet.

[17] The Lambda Complex, Why does Haskell matter?, http://www.haskell.org/complex/why_does_haskell_matter.html.

[18] Thompson S., Haskell: The Craft of Functional Programming, p. 239-275, Addison-Wesley, 1996.

[19] Unicode.org, What is Unicode, http://www.unicode.org/standard/WhatIsUnicode.html.

[20] W3C.org, World Wide Web Consortium, http://www.w3.org/.

[21] Wadler P., Why no one uses functional languages, ACM SIGPLAN Notices, august/1998.

[22] Westwind.com, Creating and using Web Services with the .NET framework and Visual Studio.Net, http://www.west-wind.com/presentations/dotnetwebservices/DotNetWebServices.asp.