

HAIFA : An XML Based Interoperability Solution for Haskell

Simon Foster

University of Sheffield

Abstract

Recent research in orchestration of composite web-services has highlighted Haskell as a language ideal for workflow orchestration semantics at an implementation level. Since Haskell is built on the concept of building larger functions by composing a number of smaller functions, it is ideal for composing services as well. In order to compose service operations though, it is necessary to enable Haskell to view services as side-effecting functions. Our goal, accordingly, is the creation of a framework both for client-side access of web-service operations, in terms of Haskell functions, and also a server-side function publishing framework, which would allow Haskell functions to be bound to URLs for execution. This paper introduces HAIFA, a service interoperation toolkit which goes some of the way to providing these features, and provides a foundation on which service composition can be achieved.

1 INTRODUCTION

Service oriented computing is being increasingly accepted as the paradigm with which web-based interoperability can be achieved. By re-using much of the existing protocol stack of the web, such technologies enable businesses to propagate their services using invocable operations with which applications may communicate to enable increased functionality. However, this paradigm alone does not deal with the question of composite functionality which can be built by combining the capabilities of several services. Service composition is a very important issue as single atomic services only serve to provide functionality relevant to their domain. For example a banking service only knows how to manipulate bank accounts, and a travel service only knows how to book transport arrangements.

Recent research [1] in orchestration of composite web-services using the service ontology OWL-S [3] (formerly DAML-S) has highlighted Haskell as a language ideal for workflow orchestration semantics at an implementation level. Since Haskell is built on the concept of building larger functions by composing together a number of smaller functions, it is ideal for composing services as well. In order to compose service operations though, it is necessary to enable the viewing of services as side-effecting functions in Haskell. Many features in Haskell also provide an ideal platform on which various data-processing applications can be elegantly constructed, and this motivates the ability to create atomic services in Haskell. Our goal is, accordingly, the creation of a framework for both client-side access of web-service operations, in terms of Haskell functions, and a server-side function

publishing framework, which would allow Haskell functions to be bound to URLs for execution.

In light of this we have implemented the Haskell Application Interoperation Framework/Architecture (HAIFA), a library providing the basic components for web-based interoperability, in the form of a service modelling framework. This paper seeks to further motivate the implementation of such a framework in Haskell, and then describe the core components of HAIFA. The web-service architecture is first set forth in Section 2 along with a description of how this can be modelled in Haskell. Following the motivation of an annotating XML serializer, in Section 3 we describe how type-classes can be utilized for this purpose as an extension of existing Haskell XML tools. In Section 4 the realization of client side service interoperation is achieved, and finally in Section 5 a method of publishing Haskell functions in a service is described.

2 MODELLING SERVICES

In order to take advantage of the functionality present in Haskell for producing composite services, it is first necessary to define a method for modelling single services in Haskell. The Simple Object Access Protocol (SOAP/1.1) has, for several years, been the de-facto standard for forming invocation messages. Despite being superseded by SOAP 1.2 and not itself being a World-Wide Web Consortium¹ (W3C) recommendation, it still retains a strong user base.

SOAP style invocations are usually “executed” via HTTP/1.1 [4], that is to say the POST mechanism present in HTTP is used to model remote procedure calls with its request/response procedure. Although this is not necessarily the case, in most instances the HTTP method is used, since it makes use of port 80, which is usually always open on servers. As a result our research has primarily focused on SOAP over HTTP.

Informally, a SOAP invocation occurs when a client passes a SOAP envelope containing a request message to a server running a service, the operation is executed on the server and a response message in another envelope is sent back. SOAP tends to be encoded using XML (although this is not strictly necessary), and thus parameters of the encapsulated operation are as well, with W3C XML Schema usually providing types.

A SOAP envelope consists of a body, which encapsulates the message, and a set of headers, which convey meta-data about the body. The body itself can hold either a message or a SOAP fault, which is essentially a serialized exception. This leads to the conclusion that a SOAP envelope is intrinsically parametrically polymorphic, and this is represented in its XML schema, which leaves the type of the body undefined.

As a result, in Haskell the SOAP envelope is represented by a parametric type,

¹Specifications for XML, XML Schema, SOAP and WSDL can be found at the Web Consortium’s site at <http://www.w3.org>

and each message's structure can be any serializable data-type. The overall types adopted for the `Envelope` and `Body` have the following structure:

```
data Envelope a =  
    Envelope { header :: [Content]  
              , body   :: Body a  
              }  
  
data Body a = Body (Either Fault a)
```

Note that the `Body` can be either a `Fault` or a message represented by `a`. The structure of the `Fault` is show below:

```
data Fault =  
    Fault { faultcode :: QName  
          , faultstring :: String  
          , faultactor :: Maybe String  
          , detail     :: Maybe Content  
          }
```

It is a four part structure consisting of a fault code which defines the error with a qualified name, a fault string to provide a description, and optionally a fault actor which is the identity of the link partner who the error relates to and detail which contains further detail on the fault encoded as XML.

In the web-services world, services can be described in the language WSDL (Web-Service Description Language), though there is no formal link to the implementation languages. WSDL, along with suitable semantic descriptions, facilitates web-service discovery by defining a service's interface, thus allowing programs to gain access without intervention. WSDL defines a service operation in terms of

- Messages (i.e. the inputs and outputs)
- Interfaces (the abstract types of operations within the service, in terms of their input and output messages)
- Bindings, which concretize the interface with an execution medium (e.g SOAP over HTTP)

A simple service operation interface in WSDL could be defined like this:

```
<wsdl>  
  ...  
  <message name="inputMessage"> ... </message>  
  <message name="outputMessage"> ... </message>  
  
  <interface name="MyServiceInterface">  
    <operation name="MyOperation">
```

```

        <input message="inputMessage" />
        <output message="outputMessage" />
    </operation>
</interface>
...
</wsdl>

```

This defines the structure of the input and the output messages (details hidden) and defines a service interface containing a single operation, whose constituent input and output messages are defined. In reality, this interface would then be bound to an execution protocol and assigned to an endpoint URL, but for the moment only the interface abstraction is of interest. With this in mind, one view of the service bound to SOAP - which we shall shortly discard - is as a pure function mapping the input message to the output message. The resulting SOAP operation stub could be typed as:

$$\textit{Envelope InputMessage} \rightarrow \textit{Envelope OutputMessage}$$

or ignoring exceptions and headers, simply

$$\textit{InputMessage} \rightarrow \textit{OutputMessage}$$

This model is, however, limited to the domain of services which do not affect the state of the world. The reality is that most services do have side-effects (even if they are invisible), and thus just modelling services as pure functions is insufficient. In Haskell, functions explicitly state whether they change the world's state since side-effects are propagated via the `IO` monad. It is safe to say then that web-services, even abstractly, cannot be modelled as functions, but should instead be modelled as monadic operations. By modelling service operations in this way, sequential composition of services can be achieved via the usual `bind` (`>>=`) and `return` functions. An example sequential service composition is considered below.

Assume that there exist three functions which act as proxies for three services (all of which have invisible side-effects which we will presume can be satisfactorily modelled by the `IO` monad);

$$\begin{aligned} \textit{findConference} &:: \textit{Name} && \rightarrow \textit{IO Date} \\ \textit{bookTravel} &:: \textit{Date} && \rightarrow \textit{IO TravelDetail} \\ \textit{updateMyDiary} &:: \textit{TravelDetail} && \rightarrow \textit{IO ()} \end{aligned}$$

These represent services for finding a conference with the given name, and returning the date it occurs, booking travel on a particular date and updating a diary with the new arrangement. Each of the functions represents a single concurrent process which must be orchestrated within a workflow to form a single process. In this case the workflow is simply the execution of each service sequentially, with a single parallel data-flow. If we elide details of scheduling (which is required in

more complex composition patterns and to guarantee a service's pre-conditions are fulfilled), the function representing the composite service can be constructed by composing together the three service's functions:

$$\begin{aligned} \text{conferenceBooker} &:: \text{Name} \rightarrow \text{IO} () \\ \text{conferenceBooker } n &= \text{findConference } n \gg \text{bookTravel} \\ &\gg \text{updateMyDiary} \end{aligned}$$

In order to achieve this level of abstraction, and assuming that the type of each operation is known beforehand, three components are required

1. An XML serializer for producing and consuming message representations.
2. An HTTP-based client-side executor for services.
3. A function publisher, so that composite services can be propagated.

We now describe how this architecture can be realised.

3 EXTENSIBLE XML SERIALIZATION

In Section 2 we fixed a data-type for representing SOAP envelopes and defined a method for representing invocations in Haskell. It is now necessary to define how SOAP envelopes and encapsulated messages are to be encoded as XML. Existing XML serialization solutions for Haskell provide most of the functionality required for building XML based applications. In particular, HaXML² [13], the most well known XML library for Haskell, provides an elegant, yet simplistic, `Haskell2Xml` type-class, which performs serialization and deserialization of Haskell types to XML. Essentially it provides an XML version of the `Read` and `Show` classes, which perform serialization in a similar way, but on `String` data. With the addition of DrIFT³ generated instances to derive serializers automatically, it would appear that the only feature currently missing is XML namespaces (vital for modular service interoperation and schema typing), which can be added by updating the relevant types to enhance expressivity.

However, there is one key feature which HaXML's serializer cannot currently deal with, and to which we intend to show there is currently no adequate solution provided; the concept of data annotation. Within the sphere of interoperability, annotation of XML data is of paramount importance, as on its own XML data conveys no meaning, relying instead on a pre-processed schema for the description of the structure. In many cases this is adequate, but when message structure is left undefined (for example, via wild-card schema constructs), it is important that data

²Although we refer to HaXML in the paper, HAIFA uses the Haskell XML Toolbox [11] as it provides more features and a more expressive data-structure. Although different types are used to represent XML trees, the type-class principles are the same.

³DrIFT is a type-sensitive pre-processor for Haskell, c.f. <http://repetae.net/john/computer/haskell/DrIFT/>

is annotated. Further, as more come to the conclusion that a syntactic structure definition alone is inadequate to describe data, and semantic annotation becomes important to convey the meaning of the data [2], data annotation becomes increasingly important.

To exemplify, a simple SOAP envelope which supplies the input to an operation for adding two numbers together might be encoded in XML like this:

```
<Envelope><Body>
  <addRequest>
    <num1>5</num1>
    <num2>10</num2>
  </addRequest>
</Body></Envelope>
```

Without an external schema it is impossible to determine the type of the data in the elements named `num1` and `num2`. One way around this problem is, as we have seen, to define a schema for the message structure. However if W3C XML Schema is used to define the message structure it is necessary to fix these as values having a monomorphic type. If instead we wish to take advantage of polymorphism, the type data needs to be carried along with the data itself.

Haskell's type-classes allow elegant attachment of meta-data to types, but it is not possible to utilize this data within a closed type-class definition, as provided by `Haskell2Xml`, because the context is fixed. It is therefore necessary that serializers can be provided with a customizable context, which can convey meta-data and allow further functionality to be hooked in.

A method for context customization has been described by Hughes in [7]. This allows type-classes to be referenced as first-class types by assigning an algebraic type, called a *dictionary*, which allows a class' functionality to be determined from its structure. This proxy type can then be inserted into the serialization process by means of an extra parameter of the main serialization class, along with a hook function which encodes the extra data into the tree.

We thus introduce a version of `Haskell2Xml`⁴ with the following class head

```
— h is the context hook and a is the serialized type.
class Data (DictHaskell2Xml h) a => Haskell2Xml h a
where
  — Serialize the type a in the context of hook h.
  toContent :: h -> a -> [Content]
  — Deserialize in the context of h to type a.
  fromContent :: h -> [Content] -> (a, [Content])
```

where `h` is the type of the hook that is to be applied to the serialization process and `Content` is the HaXML type representing the XML tree. The `Data` class

⁴The class in HAIFA is actually called `XMLData` and has different method names, but has the same functionality

(as defined in [8]) acts as the carrier for the customizable context (it also provides reflection functions, but we elide these for now). `DictHaskell2Xml` is the proxy type for `Haskell2Xml`, as shown below:

```
data DictHaskell2Xml h a =
  DictHaskell2Xml { toContentD :: a → [Content]
                  , fromContentD :: [Content] → (a, [Content])
                  }
```

The `h` parameter is missing from the function proxies because it is automatically propagated via the dictionary definition, as shall be demonstrated. This new type is currently only an interface for the class and needs to be populated with actual function definitions to be useful. This task is done via the `dict` function of the `Sat` class, which has the job of returning a value of the type it is parameterised over, but populated with actual data. The definition of the class, and the instance for the `Haskell2Xml` dictionary is as shown:

```
— Return a populated dictionary for the given proxy type
class Sat a where dict :: a
— Populate the Haskell2Xml dictionary.
instance (XmlHook h a, Haskell2Xml h a) ⇒ Sat (DictHaskell2Xml h a)
where
  dict = DictHaskell2Xml
    { — Perform serialization and annotate
      toContentD = λ x . annotate q x (toContent q x)
      — Perform deserialization
      , fromContentD = fromContent q
    }
  where q = ⊥ :: h — Propagate the hook via q.
```

The instance of `Sat` for `DictHaskell2Xml` populates the dictionary with functions from the actual class by applying the constraint that for a dictionary to be extracted for a particular type and hook, there must be a suitable instance available of `Haskell2Xml`. The `Sat` instance is also constrained with a further class called `XmlHook` whose single function performs a transformation on the XML document, inserting the annotation. The presence of the constraint here guarantees that every type down the serialization tree can have the hook applied to it. The class structure is shown below:

```
class XmlHook h a
where
  — Produce a document transformation for hook h and type a.
  annotate :: h → a → ([Content] → [Content])
```

Each hook should be represented by a dummy type with no constructors (indicated by the propagation of its value as `⊥` in the dictionary population), since it

merely acts as a label. The proxy for `toXml` in the dictionary definition automatically annotates the serialized output, making this application transparent.

Although `XmlHook` exposes the type `a` directly, rather than specifying annotations for specific types, it is more convenient to constrain the type `a` with a given class, thus allowing the hook to pull in external data from other classes. For instance, a simple Haskell type annotator might be:

```

— Label type with no constructors
data TypeHook
— Typeable stores unique type identifiers for Haskell types
instance Typeable a => XmlHook TypeHook a
where
    annotate _ x = ⟨Add XML attribute for annotation⟩

```

Using this annotator has the effect of adding an attribute specifying the Haskell type to XML element declarations, for example:

```
<year haskellType="Data.Int.Int">2005</year>
```

This method can thus be used to annotate serialized data with a wide variety of meta-data, which is decoupled from the original serializer definition, and highly modular in nature. It should be noted that Haskell cannot, as such, easily take advantages of type annotations. Type meta-data cannot be used beyond simple validation since all types must be decided on at compile time. A more advanced method involving dynamic typing would be required in this instance.

Finally we can create the actual functions which will serialize and deserialize a type within a given hook's context. This function uses the type of hook and data to create a suitable dictionary, acquire the `toXml` function via `toXmlD` and run it on the data.

```

toXml' ::(Data (DictHaskell2Xml h) Int, XmlHook h a)
        => h -> a -> [Content]
toXml' (q :: h) (x :: a) = toXmlD (dict :: DictHaskell2Xml h a) x

```

However, one caveat still exists in the serializer as it stands. The reader may have noticed that no code has been supplied for adding the XML attribute for Haskell type annotation; this is simply because it is not possible to do this using the HaXML data-type if our serializer recursively produces content. In HaXML attributes are not treated as regular content, but as sub-properties of an element and cannot exist outside of one. Now in the context of an entire XML document this is absolutely correct - it is not possible for attributes to exist outside elements, but when recursively building the tree by composing a number of functions it *must* be possible. This is particularly important when the attribute belongs to the data itself and not to the encapsulating element, as is often the case when dealing with polymorphic data. This, along with HaXML's lack of namespace support, motivates our

using the Haskell XML Toolbox [11], which has a more liberal attitude, allowing attributes to appear anywhere. This naturally means that false XML structures can be represented, but in practice this is not a problem. The reader must not assume that the intention is to condemn HaXML’s model; it perfectly matches the W3C XML Schema data model. Rather a more expressive model is sought.

One further feature which can be added to the serializer is an internal schema definition, which can define the structure of algebraic types in terms of their elements and attributes. Each type is assigned a list of schemas, defining how each of the child terms of the type is represented in XML. This can then be used to automatically serialize and deserialize data and requires much less effort than creating the serializers manually. For the purpose of HAIFA, a schema structure was chosen which would be more natural for use with Haskell data-types, rather than using an established schema, such as XML Schema or RELAX-NG. Nevertheless, the schema structure has been defined with type mapping in mind, and is thus strictly more general than XML Schema (whilst RELAX-NG is a future target) so as to allow complete representation of its data-model. The data-type used to represent XML structures (or *particles*) of each term is shown below:

— Schema representation of XML particles

```

data PartSchema = Attr Occurs String (Maybe URI) |
                  Elem Occurs String (Maybe URI) |
                  Choice Occurs [PartSchema]      |
                  Sequence Occurs [PartSchema]     |
                  Inter Occurs [PartSchema]         |
                  AnyAttr AnyRes                    |
                  AnyElement Occurs AnyRes         |
                  TextContent

```

Each particle’s cardinality constraint is defined by the *Occurs* type, which defines the upper and lower bounds of the number of times the particle can occur. Such constraints are vital for list data, where each list member may be encapsulated by a recurring tag. A schema is allocated to a type by means of new method of `Haskell2Xml`. Each descriptor respectively represents

- Attributes, which are treated uniformly with elements
- Elements (i.e. XML tags)
- Choices of multiple structural options
- Ordered sequences of particles
- Unordered (Interleaved) particles
- Wildcard attributes (where *AnyRes* defines any restrictions on what type of attribute can occur)

- Wildcard elements
- Textual content (at leaf nodes)

The `Haskell2Xml` class is then assigned default definitions for `fromContent` and `toContent` which contain rules for processing the data-type using the schema definition. Defining serializers for mapped types thus becomes simply a case of mapping from the type representing the actual XML schema, to the internal schema of `Haskell2Xml`. Further, the reflection data available via the `Data` class can be used to generate serializers automatically for particular types. For example, record syntax data-types can have serializers generated which serialize as a sequence of elements named after the type's field names.

4 SERVICE INTEROPERATION

Having equipped ourselves with a suitable XML serializer, we are now in a position to satisfy our original goal, the invocation of SOAP operations as functions. We do this via a general invocation function, which views calls as consisting of three components

1. A serializable input message data-type.
2. A serializable output message data-type.
3. An executor, which performs the invocation.

The two message data-types can be made serializable with suitable instances of `Haskell2Xml`, but the executor requires more discussion. An executor can be viewed as a side-effecting IO operation which transforms one serialized message represented by a `String` to another. Thus, executors are represented by the following class:

```
class ServiceExecutor a
  where
    — Produce an execution function from the executor type a.
    executeService :: a → String → IO String
```

Thus a serialized call can be formed from the input message, executed, and the response deserialized to form the output message. This gives rise to the function `soapCall` which acts as the general function wrapper for services, and has the following type:

```
soapCall :: (Haskell2Xml h a, Haskell2Xml h b,
             ServiceExecutor e)
          ⇒ e → h → a → IO (Either Fault b)
```

This simple function makes the creation of SOAP stubs simply a case of creating serializers for the message types and defining a suitable executor for the service. As an example the executor type for HTTP is shown below:

```
data HTTPExec =
  HTTPExec{ httpEndpoint    :: URI
            , httpSOAPAction :: Maybe URI }
```

It simply defines a HTTP request as consisting of a suitable URL⁵, and maybe a SOAP Action (rarely used, but part of the SOAP/1.1 specification). The Haskell HTTP library⁶ can then be utilized to build an instance of `ServiceExecutor` for the `HTTPExec` type which performs a suitable POST request to the given URL, returning the body of the reply. Thus our execution paradigm is complete, and creating stubs is simply a case of defining the interface's type structure and providing a suitable binding to `soapCall`.

For example, the `googleSearch` operation of the the Google web-service⁷ can be accessed with the following stub:

```
googleSearch :: DoGoogleSearch →
              IO (Either Fault DoGoogleSearchResponse)
googleSearch = soapCall (HTTPExec googleURL Nothing) XSITypeHook
```

where `DoGoogleSearch` and `DoGoogleSearchResponse` are the pre-defined input and output message types respectively, `googleURL` is the endpoint URL of the Google Web-Service and `XSITypeHook` is the annotator which inserts XML Schema type annotations.

5 PUBLISHING FUNCTIONS

In this section the publishing of Haskell functions as services will be examined. Since we've established that a service operation can be modelled as a function from the input to the output message plus a side-effect, we can generalise further from the previous function model and say any function of the form

— `MonadIO` is the class of IO based side-effecting monads.
 $f :: (Haskell2Xml\ h\ a, Haskell2Xml\ h\ b, MonadIO\ m) \Rightarrow a \rightarrow m\ b$

can be directly be invoked via a service. All that is required is the machinery to bind it to an execution protocol. As mentioned, this research primarily focuses on SOAP/HTTP, and so the first step is to create a simple HTTP server for Haskell.

There already exists such a server in reasonably good working order, called HWS-WP [9][12], but it cannot be used directly because the HTTP/1.1 implementation is meant for testing only, essentially only supporting propagation of HTML

⁵Note that we use the URI data-type since it represents a superset of URLs.

⁶See <http://www.haskell.org/http/>

⁷See <http://www.google.com/apis/> for details of the Google API

documents. What it does provide however is a good starting point for creating the request handler, and thus it has been combined with the Haskell HTTP library, which provides a full implementation. The HTTP library contains two key data-types, `Request` and `Response` which represent a HTTP request and response respectively. A notion of HTTP handlers is adopted which mirrors the operation architecture, in that a handler is viewed as a function from a request to a response and side-effect, which can then be bound to an endpoint URL. The HTTP server thus takes a set of handlers, as well as various configuration options, and builds a server in the `IO` monad, as shown below.

```

— Associate a server URL string with a handler function
type HTTPHandler =
    (String, Config → Request → IO (Either String Response))
— Run the HTTP Server, using the configuration filename given with a set of
— HTTP Handlers and XML configuration readers.
httpServer :: FilePath → [HTTPHandler] →
    [Document → Config → Config] → IO ()

```

A HTTP handler is viewed as a pair, mapping a local URL on the server to a handler function taking a global configuration, a HTTP request and producing either an error string or a HTTP response in the `IO` monad.

In the SOAP service architecture, it is possible to have multiple operations bound to a single URL. Many authors (ourselves included), criticise this feature, due to its non-composable view of an application's interface, exemplified by the fact that a service cannot simply be expanded or contracted by adding and removing URLs. Nevertheless, it is a feature which, for now, must be dealt with. In the future we would prefer a view of endpoint advertisement similar to that taken by the REST [5] architecture, in which the URL hierarchy is used to organize operations directly.

The process of binding Haskell functions to a URL via SOAP is a process of creating an interface for the functions between serializable data-types, and then converting these heterogeneous functions to a homogeneous handler function which can be inserted into the HTTP server. To facilitate this conversion we have created a class called `Service`, containing a single function, `publish`. This takes a HTTP server configuration, a wrapped Haskell function parameterised over a `MonadIO` monad and produces a transformation from one XML tree to another in the monad, encapsulating the SOAP operation. The structure of this class is show below:

```

— Parameterised over the function wrapper type and the monad
— being used to execute the service.
class Service s m
  where
    publish :: Config → s m → (Document → m Document)

```

Each wrapper allows conversion of different types of function to a service operation. The wrapper is primarily there so that all functions can be parameterised

over the monad represented by `m`, even if it is not used in the function itself. For example, the wrapper for a pure transformation function is

```
data MonadIO m => SimpleFunc a b m =
    SimpleFunc { sfunc :: (a -> b) }
```

This type is parameterised over the input type, the output type and the monad being used. It is important that `m` is the last variable so that the type `SimpleFunc A B` (where `A` and `B` are types) can unify with `s` in the `publish` function, which is parameterised over `m`.

The possibility of errors in code exists, and thus exceptions must be dealt with. This can be achieved via a function which converts the Haskell `Exception` type, which can be thrown by a function, to a `SOAP Fault` type and serializes the representation.

Finally each function can be integrated into the handler, and fitted with a guard to pick which of the functions is being called by checking the name of the incoming message. Each valid request will yield a 200 HTTP response (i.e. request succeeded, here is the response), with the body of the HTTP response conveying the SOAP envelope.

All of this is done by the `buildWS` function which allows multiple heterogeneous functions to be bound to an operation name and turned into a single HTTP handler via an existentially quantified type which wraps the polymorphic service functions. The function also takes an optional HTML document which will be displayed if the service's URL is browsed, i.e. when an HTTP GET is received, as opposed to a POST. The structure of `buildWS` and the associated existentially quantified `PubFunc` type is shown below:

- Allows any member of the `Service` class to be wrapped up as
- a monomorphic type parameterised over the operation monad.

```
data PubFunc m =  $\forall s$  . Service s m => PubFunc (s m)
```

- Build a service from a set of name/operation pairs and a
- HTML document for display. Produce a function from a config
- and request to either an error or response.

```
buildWS :: MonadIO m => [(String, PubFunc m)] -> Maybe Html ->
    (Config -> Request -> m (Either String Response))
```

Please see Appendix A for an example service module built using this API.

6 FUTURE WORK

A key assumption made in this paper is that the type of a service's operation is known before hand, although in reality this is rarely the case. We are currently working on an XML Schema type mapper which works in concert with the XML serializer class, and currently has the ability to map Schema types to Haskell types,

as well as generating serializers via the intermediate schema representation, though currently not from Haskell to XML Schema (primarily because Haskell algebraic types are a lot more expressive than XML Schema). This is one step closer to the development of a discovery framework to complement HAIFA, which would allow the automatic integration of particular services with Haskell using the service broker architecture and service description languages such as WSDL.

All this is leading toward our ultimate goal which is the creation of a service composition framework, using Haskell to provide orchestration of workflows. It is our hope that by making use of semantic web-services and a future discovery framework it will be possible to automatically find the appropriate web-services required for a task and then compose them together.

The example of service composition in Section 2 is only very simple and in reality can have little practical use. Instead, a wider variety of constructs are required other than sequence, and a model of data-flow so that outputs can be mediated between services. All of this can be achieved with a suitable process calculus, which can provide orchestration of a composite workflow and serialization of the associated I/O operations. This work is currently being undertaken in the Cashew project⁸ [6], with HAIFA providing the low-level web-service support.

7 CONCLUSION

We have outlined a web-service framework for Haskell which allows both the accessing of existing services, and the creation of new ones, in Haskell. This framework can be used both for client-side mapping of web-services to functions, and the creation of services from Haskell functions. Further, this work leaves the door open for using Haskell as service composition language.

The work on XML that we have demonstrated represents only a fraction of the features exhibited by the serializer in HAIFA, and in future we hope to achieve full schema support to enable transparent mapping of schema types to Haskell types and service discovery.

It is fair to say that this paper has not described anything which can be considered as “new” to the functional programming community; instead it has used a number of well known techniques to give an elegant implementation of the web-service architecture. This work gives rise to a grounding for Cashew based orchestrations, in the form of monadic bindings for the underlying calculus’ processes. How such processes form operation serializations has been demonstrated, whilst eliding the more complex issues of scheduling. The details of how high-level orchestration semantics are formed can be found in our operational semantics paper[10].

⁸The Cashew project page can be found at
<http://www.dcs.shef.ac.uk/~barry/CASheW-s/>

ACKNOWLEDGMENTS

We would like to thank Dr. Matt Fairtlough and Dr. Marian Gheorghe for supervising the original undergraduate dissertation which led to this work, and for providing help and encouragement along the way. Thanks also go to Andrew Hughes for providing helpful suggestions regarding this work and Atheesh Sanka for being the first person to test these tools out. Finally thanks go to Barry Norton, for his continuing support, inspiration, and for having put forward the original proposal which set this project in motion.

REFERENCES

- [1] A. Ankolekar, F. Huch, and K. P. Sycara. Concurrent semantics for the web services specification language DAML-s. In *Proc. 5th Intl. Conference on Coordination Models and Languages*, pages 14–21, 2002.
- [2] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):35–43, 2001.
- [3] M. Burstein, J. Hobbs, O. Lassila, and D. McDermott. OWL-S: Semantic markup for web services. <http://www.daml.org/services/owl-s/1.1/overview/>, 2004.
- [4] R. Fielding, U. Irvine, and J. Gettys. Hypertext Transfer Protocol – HTTP/1.1 (RFC 2616). Technical report, Internet Society, 1999.
- [5] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
- [6] S. Foster, A. Hughes, and B. Norton. Composition and semantic enhancement of web-services : The CASheW-s project. In *Proc. 1st Young Researcher’s Workshop on Service Oriented Computing (YR-SOC 2005)*, pages 29–32, 2005.
- [7] J. Hughes. Restricted Data-types in Haskell. In E. Meijer, editor, *3rd Haskell Workshop*, September 1999.
- [8] R. Lämmel and S. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. 10th Intl. Conference on Functional Programming (ICFP 2005)*, September 2005.
- [9] S. Marlow. Writing high-performance server applications in Haskell, case study: A Haskell web server. In R. Hinze, editor, *Haskell Workshop*, Montreal, Canada, September 2000.
- [10] B. Norton, S. Foster, and A. Hughes. A compositional operational semantics for OWL-S. In *Proc. 2nd Intl. Workshop on Web Services and Formal Methods (WS-FM 2005)*, September 2005.
- [11] M. Schmidt. Design and implementation of a validating XML parser in Haskell. Master’s thesis, University of Applied Sciences in Wedel, 2002.
- [12] M. Sjögren. Dynamic loading and web servers in Haskell. <http://www.mdstud.chalmers.se/~md9ms/hws-wp/>, October 2000.
- [13] M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP’99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.

A “HELLO WORLD” SOAP SERVICE IN HASKELL

The module below defines a simple SOAP/HTTP service which accepts a message encapsulating a person’s name as a `String` and returns a message containing a greeting to that name.

```
module HelloWorldService where

-- Import the XML Serializer and service publisher.
import Text.XML.Serializer
import Network.Service

-- Input message type, contains a single
-- string representing the name.
data HWRequest = HWRequest String
-- Output message type, contains the returned greeting.
data HWResponse = HWResponse String

-- The actual pure-function which will be published,
-- from the input message to the output message.
helloWorld :: HWRequest -> HWResponse
helloWorld (HWRequest n)
    = HWResponse ("Hello there " ++ n ++ "!")

-- Make messages serializable using Haskell2Xml class.
-- Data constraint is explicit to allow hook propagation.
instance Data (DictHaskell2Xml h) HWRequest
    => Haskell2Xml h HWRequest where ...

instance Data (DictHaskell2Xml h) HWResponse
    => Haskell2Xml h HWResponse where ...

-- Define the service’s HTTP handler
helloWorldService :: HTTPHandler
helloWorldService =
    ("helloWorldService",
     buildWS [("helloWorldRequest",
                PubFunc (SimpleFunc helloWorld))])

-- Create the server with the given configuration file and
-- a single handler function.
runHelloWorld :: IO ()
runHelloWorld
    = httpServer "config.xml" [helloWorldService] []
```