One Document to Bind Them: Combining XML, Web Services, and the Semantic Web

Harry Halpin
Institute for Communicating and Collaborative Systems
University of Edinburgh
2 Buccleuch Place
Edinburgh, United Kingdom
H.Halpin@ed.ac.uk

Henry S. Thompson
World Wide Web Consortium
and University of Edinburgh
2 Buccleuch Place
Edinburgh, United Kingdom
ht@inf.ed.ac.uk

ABSTRACT

We present a paradigm for uniting the diverse strands of XML-based Web technologies by allowing them to be incorporated within a single document. This overcomes the distinction between programs and data to make XML truly "self-describing." A proposal for a lightweight yet powerful functional XML vocabulary called "Semantic fXML" is detailed, based on the well-understood functional programming paradigm and resembling the embedding of Lisp directly in XML. Infosets are made "dynamic," since documents can now directly embed local processes or Web Services into their Infoset. An optional typing regime for infosets is provided by Semantic Web ontologies. By regarding Web Services as functions and the Semantic Web as providing types, and tying it all together within a single XML vocabulary, the Web can *compute*. In this light, the real Web 2.0 can be considered the transformation of the Web from a universal information space to a universal computation space.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages

General Terms

Design, Theory

Keywords

Pipelining, Semantic Web, Web Services, XML, functional programming

1. INTRODUCTION

1.1 Standard Confusion

Given the current plethora of XML-based standards and technologies, XML is being pulled in multiple directions: the Semantic Web, Web Services, and maintaining the good-old fashioned Web based on REST and HTML. As arguments over the relative merits of tree-structured XML versus graph-structured RDF occupy some, on the ground the Web is moving fast with the rapid adoption of microformats such as RSS and the use of AJAX (Asynchronous Javascript and XML) to deliver applications quickly over the Web. Currently, the Semantic Web and the infamous "Web Service Stack" seem far removed from the Web of everyday users. However, many of the "Web 2.0" technologies like AJAX and the various microformats that come "close to RDF" are clever but present abstraction and data integration issues. The question of the hour is: Can the Web community articulate a vision that does justice to the direction the Web is heading? By re-inspecting the idea of "self-describing" that gave XML much of its original impetus, we believe an answer can be found. Note that the original impetus for this thinking on the role of "self-describing" was due to Tim Berners-Lee, and a preliminary description of the proposed solution, in the form of a reconstruction of XML pipelines as functional programs, was presented at XML 2005 [26] and is briefly recapped in Section 2 in this paper.

WWW 2006, May 23–26, 2006, Edinburgh, Scotland.
ACM 1-59593-323-9/06/0005.

1.2 Is XML Self-Describing?

XML has always been touted as "self-describing," and Tim Berners-Lee has put forward "self-describing" as a central design principle of the Web [2]. Yet, Tim Berners-Lee has noted that XML documents are not "self-describing" since they do not provide a description of their preferred method of processing or their meaning. The Semantic Web is meant to correct the latter, although as of yet nothing corrects the former. In one important sense, XML documents are self-describing; the element and attribute names provide a hint as to their usage, although this constraint is too informal to be enforced. An element can be called "brillig" just as easy as it can be called "date," and the XML document that contains these elements can still be well-formed and valid. The informal semantics of tag names rests upon the assumption that authors will actually use sensible names.

Tag names are not enough. If a document is shared between two or more parties, there is an assumption that the meaning of the document is implicit in some common understanding. The hints given by the element names are just useful reminders of this understanding. Every schema is a public agreement, and the details of this agreement are usually not described by the document itself or known to outside parties. Only a handful of microformats like RSS are well-understood by virtually "everybody on the Web." In these cases the meaning and usage of microformats are well-

```
cat input.xml |
xsv someschema.xsd |
saxon xhtmlout.xsl >
output.xml
```

Figure 1: UNIX Pipelines

```
<pipeline>

 <param name="target" select="result"/>

  <process id="p2" type="transform">
    <input name="stylesheet" label="xhtmlout.xsl"/>
    <input name="document" label="valid"/>
    <output name="result" label="result"/>
  </process>

  <process id="p1" type="validate">
    <input name="document" label="input.xml"/>
    <input name="schema" label="someschema.xsd"/>
    <output name="psvi" label="valid"/>
  </process>

</pipeline>
```

Figure 2: XPDL Example

known due to the clear and easily-accessible description of these formats and their widespread adoption. In fact, current progress on the Web is in a large part driven by the increasing usage and combinations of these microformats. Are all but the most popular of microformats doomed to be misunderstood? With additional processing and explicit semantics, the answer should be no.

When XML document types exploit namespaces, the opportunity for at least recording the intended understanding of names 'in' the namespace is available, but in practice we find that either no information at all is retrievable from a namespace URI, or the information which *is* available is neither standardised nor complete.

2. COMBINING DATA WITH PROGRAMS

2.1 Shell Scripts to Pipelines

One simple and well-known way of processing text files in UNIX is to arrange diverse UNIX components in a shell-script or "pipeline," with the input given by standard input and the output returned to standard output. This processing model is usually a strictly linear ordering of components, informally called "One Damn Thing After Another." Figure 1 is an example of a UNIX pipeline that does simple validation against a schema and then uses XSLT to transform the valid XML document into XHTML, using shell wrappers around components like Saxon to get them to conform to receiving input on standard input, and redirecting standard output to a file.

Many XML-based applications can be described in terms of pipelines. Instead of making one large, and often clumsy, program to accomplish a complex XML processing task, one breaks the larger problem into smaller sub-problems that can each be solved by specialized XML processors. These specialized processors then communicate by passing XML documents to each other in a linear order. As noted by Sean McGrath, this supports effective debugging and quick development time [16]. A host of XML pipelining products have appeared, with SUN and other XML pipeline vendors submitting the XML Pipeline Definition Language(XPDL) [27]. To translate our previous example in Figure 1 into XPDL, see Figure 2.

```
<page xmlns:fx="http://www.ltg.ed.ac.uk/~ht/functionalXML">
  <fx:transform xmlns:xi="http://www.w3.org/2001/XInclude"
             stylesheet="xmlhtmlout.xsl">
    <fx:validate schemaDocument="someschema.xsd">
      <document version="2.0">
        <head profile="http://www.w3.org/2003/g/data-view">
          <link rel="transformation"
            href="http://www.example.org/xml2rdf.xsl" />
          </head>
          <body>
            <title>Some Document</title>
              <author = "Harry Halpin" />
            <title>
              My input data is below!
            </title>
            <data>
              The <b>data itself</b>!
            </data>
          </body>
        </document>
      </fx:validate>
  </fx:transform>
</page>
```

Figure 3: *f*XML Example

2.2 From Pipelines to Functional Programming

XML pipelining languages do not make an XML document self-describing, since the processing instructions are outside the document itself, and so cannot take advantage of the natural compositionality of XML documents. In a recent paper [26], the same processing steps are easily put *inside* the XML document itself, by nesting the data to be processed inside an element that describes its processor. Arguments needed by the processors are given either as attributes or child elements of the processor. Therefore, certain elements of an XML document no longer describe data, but functions whose data is other elements and attributes in XML. Including the literal data of our initial input, the example can be rephrased using the *fx* namespace of this new theoretical language, called "functionalXML" or just "*f*XML," as shown in Figure 3. In this example a W3C XML schema-validation process is nested within an XSLT transformation process.

From a formal perspective, with their deterministic input and output, both XML pipelines and the simple functional expression thereof map onto finite state automata (FSA) that are guaranteed to terminate. There is a stark choice awaiting the use of the functional programming paradigm for XML processing. As one adds features such as variables, conditionals and functions, it is easy to lose FSA equivalence, and subtle interactions between features determine whether or not the language is Turing-complete. Once this has happened, the language is a full-blown programming language, theoretically equivalent in power to XQuery and Java! This path is already being pursued by XML pipeline languages such Orbeon's XPL [4] and NetKernel [21].

Note that the design choice of embedding pipelines in XML is far from controversial. The question of embedding XML processing languages in XML was at first held to stalwartly in early XML design, such as in XSLT and W3C XML Schema. However, more recent work in XML has gone in the other direction, attempting to simplify programming by keeping the syntax of the processing language in a non-XML compact notation, such as RELAX NG and XQuery. This viewpoint sees XML, instead of making the data more human readable, as making data more difficult to manipulate for humans. This viewpoint holds that XML syntax is fine for tree-structured data, but for complex programming XML syntax is a verbose overload that strains the finite memory of programmers. After all, programmers come well-equipped with parsers!

Even though we are proposing to directly embed a small programming language within arbitrary XML, we agree with the points made by XQuery, RELAX NG, and others. Indeed, no complex programming *should* be done within XML documents. However, pipeline languages are by definition *simple* languages that encapsulate more *complex* processors, and this *can* be done within XML. First, if the pipeline is chaining together other processors, in effect creating a meta-language for XML processors, it is overkill to create a pipeline using Java and underkill for the pipeline to be kept at the shell script level. One wants something powerful yet light-weight that neither extreme offers. Also, by exploiting the universality of the Infoset one can easily attach processing to XML documents in a way that takes advantage of the natural compositionality of XML. Lastly, by attaching the processors to XML documents, one makes the XML "self-describing" in a manner that sensibly and elegantly ties together current Web technologies, as we demonstrate in the rest of the paper.

2.3 Functional Programming in XML

With the chains of pipelining determinism broken, one should proceed carefully in order to keep it simple. This is especially relevant when transforming what was formerly a static XML Infoset into a Turing-complete program. XML programming languages have in general been going in a direction of increasingly complexity. XSLT 2.0 can now schema validate documents and includes a host of useful built-in functions. XQuery can search databases and provide type inference. This complexity makes life easier for professional programmers, but for novices this complexity can make using these languages difficult. Part of the genius of XML was that it initially was so simple that any computer-literate person could learn its main points in about ten minutes. The question is what are the most elementary yet most powerful programming language constructs that can create and combine processes within an XML document?

Variables, scope, and control structures are needed. Following Scheme, scoping can be introduced via the use of a *fx:let* construct, and variables can be bound within a scope by a *fx:bind* construct that binds data to a variable [25]. A *name* attribute of *fx:bind* can then be used to name the variable, and the variable can be distinguished from other data by the use of a dollar sign within the XML document. Control structures could be accomplished in a way similar to the XSLT *choose*, with a *cond* keyword to serve as a wrapper for *case…else* conditionals. These should natively incorporate XPath for their use test expressions. In our example given by Figure 4, we only transform an XML document if the version attribute is greater than 1.0.

There is no reason that a fXML variable could not hold a function itself, and there should be some process for creating new functions. This can be accomplished via the use of *fx:defun* to define functions, binding them to code and data via the use of a *fx:lambda* construction and giving them a grounding on the local machine via use of a *fx:processdef* element. Arguments can then be given using the children of an *fx:args* element of the *fx:lambda* element. This allows a name to be defined not only for XML data, but for a process such as GRDDL [12], as in Figure 5. In this example we check for a GRDDL attribute and do the transform to

```
<fx:let xmlns:fx="http://www.ltg.ed.ac.uk/~ht/functionalXML">
  <fx:bind name="myvariable">
    <fx:include href="document.xml"/>
  </fx:bind>
  <fx:cond>
    <fx:case test="$myvariable/document/@version &gt; 1.0">
      <fx:transform stylesheet="xhtmlout.xsl">
         <fx:validate schemaDocument="someschema.xsd">
           <fx:include href="$myvariable"/>
         </fx:validate>
      </fx:transform>
    </fx:case>
  </fx:cond>
</fx:let>
```

Figure 4: Variables and Binding in fXML

```
<fx:let xmlns:fx="http://www.ltg.ed.ac.uk/~ht/functionalXML">
  <fx:defun name="myGrddl">
    <fx:lambda>
      <fx:args>
        <fx:arg name="inputXML" />
      </fx:args>
      <fx:processdef grounding= "glean.py --output $stdout $inputXML"  />
    </fx:lambda>
  </fx:defun>
  <fx:bind name="myvariable">
    <fx:validate schemaDocument="someschema.xsd">
      <fx:include href="document.xml"/>
    </fx:validate>
  </fx:bind>
  <fx:cond>
    <fx:case test="$myvariable/head/link/@rel
                 &eq; &quot;transformation&quot;">
       <fx:apply name="myGrddl">
         <fx:include href="$myvariable" />
       </fx:apply>
    </fx:case>
    <fx:else>
      <fx:transform stylesheet="xhtmlout.xsl" />
        <fx:include href="$myvariable" />
      </fx:transform>
     </fx:else>
   </fx:cond>
</fx:let>
```

Figure 5: Defining Functions

RDF using our new function if such an attribute is found, and otherwise continue with our transformation to XHTML. Note that the function is defined using *fx:defun*, and then applied using *fx:apply*. These in principle should not differ from other functions such as *fx:transform* in their ability to take as their first argument their first child and so on, although an alternative attribute-driven syntax could also be used. "Standard output" is redirected to the document at hand, in traditional pipeline style.

It should be noticed that the very simple language being proposed here is very much like Lisp, or to be exact Scheme [25]. In fact, a compact notation for fXML can be given in a Scheme-like way. This is shown in Figure 6, which is just the example given in Figure 4 with a non-XML notation. Although neither Lisp nor Scheme has dominated industry, they are both recognized for being exceptionally elegant and extensible languages whose syntax belies their real power. As an added bonus, by following in their footsteps fXML maps onto the λ calculus formalism, one of the oldest and most well-understood formal foundations of programming languages.

This parallel suggests other constructs that could be added to fXML. First, some sort of error control similar to XPDL's **`error`** element or the **`try/catch`** mechanism could be used.

```
(let ((myvariable "input.xml"))
  (cond ([$myvariable/document/@version > 1]
          (transform (validate myvariable "someschema.xsd") "xhtmlout.xsl"))))
```

Figure 6: fXML compact notation

One could even imagine porting a number of constructions from Lisp or Scheme into *f*XML, for example an equivalent of `mapcar`. Each of these would have to be modified to make sense in terms of XML, which is at its most basic level more complex than S-Expressions [22]. Unlike S-Expressions, attributes are inherently unordered while elements are generally taken to be ordered. Yet even with only the minimal syntax presented so far, at this stage *f*XML overcomes one of the more cited "drawbacks" of both Lisp and Scheme. *f*XML, by using an XML-based syntax, overcomes the "curly bracket" paranoia that makes some hate Lisp, since *f*XML uses tag names to disambiguate brackets. It also increases the power of XML by providing a coherent way to easily embed processes in XML using a tried-and-tested paradigm that novices should find simple and experienced programmers familiar.

3. WEB SERVICES AS FUNCTIONS

3.1 Embedding Web Services

Is there a way to combine Web Services with XML documents easily? While many Web Services wrap their data in XML for transfer, and Web Services describe themselves in XML, it is shockingly difficult to include the results of a Web Service directly into an XML document. From a computational view, there are truly difficult complexities about policies, trust, and non-functional properties at the core of Web Services that are beyond the scope of this paper. A number of solutions exist that tackle this complexity explicitly, primarily BPEL (Business Process Execution Language) for process choreography, and are useful but aim at a different problem domain, that of allowing a business analyst to describe the interactions of processes across service end-points [8]. However, for pipelining and data integration, we would argue for hiding as much of this complexity away from the user as possible when dealing with Web Services. Many Web Services are at their core *functions that are available over the Web*. In most cases, what we are doing with Web Services is giving a program a URI so that it can be accessible on the Web.

Currently, we have assumed that all functions bound in *f*XML would be functions that are locally accessible. To extend this to Web Services one must allow another type of function, one that instead of being on the local machine is a remote service identified by a URI. The use and composition of Web Services, making many assumptions about trust and more, can be construed as analogous to function application and composition as done in functional programming. In our previous example as given by Figure 3, there is no reason why the XML Schema validator has to be local and can not be a Web Service.

In Figure 7 our XML Schema validation is not done locally, but by a Web Service that is defined as a function in a manner similar to local functions, using *fx:restWebService* instead of *fx:processdef*, with the URI of the Web Service given by the *location* attribute. In this example, we use a REST web service for validation that functions by using an HTTP POST with the document to be validated delivered as the payload, and the resulting valid XML document or error is accessed via HTTP GET on the URI of the Web Service [9]. This HTTP boiler-plate is automated by the grounding of the function in the *fx:restWebService*. This allows us to bind our validator *wsValidate* to a Web Service.

```
<fx:let>
  <fx:defun name="wsValidate">
    <fx:lambda>
      <fx:args>
        <fx:arg name="inputXML" />
        <fx:arg name="schemaDocument" />
      </fx:args>
      <fx:restWebService
        location="http://www.example.org/SchemaValidator" />
     </fx:lambda>
  </fx:defun>
  <fx:transform stylesheet="xhtml.out">
    <fx:apply name="wsValidate"  schemaDocument="someschema.xsd">
      <fx:include href="input.xml"/>
    </fx:apply>
  </fx:transform>
</fx:let>
```

Figure 7: Web Services Inside Documents

3.2 REST and SOAP

The debate between REST and SOAP has engulfed Web Services for some time, and we would not want it to also engulf *f*XML. It would be better to imagine a continuum of Web Services, with one end having a complex layering of security and orchestration that standards like those in the WS stack provide [7], and the other end of the spectrum allowing the relevant data to be accessed by using HTTP in a RESTful manner [9]. These, and any other way of retrieving XML remotely over the Web, can be incorporated easily in *f*XML by adding additional grounding definitions to functions that automate the boilerplate needed. A *soapWebService* should automatically wrap the input and unwrap the output using a SOAP envelope, and also use the location of a WSDL file so that operations as defined by ports and operations given by a WSDL can be used.

3.3 Scripting as Functions

Another popular technology is AJAX, yet another development in scripting. In a nutshell, AJAX is the use of Javascript to asynchronously update XML on the client from the server using the XMLHttpRequest object. However, as others have pointed out, the use of Javascript in this technology is arbitrary and mostly driven by the fact that Javascript is the only client-side programming language that is *nearly* universally available in most recent browsers. In the case of AJAX, all user behaviors that normally would invoke a server response are captured and processed by the AJAX engine, and channeling user behavior comes at the cost of burdening the web page with an immense amount of Javascript. This can leave arbitrary DOM nodes up for modification by Javascript. The use of Javascript is just one example of how scripting languages are currently mixed with XML, and server-side scripting such as PHP is another trenchant example. Usually these various scripting technologies use *script* elements or processing instructions to put their code directly into an XML document, often leading the interesting parts of XML documents to be veritable black holes.

There is no reason that the advantages of scripting cannot also be incorporated into an XML document itself using a functional approach. This would force encapsulation and abstraction over the script engine(s) as the relevant functions would be invoked via `fx:apply` in *f*XML. "View Source" can be comprehensible again, as long as one is kept to the XML at the right stage of processing. A scripting application could be described as binding Javascript and PHP to a particular elements in an XML document, allowing user behavior to asynchronously update *that and only that* ele-

```
<body xmlns:fx="http://www.ltg.ed.ac.uk/~ht/functionalXML">
  <fx:defun name="myMap">
    <fx:lambda type="javascript">
      <fx:script location="ajax_access_map.js"/>
    </fx:lambda>
  </fx:defun>
  <fx:defun name="myDirections">
    <fx:lambda type="php">
      <fx:include href="directions.php" />
    </fx:lambda>
  </fx:defun>
  <h1>Welcome to My Web Page</h1>
  <p>Get <b>directions</b> to my location type either my
work or home address here:</p>
  <fx:apply name="myDirections"/>
  <p>You may find this <b>map</b> useful as well!</p>
  <fx:apply name="myMap"/>
</body>
```

Figure 8: Scripts as Functions

ment and encouraging modularization of scripting code as is already good practice. The elements can then be typed with their script engine, allowing any processing instructions or script tags to be generated by fXML. In Figure 8 we present an application that uses PHP to get an address from the user and then provides a Javascript-enabled map of the area around the address, but abstracts away the scripting by hiding it in fXML elements. Server-side scripting can be encapsulated via XInclusion and client-side scripting given a function as shown before. While this may not provide any increased functionality for the web page, it does allow the XML code to be inspected and possibly reasoned about in a functional manner.

4. THE SEMANTIC WEB AS TYPES

4.1 The Semantic Web for Data Integration

The Semantic Web is a solution in need of a problem. The problem of the hour for the Semantic Web is *data integration*. Seamless data integration is the holy grail of XML. While XML provides a well-understood syntax for the exchange of data, unless two parties have a very clear idea of what XML they are exposing or capable of receiving, encoding data in XML will under no circumstances magically cause data to integrate. The best way for data to integrate is for all the data to have compatible abstract models regardless of their particular syntax, as could be given by a model theory of the data using formal semantics. This is precisely what the Semantic Web offers. Only when the data in two XML documents can be shown to be compatible in their models can we be assured that we can combine the two sources of data safely, such that I know your tag name for "lastName" is the same as my tag name for "surname." In the "open world" of the Web, unlike in traditional "closed world" databases, one has no idea what data might attempt to be integrated with what other data. Knowledge representation systems based on describing taxonomies of data and their relationships, such as description logics and first-order logic (the upcoming Semantic Web rule language), are in general based on well-tested and developed methods from artificial intelligence that can make the semantics of data explicit [11, 3, 13]. However, for many cases the mapping of XML to RDF is the key to giving XML documents a formal model semantics that operates outside the level of a single XML document. After converting relevant parts of the XML document to RDF, the merging of RDF graphs can then be a step towards data integration on the Web. After the data integration is complete on the level of RDF, one might be able to serialize the results back as an XML document that provides a link to a transformation or a mapping back to the RDF graph. However, this final step is currently not even the topic of much research or approaching being standardized. Only once this step is done can we successfully "round-trip" from XML to RDF and back again. Yet for the applications described here this final step is not needed.

The use of RDF to model the semantics of a XML vocabulary has pleasant ramifications for both XML and the Semantic Web. First, XML infosets and the Semantic Web should be viewed not as competing paradigms but as complementary efforts. XML is a great syntax for transferring just about anything over the Web, since tree models are well-studied in computer science and naturally fit onto much of the data that the Web traffics in, and since raw XML can be deciphered by humans, unlike the XML serialization syntax for RDF. As more and more industries move towards using XML as the *de facto* standard for transferring data, asking them to forsake their investment in their own XML formats is wishful thinking at best. However, there is a compelling case for using semantics for data integration, and while XML provides no application semantics as such, RDF at least enables it to be made explicit [18]. So the Semantic Web, or *something like it*, is what is needed to provide the formal semantics for data integration. However, instead of waiting for people to explicitly encode their data using Semantic Web standards, the Semantic Web needs to find ways to layer itself onto XML.

There are two main methods for integrating the Semantic Web with what one might call *vernacular* XML (the myriad of custom XML vocabularies not given in RDF), one declarative and the other procedural. Both can be made to work rather easily with fXML. The declarative model imports mapping attributes into either an XML Schema or directly into the XML document itself [14]. This sort of mapping has been receiving increased attention lately, as exemplified by WSDL-S [23]. However, usually the mapping from XML data to the Semantic Web is not a trivial case of correspondence between Semantic Web classes and properties to vernacular XML elements and properties, but instead is dependent on a variety of factors within the XML document. In this case a *procedural* model, called GRDDL (Gleaning Resource Descriptions from Dialects of Language) provides a mapping from vernacular XML data to XML-encoded RDF, often via an XSLT stylesheet [12]. In essence, a pointer to a transformation process can be given as the root element in a document. Since this methodology is so straightforward, it has been gaining in popularity rapidly. Due to the fact that the results of the first approach can be, and of the second already are, expressed as XML infosets, fXML can easily incorporate both the declarative and procedural approaches. In the first case, the mapping from a Semantic Web ontology could be given either by an explicit reflected Post-Schema Validation Infoset and a generic PSVI to RDF-in-XML transformation [15]. In the second case fXML can be used to evaluate the GRDDL binding explicitly, as we illustrate in Figure 9. Either way, with fXML allows one to use the methodology of one's choice in connecting an ontology to an XML document. In Figure 9 we use fXML to call GRDDL to produce RDF-in-XML for an XML document and merge the result with the interpretation of a pre-existing RDF-in-XML document level. Therefore, at its most basic "semantic" fXML (given by the *sfx* namespace)

```
<document xmlns:sfx="http://www.ibiblio.org/hhalpin/sfXML"
          xmlns:fx="http://www.ltg.ed.ac.uk/~ht/functionalXML">
  <sfx:grddl>
    <fx:include href="neworder.xml" />
  </sfx:grddl>
  <fx:include href="bookdata.rdf" />
</document>
```

Figure 9: Data Integration with RDF in XML

```
<fx:let xmlns:sfx="http://www.ibiblio.org/hhalpin/sfXML"
        xmlns:storeOnt="http://www.example.org/StoreOntology/"  >
  <sfx:bind name="myPerson">
    <sfx:grddl>
      <fx:include href="customerOrder121.xml" />
    </sfx:grddl>
  </sfx:bind>
  <p>Thank you for your purchase.
    <fx:cond>
      <sfx:case "$myPerson &eq; storeOnt:FrequentCustomer" >
        You will receive a <b>discount</b> for
        your loyalty to the store!
      </sfx:case>
      <sfx:else>
        Shop more and you'll receive <b>special</b>
        discounts!
      </sfx:else>
    </sfx:cond>
  </p>
</fx:let>
```

Figure 10: Using the Semantic Web in *sf*XML

```
<fx:let xmlns:sfx="http://www.ibiblio.org/hhalpin/sfXML"
        xmlns:storeOnt="http://www.example.org/StoreOntology/"  >
  <sfx:bind name="myPerson" type="storeOnt:StoreCustomer">
          <fx:include href="customerOrder121.xml" />
  </sfx:bind>
</fx:let>
```

Figure 11: Ontologies as Types in *sf*XML

can be considered mappings of infosets to triples (such as the case with GRDDL) and the mapping of triples to triples, as opposed to mappings of infosets to infosets. When we can use triples and infosets together we gain the advantages of both, especially if we can hide the details at a high level of abstraction.

4.2 The Semantic Web as Types

Both type theories and ontologies classify data abstractly. If XML is data, then can the Semantic Web provide types for that data? However, isn't this what XML Schema is supposed to do? If the XML document has been schema validated, additional data such as the type of the XML data is given by the Post Schema-Validation Infoset [22], and this type information can easily be used to export the types found in most conventional databases and programming languages [15]. On one hand, XML Schema does provide the types needed if those types happen to be "integers," "strings," and the like. Traditionally in programming languages like C this is all one needs. Even in languages with a rich type system like Haskell, the purpose of a type system is to be a "syntactic system for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute" [19]. Calling the variable *firstName* (as in *firstName* equals "Harry") a *string* is important, and like an XML tag, the decision to name a variable "*firstName*" is meaningless to the compiler in of itself, which just considers it another variable that can have as its value a string. However, on the Web it may be important that a string is a *firstName* as in "the first name of a person," and not just any string. Moving from the closed-world programming paradigm to the open-world programming paradigm, it makes sense to explicitly model the *world* semantics of your data, not just give them *data* types. By *world semantics* we mean the denotations of a symbol in the world outside the computational process, while by *data types* we mean the classification of possible behaviors of data inside the computation. For example, the world semantics of "Pat Hayes" is a particular person who was born in Newent in the United Kingdom, yet this and innumerable other facts are just not relevant to the possible behaviors of the string "Pat Hayes" (like being concatenated with other strings) inside a computational process. On the Web, one is more likely to determine whether or not a computation is valid based on what the data represents in the world outside the Web rather than its data type alone. This is based on the distinction between using ontologies as world semantics, where we model the world semantics with a formal model [1], and data types, where we model the computational process itself [24]. Yet both are methods of enforcing "patterns of abstraction" [20].

For example, when determining if a discount should be applied to Pat Hayes when he is shopping at an online store, one needs to know a lot beyond data types. One needs to know if "Pat Hayes" is a *person*, not merely a string that can be equivalent to other strings and so on. In Figure 10, we want to know if Pat Hayes has bought enough books online store as to qualify for a "frequent customer" discount. This type of world semantics can be encoded by using the Semantic Web, e.g. by having *Pat Hayes* belong to class *StoreCustomer*, subclass of *Person.* The question can then be reframed as does *Pat Hayes* also belong to the class *FrequentCustomer*, subclass of *StoreCustomer*? Our example shows how *sf*XML can determine this by using GRDDL to transform the data into triples and then explicitly checking the triples for *FrequentCustomer.* This checking is done via an *sfx:cond* and *sfx:case* statement that uses an "equality" test to test whether or not a triple of the needed class (or possibly property) is given in the tested triples. We can construe subclass and subproperty relationships as analogous to subtyping relationships. In which case, the *type* attribute allows us to bypass the boilerplate coding and "equality" test. The *type* attribute just automates the RDF transformation of the XML input and checks whether the results are valid based on a check for the wanted class (the "type"). If this test fails, the assignment to the variable fails as an invalid type of data has been assigned to a typed variable. This simplification is shown in Figure 11.

In other words, while XML Schema gives us types that tell us how data should be encoded in XML, the Semantic Web can give us types that tell us what things in the real world our data represents. In that regard, the Semantic Web can be considered as a type system for the real world. Unlike most type systems for traditional programming languages, the Semantic Web is an open-world system.

4.3 Semantic Web Services as Typed Functions

If the Semantic Web is an open-world system that can be used as types, and Web Services are functions available on the Web, then Semantic Web Services can be construed as typed functions for the Web. This analogy is useful, for it provides the ability to invoke Web Services not just by their

```
<body>
  <fx:let xmlns:fx="http://www.ibiblio.org/hhalpin/sfXML"
        xmlns:dirOnt="http://www.example.org/DirectionsOntology/">
    <sfx:bind name="myAddress" type="dirOnt:Address">
      <sfx:include href="myaddr.rdf"/>
    </sfx:bind>
    <sfx:bind name="yourAddress" type="dirOnt:Address">
      <sfx:include href="your_addr.rdf"/>
    </sfx:bind>
    <h1>Welcome to My Web Page</h1>
    <p>The <b>directions</b> to my House are here:</p>
    <sfx:defun name="doDir">
      <sfx:lambda type="dirOnt:Directions">
        <sfx:args>
          <sfx:arg name="origin" type="dirOnt:Origin"/>
          <sfx:arg name="destination" type="dirOnt:Destination"/>
        </sfx:args>
      </sfx:lambda>
    </sfx:defun>
    <sfx:defun name="doMap">
      <sfx:lambda outputType="dirOnt:Map">
        <sfx:args>
          <sfx:arg name="address" type="dirOnt:Location"/>
        </sfx:args>
      </sfx:lambda>
    </sfx:defun>
    <sfx:apply name="doDir" origin="$yourAddress" destination="$myAddress" />
    <p>You may find this <b>map</b> useful as well!</p>
    <sfx:apply name="doMap" address="$myAddress"/>
  </fx:let>
</body>
```

Figure 12: Semantic Web Services

URI, but by the type of information they have. This type system can then invoke service composition machinery and discovery processes [17]. The implementation details of this is a hard problem in of itself, but for a user discovery should be transparent.

For example, one may not even know a particular Web Service that provides directions from one street address to another. Web Services have an irritating propensity to disappear, so even when one has a particular URI for this sort of Web Service it is unwise to rely upon that URI solely. Almost all Web Services providing driving directions will have as input parameters the two street addresses, and as their output a set of driving directions. Assuming a fully operational Semantic Web, these Web Services can categorize themselves as Semantic Web Services and type their input and output to commonly available ontologies via a mapping from WSDL to RDF as given by WSDL-S or an equivalent technique [23]. If we have a *Directions* class and an *Address* class in an ontology of driving directions, then a *Directions Semantic Web Service* would take as its input two arguments of class *Address*, one being also a member of the *Origin* class and the other a member of the *Destination* class. Likewise one can imagine a *Map* class for map images, and we also need an *Address* to find a map. Building on our previous scripting example, a new Semantic Web Service framework is sketched Figure 12. By binding the Web Services with types, we make sure the XML document used as input has a triple mapping to the input type, and the output document has a triple mapping to the output type. For type checking, the transformation from XML to triples for the input document could be done automatically by sfXML, and it could be done for the output via an explicit mapping like WSDL-S for the output of a Web Service. Assuming the type checking has been successful, the output assumes the results of the service are given in vernacular XML, not the RDF triples themselves. This allows the output to then be easily included back into the original XML document that invoked the service.

In defining Web Services purely in terms of their input and output type signatures using the Semantic Web, we consider the Web Service needed to be an *anonymous* function that gets bound to an actual Web Service when a Semantic Web Service for the actual service has input and output types that match the desires ones. While the Web Service discovery process can be complex, the ordinary use of Semantic Web Services may be even easier than using Web Services without Semantic Web types, since using Semantic Web Services should let one just specify *what one wants* instead of *how to get it*. This process could also inter-operate with existing initiatives that allow the binding of Semantic Web ontologies to Web Services [23], and in fact can directly rely on these methods to provide the types for functions. Semantic fXML can serve as a wrapper for various Semantic Web Services technologies, and put the result of these sometimes obscure and hard-to-use technologies directly into an XML document.

5. ARCHITECTURAL VISION

5.1 The Web is a Computer

The Web is currently faced with the danger of being fragmented between XML, Web Services, the Semantic Web, and the "Web 2.0" initiative, as exemplified by AJAX and microformats. Instead of attempting to argue that one of these initiatives is superior to the others, we instead present a unified abstraction of data, types, and functions to unify the initiatives at a suitable level of abstraction: XML encodings of a functional programming language for infosets, and an extension for it to deal with triples and the use of triples as analogous to type checking. Although it would have been perhaps easier to explain this abstraction in theoretic terms, we ground our examples using two XML vocabularies, fXML and sfXML, illustrating how such a unified viewpoint can be used to build the next generation of the Web solidly upon the success story of XML. As should be clear, what the functionalXML perspective presents is a vision of the Web that is remarkably simple: data, types, and functions all accessible over the Web and composed within XML documents. Given this framework, one can treat the Web itself as one computer [10]. The Web is moving from a universal information space to a universal computation space.

5.2 Turing's Promise

As explored by the philosopher Andy Clark, one criteria used to judge a system as an unitary object is the latency between its potential components [6]. As latency decreases, we are likely to regard two components as part of the same system. For example, because there is such low latency between your hand and your brain, you naturally regarded your hand as part of yourself. As more feedback with less latency is received from artifacts like a laptop, it is not unreasonable for people to consider these artifacts to be almost part of their selves, to the point that when when a laptop breaks down it as if part of the self has been disabled. There can be an almost a physical feeling of pain [6]. In the same manner, one argument against both the Web and Web Services is that the process of wrapping data in *http* and formatting it in verbose XML increases latency. However, as network infrastructure decreases latency, the distinction between what is "local" to a machine and what is "on the Web" will become less and less noticeable to the user, even when the data is encoded in XML and wrapped in *http*.

In that regard, as latency decreases, the difference between the local computer and the Web dissipates. Therein lies the true power of Web Services. Once the Web has matured past a certain point, there should be no noticeable difference between functions on the Web and functions available locally on a hard disk. Before ASCII, computers had their own non-universal codes for storing data such as numbers and letters, making it difficult to exchange data between computers. The same problem faces structured data on the Web, and XML is currently provides one standard for making complex data truly universal. As more and more computation moves to the Web, the combination of XML and Unicode will doubtless replace ASCII as the data format of choice. Web Services are already becoming the de facto way to access programs over the Web. Add to this the ability to make what our data means explicit to both humans and machines through using the Semantic Web, and one has the Web functioning as one open-ended, vast, decentralized computer.

Turing proved long ago that all computers instantiate one universal formal model of computation, the Turing machine, which the Church-Turing Hypothesis notes is equivalent to the λ calculus that fXML is built upon [5]. However, it is the details of incompatible hardware and idiosyncratic software that has made our data and programs unable to be universally shared. With the advent of the next stage of the Web, the software can finally live up to the promise implicit in Turing machines: the Web will finally serve as one universal computer, not just in theory, but in practice.

6. REFERENCES

[1] J. Barwise and J. Perry. *Situations and Attitudes.* MIT Press, Boston, Massachusetts, 1983.

[2] T. Berners-Lee. *Weaving the Web.* Harper, San Francisco, 1999.

[3] H. Boley and S. Tabet. Design Rationale for RuleML: A Markup language for Semantic Web Rules. In *Proceedings of Semantic Web Working Symposium*, 2001.

[4] E. Bruchez and A. Vernet. XML Pipeline Language (XPL) Version 1.0. Member submission, W3C, 2005.

[5] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40–41, 1936.

[6] A. Clark. *Natural Born Cyborgs.* Oxford University Press, 2004.

[7] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, and N. Mukhi. Unraveling the Web Services Web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6, 2002.

[8] F. Curbera, R. Khalaf, N. Mukhi, S. Tai, and S. Weerawarana. The next step in Web Services. *Communications of the ACM*, 6(10):29–34, 2003.

[9] R. Fielding and R. Taylor. Principled design of the modern Web architecture. In *Proceedings of International Conference on Software Engineering*, Toronto, Canada, 2001.

[10] H. Halpin. The Semantic Web: The origins of AI redux. In *the Proceedings of the Fourth International Workshop on the Philosophy and History of Mathematics and Computation*, San Sebastien, Spain, 2004.

[11] S. Hawke and S. Tabet. Workshop for Rule Languages for Interoperability. Report, W3C, 2005.

[12] D. Hazael-Massieux and D. Connolly. Gleaning resource descriptions from dialects of language. In *Proceedings of XTech*, Amsterdam, Netherlands, 2005.

[13] I. Horrocks, P. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web rule language combining OWL and RuleML. Member submission, W3C, 2004.

[14] J. Hunter and F. Nack. Combining RDF and XML Schemas to enhance interoperability between metadata application profiles. In *Proceedings of International World Wide Web Conference*, 2001.

[15] A. Krupnikov and H. Thompson. Data binding using W3C XML Schema Annotations. In *Proceedings of the XML Conference*, Orlando, USA, 2001.

[16] S. McGrath. XML Pipelines. In *Proceedings of XML Open*, Cambridge, UK, 2004.

[17] S. Narayanan and S. McIlrath. Simulation, verification and automated composition of web services. In *Proceedings of the World Wide Web Conference*, Honolulu, USA, 2002.

[18] P. Patel-Schneider and J. Simeon. The Yin-Yang web: XML syntax and RDF semantics. In *Proceedings of the World Wide Web Conference*, Honolulu, USA, 2002.

[19] B. Pierce. *Types and Programming Languages.* MIT Press, Boston, USA, 2002.

[20] J. Reynolds. Types, abstraction, and parametric polymorphism. In *Proceedings of Information Processing Conference*, Amsterdam, the Netherlands, 1983.

[21] P. Rodgers. Service-oriented-development on netkernel. In *Web Services Edge East*, Boston, USA, 2005.

[22] J. Simeon and P. Wadler. The essence of XML. In *Proceedings of ACM Symposium on Principles of Programming Languages*, New Orleans, USA, 2002.

[23] K. Sivashanmugam, K. Verma, and A. Sheth. Discovery of Web Services in a federated registry environment. *Proceedings of IEE Second International Conference on Web Services*, 2004.

[24] B. C. Smith. The Correspondence Continuum. In *Proceedings of the Sixth Canadian Conference on Artificial Intelligence*, Montreal, Canada, 1986.

[25] G. Steele and G. Sussman. Lambda: The Ultimate Imperative. AI Lab memo aim-353, MIT, 1976.

[26] H. Thompson. FunctionalXML. Member submission, W3C, 2005.

[27] N. Walsh and E. Maler. XML Pipeline Definition Language. Note, W3C, 2002.