

Scalable Programming Abstractions for XML Services

Burak Emir, Sebastian Maneth*, and Martin Odersky

School of Computer and Communication Sciences

EPFL, CH-1015 Lausanne, Switzerland

burak.emir@epf.ch, sebastian.maneth@nicta.com.au, martin.odersky@epfl.ch

Abstract. Traditional programming paradigms and styles do not lend themselves easily to XML services. This has led to engineered systems that are characterized by a mix of special purpose and general purpose languages. Such systems are brittle, hard to understand and do not scale well - hence they are not dependable. We describe some facets of the Scala programming language targeted at XML services that unify the disparate worlds through a judicious combination of existing and new programming language constructs. More concretely, we describe use cases of case classes, regular pattern matching and comprehensions. Programs that use these abstractions can deliver XML services in a scalable and manageable way. We discuss the essential design decisions we took, the experience we gained during development, and identify directions of further research.

1 Introduction

Service-orientation is an emerging paradigm that promises a deep impact on programming. Similar to the rise of object-oriented systems when graphical user interfaces became the norm, service-oriented systems are motivated by two strong trends: the move from single-machine, homogeneous execution environments to distributed and heterogeneous ones, and the move from fine-grained and proprietary transmission formats to coarse-grained, standardized and semistructured ones.

The reasons behind this paradigm shift lie in new challenges posed by programming applications and services for the internet. For the last 20 years, the most common programming model has been object-oriented: System components are objects, and computation is done by method calls. Methods themselves take object references as parameters. This is a beautifully simple abstraction, which describes computation adequately as long as we are dealing with a single computer. At first, it seems that the concept of remote method calls lets one extend this programming model to distributed systems. However, this approach does not scale up well to wide-scale networks where messages can be delayed and components may fail.

Web services address the message delay problem by increasing granularity, using method calls with larger, structured arguments, typically represented as XML data. They address the failure problem by using transparent replication and avoiding server state. Conceptually, they are *tree transformers* that consume incoming message documents and produce outgoing ones.

Should this paradigm shift have an effect on programming languages? There are at least two arguments that suggest this: First, today's object-oriented languages are not very good at analyzing and transforming structured data, such as XML trees. Since such trees usually contain only fields but no methods, they have to be decomposed and constructed from the "outside", that is from code that is external to the tree definition itself. In an object-oriented

* Present address: National ICT Australia, Kensington NSW 1466, Australia

language, the ways of doing so are limited. In the most common solution (characterized by W3C's Document Object Model [16]), all tree nodes are values of a common type. This makes it easy to write generic traversal functions, but forces applications to operate on a very low conceptual level, which often loses important semantic distinctions present in the XML data. More semantic precision is obtained if different internal types model different kinds of nodes. But then tree decompositions require the use of run-time type tests and type casts to adapt the treatment to the kind of node encountered. Such type tests and type casts are generally not considered good object-oriented style. They are rarely efficient, and not easy to use.

By contrast, tree transformation is the natural domain of functional languages. Their algebraic data types, pattern matching and higher-order functions make these languages ideal for the task. It's no wonder, then, that specialized languages for transforming XML data such as W3C's XSLT [16] are functional.

The second reason for the popularity of functional languages in web-service programming is the fact that handling mutable state is problematic in this setting. Components with mutable state are harder to replicate or to restore after a failure. Data with mutable state is harder to cache than immutable data. Functional language constructs make it relatively easy to build components without mutable state.

Many web services are constructed by combining different languages. For instance, a service might use XSLT to handle document transformation, XQuery for database access, and Java for the "business logic". The downside of this approach is that the necessary amount of cross-language glue can make applications cumbersome to write, verify, and maintain. A particular problem is that cross-language interfaces are usually not statically typed. Hence, the benefits of a static type system are missing where they are needed most – at the join points of components written using different paradigms.

Conceivably, the glue problem could be addressed by a "multi-paradigm" language that would express object-oriented, concurrent, as well as functional aspects of an application. But one needs to be careful not to simply replace cross-language glue by awkward interfaces between different paradigms within the language itself. Ideally, one would hope for a fusion which unifies concepts found in different paradigms instead of an agglutination, which merely includes them side by side. This fusion is what we try to achieve with the Scala programming language [13].

Scala is both object-oriented and functional. It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Classes can be composed using mixin composition. Scala is designed to work seamlessly with two mainstream object-oriented languages – Java and C#.

Scala is also a functional language in the sense that every function is a value. Nesting of function definitions and higher-order functions are naturally supported. Scala also supports a general notion of pattern matching which can model the algebraic types used in many functional languages. Furthermore, this notion of pattern matching naturally extends to the processing of XML data.

In this paper, we focus on the elements that make Scala suitable for programming web services and applications. These are in particular its support for pattern matching, with its specialization to XML data, as well as its support for higher-order functions, with for-comprehensions as a convenient front-end syntax for querying. Other innovations of Scala, which have to do with component abstraction and composition, are described elsewhere [14].

Related Work

There has been extensive research in general purpose languages that tackle data integration and concurrency in innovative ways, yet retain some form of static type safety.

The designers of the research language Mawl [2] have introduced a notion of forms and a notion of session with the goal to produce valid HTML services. Their form language is a custom extension of HTML, and the language ensures that only valid HTML is sent to the client.

Sessions were taken up by the authors of Bigwig [5] and its successors J Wig [6]. While the former is aimed at HTML, the latter provides more general XML transformations, while keeping the statically checked validity guarantees. Widespread use of J Wig seems however inhibited because of its reliance on a particular validation language.

Bierman et al. propose $C\omega$ [4] an object-oriented language in which an overloaded dot operator can be used for XPath like querying. Distribution is approached by chords, which in turn are based on the join calculus. This language furthermore integrates Benton et al.'s concurrency abstractions for C# [3].

XJ [11] is an extension to Java that aims at binding types from XML schemata and providing XPath primitives in an imperative context. XPath expressions are used for bulk updates, and a combination of static analysis techniques and runtime checks is used to guarantee type safety.

Xtatic [10] is an extension to C# that introduces regular expression types and sophisticated runtime representation of XML values. Programmers can take advantage of the underlying .NET concurrency model.

Our work in data binding is similar to the approach Wallace and Runciman take for Haskell XML integration [17]. The authors create specialized type definitions from a given DTD.

The W3C has standardized the XSLT and XQuery languages [16] for transformation and querying of XML. The former was intended as a language for simple transformations but quickly grew beyond its initial goal. The latter is designed as the XML equivalent of the database-hosted structured query language (SQL). Both languages are functional in the sense that they do not permit imperative update of XML trees.

The rest of this paper is structured as follows. Section 2 gives an introduction to case classes and pattern matching; these features, known from the functional programming domain, are useful for handling algebraic datatypes and structured data in particular. Section 3 is the core section of this paper; it describes the XML library of Scala. Starting with the XML data model in Scala, it describes parsing and validation, regular sequence pattern matching, and querying XML using for comprehensions. Section 4 shows two examples of simple XML services, and their implementation in Scala. When dealing with more advanced service architectures one will have to deal with concurrent requests and events. Scala's programming abstractions for concurrency are shortly described in Section 5. Section 6 concludes.

2 Case classes and pattern matching

In this section we start by recalling the conventional, object-oriented way of decomposing data. Then we introduce case classes and describe the semantics of pattern matching. Finally, regular sequence patterns are discussed; as will be seen in the next section, these are particularly useful for decomposing XML data.

2.1 Object-Oriented Decomposition

When dealing with structured data, a common object-oriented design pattern is to create a set of classes, some of which act as structural containers for others (cf. the Composite pattern [9]). For inspecting structured data, a programmer can solely rely on virtual method calls of methods provided by such classes.

As an example, consider a simple evaluator for propositional logic, consisting of propositional variables and connectives. We can decompose the evaluator according to the term structure as follows:

```

trait Term {
  def eval(env: Array[Boolean]): Boolean;
}
class True() extends Term {
  def eval(env: Array[Boolean]) = true;
}
class Var(i: Int) extends Term {
  def eval(env: Array[Boolean]) = env(i);
}
class Not(term: Term) extends Term {
  def eval(env: Array[Boolean]) = !term.eval(env);
}
class And(left: Term, right: Term) extends Term {
  def eval(env: Array[Boolean]) = left.eval(env) && right.eval(env);
}

```

The given program models propositional formulas with an abstract class (called *trait*) Term which defines a deferred eval method that takes an environment as a parameter. Concrete subclasses of Term model the various term variants. Note that the compiler has enough information to infer the result type, it is thus omitted for brevity.

This object-oriented decomposition scheme generally requires the anticipation of all operations traversing a given structure. Moreover, non-local inspections cannot be implemented by one method alone, several dispatches are necessary. Often internal methods have to be exposed to some degree. Adding new methods is tedious and error-prone, because it requires all classes to be either changed or subclassed. A related problem is that implementations of operations are distributed over all participating classes making it difficult to understand and change them.

The Visitor pattern [9] can be used to separate operations from structure, however, it still breaks encapsulation, it does not deal with non-local inspections either and requires significant amounts of boilerplate code.

2.2 Pattern Matching Over Class Hierarchies

Functional languages like ML and Haskell have embraced *algebraic datatypes* for the purpose of separating structure from operations. Operations on such datatypes are simply functions which use *pattern matching* as the basic decomposition principle. Such an approach makes it possible to implement a single eval function without exposing artificial auxiliary functions.

Scala provides a natural way for tackling the above programming task in a functional way by supplying the programmer with a mechanism for creating structured data representations similar to algebraic datatypes and a decomposition mechanism based on pattern matching.

Instead of adding algebraic types to the core language, Scala enhances the class abstraction mechanism to simplify the construction of structured data. Classes tagged with the **case** modifier automatically define a constructor with the same arguments as the primary constructor. Singleton objects (or objects, for short) are classes that have only one instance, hence serve as constants:

```
abstract class Term ;
case object True extends Term ;
case class Var(i: Int) extends Term ;
case class Not(term: Term) extends Term ;
case class And(left: Term, right: Term) extends Term ;
```

Given these definitions, it is now possible to create the propositional formula $x_1 \wedge \neg(\neg x_2 \wedge x_3)$ without using the **new** primitive, simply by calling the constructors associated with case classes: `And(Var(1), Not(And(Not(Var(2)), Var(3))))`. The fields of each case class can be accessed with the usual dot notation, e.g. as in `x.left`. Furthermore, Scala's pattern matching expressions provide a concise means of decomposition that uses these constructors as patterns:

```
def eval(term: Term, env: Array[Boolean]): Boolean = term match {
  case True          => true;
  case Var(i)        => env(i);
  case Not(t)        => !eval(t, env);
  case And(left, right) => eval(left, env) && eval(right, env); }
```

Note that apart from the pleasing localization of the intended behavior in one method, there is now a way to change the representation of environments to bitfields without needing to touch the source code of the Term classes. Moreover, it becomes easy to perform non-local inspections through the use of nested patterns as happens in the following function.

```
def simpl(term: Term): Term = term match {
  case True | Var(_)    => term
  case Not(Not(x))     => simpl(x)
  case Not(x)          => Not(simpl(x))
  case And(left, right) => And(simpl(left), simpl(right)) }
```

The matching expression `x match { case pat1 => e1 case pat2 => e2 ... }` matches value x against the patterns pat_1, pat_2 , etc. in the given order. The value x is called the *scrutinee*. A pattern pat_i is a term built up from variables, case class constructors, and some predefined match primitives (like the wildcard `_` and the choice operator `|`). To match the scrutinee against pat_1, pat_2 , etc. means to find the first pattern pat_i that can be made equal to x by binding its variables to terms appropriately. If such a pattern is found then e_i is executed, after replacing in e_i each variable of pat_i by its binding. If no such pattern is found, then a run time error is generated. The wildcard pattern `_` matches any value. Choice patterns $p_1|p_2$ may not contain variables and match the union of values that are matched by their subpatterns. A constructor pattern $Constr(p_1, \dots, p_n)$ matches any value that is an instance of the corresponding case class, and whose arguments match the arguments patterns. Hence, patterns can be terms of arbitrary depth (cf. the pattern `Not(Not(x))` in the function `simpl`).

Such a functional decomposition scheme has the advantage that new functions can be added easily to the system. On the other hand, integrating a new case class might require changes in all pattern matching expressions. For extensibility, the Scala compiler does not check *exhaustiveness* of patterns, meaning to ensure that any scrutinee must match at least one of the patterns.

For the moment it neither checks *redundancy*, i.e. when a pattern can never match due to a more general earlier pattern. We plan to add redundancy checks in the future.

Patterns in Scala are linear in the sense that a variable may appear only once within a pattern. However, it is possible to add to a pattern p a “guard”, i.e., an **if** expression that involves the variables of pat . For instance, the pattern **case** And(x , y) **if** $x == y$ matches only terms of the form And(t , t). Hence, using such equality guards it is possible to express arbitrary constraints between variables.

The **case** modifier can appear anywhere on a class hierarchy. It is possible to extend a case class, and case classes can extend non-case classes, which enables more involved designs than the flat one presented above. The only restriction that applies is that a case class may not directly or indirectly be derived from another case class.

2.3 Regular Sequence Pattern Matching

The case class declarations introduced above determine the exact number of contained objects, similar to a function declaration determining the exact number of arguments. But often programmers need to deal with a number of sequence elements that is not known in advance. This can be done using *sequence parameters*.

The last parameter of a formal parameter list can be turned into a sequence parameter by marking its type T with a star (*), which is a shorthand for the type Seq[T]. This works for case class declarations as well as for function declarations. The example from above can thus be extended with a conjunction over an arbitrary number of terms as follows:

```
case class BigAnd(terms: Term*) extends Term ;
```

The field `terms` is of the type Seq[Term]. The Seq trait offers functionality to obtain the length of the sequence, to access its elements, and to iterate over them. Such a sequence parameter offers syntactic convenience by permitting an arbitrary number of arguments in function and constructor calls. A conjunction over an arbitrary number of terms like $\neg x_1 \wedge x_2 \wedge x_3$ can now be expressed as BigAnd(Not(Var(1)), Var(2), Var(3)).

The following code transforms BigAnd's into a series of And's.

```
def toAnd(b:BigAnd): Term = {
  val it = b.terms.elements;
  var t = True;
  while(it.hasNext) { t = And(it.next, t); }
  return t; }
```

In practice, the data to be passed to the constructor may often already be in some sequence representation. In this case, a *sequence escape* is used to guide the compiler. The code **val** xs = List(Not(Var(1)), Var(2), Var(3)); BigAnd(xs:_) constructs the same conjunction as above, using the sequence escape `xs:_*`. If the annotation were missing, the compiler would signal a type error since `xs` of type Seq[Term] cannot be used as a Term.

Let us now discuss how sequences can be decomposed, using pattern matching. A *regular sequence pattern* is a regular expression that possibly is annotated with variable bindings [8]. A variable binding is written $x@p$ where p is a regular pattern that may not contain other variable bindings. For regular expression constructs we use the following standard notations;

- concatenation: p, p concatenates sequence patterns
- the star operator: p^* denotes zero or more occurrences of p

- the plus operator: $p+$ denotes one or more occurrences of p
- the option operator: $p?$ denotes p or the empty sequences.

A choice pattern is a sequence pattern if one of its branches is a sequence pattern. Variables binding a sequence pattern that matches elements of some type T are of type $\text{Seq}[T]$.

Regular patterns can be applied to sequences of any type, not just case classes. Here is an example of a text-processing task using regular patterns:

```
def findRest(z: Seq[Char]): Seq[Char] = z match {
  case Seq(_*, 'G', 'o', 'o'*, 'g', 'l', 'e', rest@(_*)) => rest }
```

This pattern is used to search for the sequence of letters "Gogle" or "Google", or ... If the input z matches, then the function returns what remains after the occurrence, otherwise it generates a runtime error. Possible ambiguities (e.g., for several occurrences of 'Goo*gle-words' in z) are resolved using the (left) shortest match policy which chooses the shortest match for each possibility (such as $_*$), coming from the left. In the example this coincides with matching the *first* occurrence of "Goo*gle" in the input z .

As (conventional) pattern matching is well suited for decomposing ranked trees (i.e., trees in which each node has a fixed number of children), regular sequence pattern matching is the counterpart for decomposing *unranked* trees. As we will see in the following section, regular sequence pattern matching is particularly useful in the context of XML, because XML documents are naturally modeled as unranked trees.

3 XML Facilities in Scala

XML [16] has emerged as the *lingua franca* of the web. Henceforth, all major programming languages are providing, in varying degrees, support to handle XML data. XML documents describe tree-structured data. Functional programming languages, starting with Lisp, have always been particularly well-suited in dealing with trees and tree-structured data. It therefore comes as no surprise that Scala with its functional features is well-suited for XML processing.

The next two subsections describe basic features of XML processing in Scala: our data model of XML, how to express XML documents in Scala code, how to parse XML documents, and how to validate a document against a given schema while parsing. Then, in Section 3.3 we describe how regular sequence pattern matching can be applied to XML data. After a short discussion on namespaces and attributes, section 3.5 describes how to express XML queries with for comprehensions. Finally, we show how to realize queries on XML data, using Scala's elegant concept of for comprehensions.

3.1 Data Model

We give an introduction to Scala's XML data model by contrasting it with the W3C's Document Object Model (DOM), which is characteristic for a number of related object-oriented XML data models. The DOM has been implemented in several programming languages, including Java. We will also compare code written using Java and DOM with code written using Scala.

Consider the XML fragment in Fig. 1.1. It shows a purchase order consisting of two items. The tree structure inherent in such a purchase order is essentially an unranked, ordered tree. Within DOM, such tree structures are doubly-linked: there are pointers to each child of a node

```

<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
  <shipTo country="US">
    <name>Alice Smith</name> <street>123 Maple Street</street>
    <city>Mill Valley</city> <state>CA</state> <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name> <street>8 Oak Avenue</street>
    <city>Old Town</city> <state>PA</state> <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName> <quantity>1</quantity>
      <USPrice>148.95</USPrice> <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName> <quantity>1</quantity>
      <USPrice>39.98</USPrice> <shipDate>1999-05-21</shipDate>
    </item>
  </items>
</purchaseOrder>

```

Listing 1.1. Extract from XML document

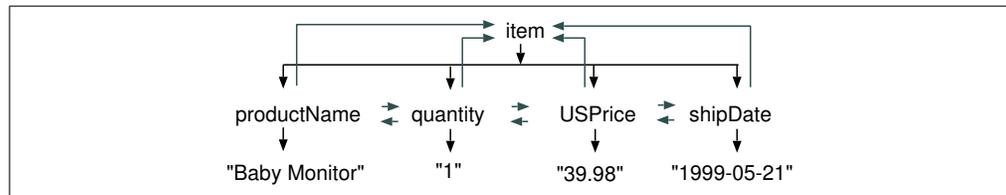


Fig. 1. DOM of a purchase order item

and back, and pointers from each node to its next and previous sibling. Figure 1 depicts the DOM tree structure of a purchase order.

From a programmer's perspective the availability of many tree pointers (parent-child, child-parent, next-sibling, previous-sibling) might offer high flexibility. However, code that uses parent and previous-sibling pointers is hard to read and possibly introduces circularities thus leading to nontermination. Moreover, it increases the footprint of DOM: (1) DOM representations are very *memory costly* (usually at least 4-5 times the space of the original XML document on disk) and (2) program code using DOM is *overly verbose*.

In contrast, Scala has a much thinner model of XML: an XML document is seen as an immutable, ordered (singly-linked) unranked tree. This model agrees with the functional view of a tree in which no pointers to parent or previous-sibling nodes are provided. Programmers can obtain those nodes by storing them in temporary variables upon a traversal. Two major

```

import org.w3c.dom.Document;
import org.w3c.dom.Element;
...
//Retrieve the Document object
DocumentBuilder fact = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document po = fact.parse(new File("po.xml"));
Element root = po.getDocumentElement();

//Retrieve all partNums and compute the grand total for the purchase order
double total = 0;
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++) {
    Node node = children.item(i);
    //Find the items child element
    if ("items".equals(node.getLocalName())) {
        NodeList itemList = node.getChildNodes();
        for (int j = 0; j < itemList.getLength(); j++) {
            Node item = itemList.item(j);

            //Get the partNum attribute value
            NamedNodeMap attrs = item.getAttributes();
            System.out.println("partNum:" + attrs.getNamedItem("partNum"));

            //Find the USPrice child element
            NodeList itemChildren = item.getChildNodes();
            for (int k = 0; k < itemChildren.getLength(); k++) {
                Node child = itemChildren.item(k);
                if ("USPrice".equals(child.getLocalName())) {
                    total += Double.valueOf(child.getNodeValue()).doubleValue();
                }
            }
        }
    }
}
System.out.println("Grand_total_=_ " + total);

```

Listing 1.2. Processing with Java and DOM.

advantages of this representation are (1) memory efficiency (usually around 1-2 times the space of the original XML document on disk) and (2) clean, concise program code.

Before describing the library classes, we demonstrate the latter point consider as example simple programming task over the purchase order data. We want to

- collect the partNum attributes and
- compute the sum of the prices in the USPrice elements.

Using DOM within Java, this task is realized by the code shown in Listing 1.2. Each node is mapped to an object, whose children are accessed by obtaining a `NodeList` using the `getChildNodes()` method, and then accessing each child using a call like `item(i)`. The label of a node is accessed via the `getLocalName()`.

A corresponding Scala code is shown in Listing 1.3. It uses a for comprehension to search for all pairs (z, y) such that z is an item-node and y is a USPrice-child of z . For comprehensions are explained in detail in Section 3.5.

```

import scala.xml.XML;
val doc = XML.loadFile("po.xml");
var total = 0;
for(val z <- doc \\ "item";
    val y <- z \ "USPrice") {
  Console.println("partnum:_" + z \ "@partNum");
  total = total + Double.valueOf(y.text)
}
Console.println("Grand_total_" + total);

```

Listing 1.3. Processing with Scala.

The `Node` class embodies Scala's tree model of XML. It describes the components of an XML node, namely the namespace prefix, the label, a pointer to the namespace bindings in scope, the attributes and the children. This library class provides also methods for XML serialization, namespace/attribute lookup, XPath selection etc.

```

abstract class Node extends NodeSeq {
  def prefix: String;
  def label: String;
  def scope: NamespaceBinding;
  def attributes: MetaData;
  def child: Seq[Node]; ... }

```

`NodeSeq` is a wrapper class that acts as a proxy for an arbitrary sequence of nodes – this can be a list, an array, or any other custom representation (a single `Node` acts as a singleton sequence). It adds methods `\` and `\\` than can be used like corresponding XPath operators `/` and `//`.

```

abstract class NodeSeq extends Seq[Node] { ...
  def \ (that: String): NodeSeq = {...}
  def \\ (that: String): NodeSeq = {...} }

```

Note that the operators `\\` and `\` are *not* build-in operators of the Scala language, but, are just ordinary methods, since operator characters can be used as method names. Scala resolves an expression of the form `a id b` into the more familiar `a.id(b)`. A fixed precedence scheme guides the parser and determines in which direction operations associate.

Concrete subclasses of `Node` exist for elements, text nodes, comments, processing instructions, and entity references. The informed reader might wonder about namespace and root nodes – the root node is represented by a class `Document`, and namespace nodes are instances of `NamespaceBinding` which is described further down.

Elements are by default represented using the `scala.xml.Elem` class, which is – not surprisingly – a case class. The accessor methods for the constructor arguments provide exactly the methods required by the superclass `scala.xml.Node`.

```

case class Elem(prefix: String, label: String, attributes: MetaData,
               scope: NamespaceBinding, child: Node*) extends Node { ... }

```

3.2 Literals, Parsing and Validation

XML syntax can be used directly in a Scala program, e.g., in value definitions.

```
val labPhoneBook =
  <phonebook>
  <descr>Phone numbers of<b>XML</b> hackers.</descr>
  <entry>
    <name>Burak</name>
    <phone where="work"> +41 21 693 68 67 </phone>
    <phone where="mobile"> +41 78 601 54 36 </phone>
  </entry>
</phonebook>;
```

The value `labPhoneBook` is an XML tree; for instance, one of its nodes has the label `phone`, a child sequence consisting of a single text node labeled by `+41 2...`, and a map from the attribute key `where` to the value `"work"`. Within XML syntax it is possible to escape to Scala using the brackets `{` and `}` (similar to the convention used in XQuery). For example, a date node with a child text node consisting of the current date can be defined by `<date>{ df.format(new java.util.Date()) }</date>`.

Parsing XML data is done by means of the `load` method of the object `scala.xml.XML`. Scala's XML parser is of course entirely written in Scala, and is also used to parse the XML literals described above. The XML standard actually describes two variants of XML parsers – the validating and the non-validating ones. The parser library parser provides support for validation, which can be enabled as shown in the following lines.

```
val fil = new java.io.File("data.xml");
val prs = new scala.xml.parsing.ConstructingParser(fil, true)
      with ValidatingMarkupHandler ; // true = preserve whitespace
prs.nextch;                          // initialize parser
val d = prs.document();                // returns Document instance
val elem = d.docElem;
val dtd = d.dtd;
```

The above code is a case of so-called mix-in composition: a mix-in class `ValidatingMarkupHandler` overrides certain members of the existing class `ConstructingParser`, changing their behavior from non-validating to validating. In general, mix-in composition is a flexible means to pull together pieces of code that have been factored out into components [14].

The user also has the possibility to connect an event handler to the parser, which gets called back whenever a subelement has been successfully parsed. This allows to traverse the document without necessarily constructing it in memory. It can be useful, for instance, in order to run an optimized query directly during parse time of the document and without actually constructing the document, or, if based on the element name different representations (of different types) are to be constructed.

3.3 Regular Matching on XML Nodes

XML nodes can be decomposed using pattern matching. Scala allows to use XML syntax here too, albeit only to match elements. The following example shows how to add an entry to a phonebook element.

```
import scala.xml.{ Node, XML } ;
def add(pbook: Node, newEntry: Node): Node =
  pbook match {
    case <phonebook>{ cs @ _* }</phonebook> =>
      <phonebook>{ cs }{ newEntry }</phonebook>
  }
val newPhoneBook =
  add(XML.load("savedPhoneBook"),
    <entry>
      <name>Sebastian</name>
      <phone where="work">+41 21 693 68 67</phone>
    </entry>);
```

The add function performs a match on the phonebook element, binding its child sequence to the variable cs (the regular sequence pattern `_*` matches an arbitrary sequence). Then it constructs a new phonebook element with child sequence cs followed by the node newEntry. Note that the pattern `cs @ _*` appears inside code braces, it would otherwise be interpreted as literal text.

The compiler turns the above shorthand into constructor calls of the class `scala.xml.Elem`. New temporary variables are introduced to deal with namespace definitions and redefinitions and prefixed and unprefixed attributes. If we ignore those for a moment, the resulting code is equivalent to the following (which could equally well be written by a user that despises angle brackets)

```
import scala.xml._ ;
def add(pbook: Node, newEntry: Node): Node =
  pbook match {
    case Elem(_, "phonebook", _, _, cs @ _*) =>
      Elem(null, "phonebook", Null, $scope,
        (new scala.xml.NodeBuffer() &+ ch &+ newEntry):_*)
  }
val newPhoneBook = add(XML.load("savedPhoneBook"),
  Elem(null, "entry", Null, $scope, Null,
    Elem(null, "name", Null, $scope, Text("Sebastian"))
    Elem(null, "phone", new UnprefixedAttribute("where", "work", Null), $scope,
      Text("+41_21_693_68_67"))));
```

More involved patterns are possible. The following pattern traverses the children of a node and finds out whether a book with the title `<scala/xml>` comes before a book with the title `XSLT Reference`, and if so, returns the book elements between those two.

```
books match {
  case <books>{ _*, <book><title>&lt;scala/xml&gt;</title>{ _ * }</book>,
    mid @ _*, <book><title>XSLT Reference</title>{ _ * }</book>,
    _* }</books> => mid
  case _ => Nil
}
```

3.4 Namespaces and Attributes

Attributes and namespaces are implemented as immutable, linked lists. Attributes with namespace prefixes are distinguished from ones without. For the reader not versed in the namespace issues regarding XML, it might suffice to know that the Namespaces in XML Recommendation [16] specification, which has been introduced long after the XML specification, provides a means to ‘package’ related names by associating them with a uniform resource identifier (URI). The association avoids name collisions and happens indirectly by (1) binding URIs to prefixes and (2) prefixing names using the syntax `ns:localname`.

For convenience, a *default namespace* may be declared that applies to unprefix names. Finally, it is also possible to undeclare prefix bindings. As an example, consider the following element:

```
<seat class="Y" ht:class="blink" xmlns:ht="urn:hypertext">33B</seat>
```

This element defines two distinct `class` attributes, the second being in the namespace determined by the `ht` prefix. This prefix is bound by a namespace declaration `xmlns:ht="urn:hypertext"`, which happens to be in the same element. Here is the equivalent Scala code, with an explanation below:

```
Elem(null, "seat",
  new UnprefixedAttribute("class", "Y",
    new PrefixedAttribute("ht", "class", "blink", Null)),
  new NamespaceBinding("ht", "urn:hypertext", TopScope),
  Text("33B"))
```

The default namespace is identified by the `null` reference. Namespaces are encoded in a linked list of namespaces bindings (and “un-bindings”). The nodes of the list are shared among nodes of the same subtree. In this way, we fully support namespaces in Scala’s literal syntax and library. The programmer can extend an existing scope `scp` with the above binding is achieved by writing `new NamespaceBinding("html", "urn:hypertext", p)`, with object `TopScope` acting as the top-level (empty) scope.

3.5 XML Queries through For Comprehension

For comprehensions are a functional feature used e.g. in Haskell to concisely express computations on lists. In general, a for comprehension consists of generators (a sequence expression whose elements will subsequently be assigned to a variable) and filters (boolean expressions that eliminate some of the elements). The following code shows how to concisely compute a list pairs of numbers i, j between 0 and 100 where $i = 3n$ and $j < i$.

```
for (val i <- List.range(0, 100); i % 3 == 0; val j <- List.range(0, i); )
  yield Pair(i, j);
```

For comprehensions are related to queries on XML data in a way that complements pattern matching. Where the latter yields at most one result for a pattern, queries are usually used to compute *all* matches of a query. Scala’s flexible comprehension mechanism can be used to this end, allowing for a concise and elegant style that closely resembles XQuery. In the following example, we select all entry elements from `labAddressbook` and from `labPhoneBook` into the variables `a` and `p`, respectively. Whenever the name contents of two such entries coincide, a result element is generated which has as children the address and phone number, taken from the appropriate entry.

```

<!ELEMENT phonebook (descr, entry*)>
<!ELEMENT descr      (#PCDATA | b)*>
<!ELEMENT b          (#PCDATA)>
<!ELEMENT entry      (name, phone+)>
<!ELEMENT name       (#PCDATA)>
<!ELEMENT phone      (#PCDATA)>
<!ATTLIST phone      where (home|mobile|work) "home">

```

Listing 1.4. A typical DTD.

```

for (val a <- labAddressBook \\ "entry";
      val p <- labPhoneBook \\ "entry";
      a \ "name" == p \ "name") yield
<result>{ a.child }{ p \ "phone" }</result>

```

Note the variable order inside the `for` comprehension: `labAddressBook` is traversed in document order (=depth-first left-to-right) and for each entry found, `labPhoneBook` is traversed in document order. Hence, the order of entries in the result will be the order of entries in `labAddressBook`; If `a` and `p` are exchanged inside the `for`, then the order of results will be as in `labPhoneBook`.

3.6 Data Binding

Types of XML documents are usually specified by so called schemas. The act of checking conformance to a schema is called validation. Popular schema formalisms are DTD (Document Type Definition), XML Schema [16], and RELAX NG [12]. At this moment a simple support for DTDs is available through the `schema2src` tool. It converts a DTD to a set of class definitions which can only be instantiated with XML data that is valid with respect to the DTD.

The idea here is to map the generic tree representation to case classes. We can create one case class per element tag, and place code that validates in the constructor. The validating code uses Scala's regular expression library. The validation functions can also be called separately, and the tool is written in an extensible way to accommodate user-defined modules for custom schema languages.

XML documents that are represented generically can be validated and optionally translated into the specialized representation, by using the so-called binder object.

```

val pb = dtd.binder.validate(<phonebook> ... </phonebook>);
pb match {
  case phonebook(_, _*,
    entry(md,
      name(Text("Sebastian")),
      p@phone(_,_*), _*)) => p }

```

Existing XML documents can then be validated against the DTD using a special load method which tries to instantiate the corresponding classes. Scala's regular expression library is used to convert content models to finite automata.

```

class MyServlet extends HttpServlet {
  def getTime = java.util.Calendar.getInstance().getTime();

  def doGet(req:HttpServletRequest, res:HttpServletResponse) = {
    val page = <html>
      <head> <title>Hello, Servlet World!</title> </head>
      <body>
        <h1>Hello, Servlet World!</h1>
        <p> The time is <b>{ getTime.toString() }</b> </p>
      </body>
    </html>;
    res.getWriter().write( page.toString() );
  } }

```

Listing 1.5. A friendly Scala servlet.

4 Services

In this section, we will give some examples that show how Scala can be used to create web services. For reasons of space, we will only give basics of servlet programming and refer the interested reader to the real life applications that were written in Scala and are in active use.

4.1 Common Gateway Interface

Web services were born out of ad hoc interfaces to webservers that became standardized in one way or another. The earliest of them, the Common Gateway Interface (CGI), may be considered as the first defining protocol of web services. CGI programs can be written in almost any language, but the cost of real life CGI programs is influenced by two crucial issues:

First, new variants of CGI use long-running processes that communicate with the web server and handle requests concurrently. Scala's suitability for concurrency will be analyzed in detail in the next section. Second, writing server side programs is made hard due to inversion of control and state [15]. We are currently working on an implementation for an infrastructure for web services that solves the control issue using continuations. This task is greatly simplified by having first-class functions in the language.

4.2 Servlets

On the Java platform, using the CGI is usually discarded in favor of *servlets*. These are small portions of object-oriented code centered around a *servlet container* or *servlet engine* which maps URL requests of the HTTP protocol to method calls.

Programmers can choose among many servlet containers to run their Java servlets. It is easy to write servlets in Scala making use of the tight integration between the two languages. A sample servlet is given in Listing 1.5.

Built on top of the servlet technology, we have implemented a range of web services that are in active use. These are the Scala bugtracking system, and an online auction software. Both are three-tier systems that generate XHTML for presentation purposes. We found developing real life applications quite useful to guide the design of the XML library and testing the implementation of the literal syntax.

4.3 Remote Invocations

The concept of a web services does not (and might never) benefit from an agreed upon definition. Nevertheless, specifications like XMLRPC and the Simple Object Access Protocol (SOAP) have emerged as language-independent remote method invocation protocols that enjoy some popularity in this context. Space limitations hinder us from giving a full example; given the above explanations it is clear that Scala provides everything needed to handle SOAP fragments like the following one of a Google service. For the networking tasks, the relevant Java network library are at the programmer's disposal.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestion xmlns:ns1="urn:GoogleSearch"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <key xsi:type="xsd:string">00000000000000000000000000000000</key>
      <phrase xsi:type="xsd:string">Boogle</phrase>
    </ns1:doSpellingSuggestion>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

We can summarize that by these properties of Scala's XML syntax and data model, applications benefit from improved integration over present solutions built from Java plus XML libraries.

5 Concurrency

Finally, we shall point out that concurrency is an application area for which Scala is particularly well suited. Consider the task of implementing an electronic auction service. We focus here on an Erlang-style actor process model [1] to implement the participants of the auction. An alternative approach to concurrent programming in Scala using pi-calculus is described elsewhere [7].

Actors are objects to which messages are sent. Every process has a "mailbox" of its incoming messages which is represented as a queue. It can work sequentially through the messages in its mailbox, or search for messages matching some pattern.

For every traded item there is an auctioneer process that publishes information about the traded item, that accepts offers from clients, and that communicates with the seller and winning bidder to close the transaction. We present an overview of a simple implementation here.

In Listing 1.6, we define the messages that are exchanged during an auction. There are two traits: `AuctionMessage` for messages from clients to the auction service, and `AuctionReply` for replies from the service to the clients. Both classes are extended with a number of cases.

Figure 1.7 presents a Scala implementation of a class `Auction` for auction processes that coordinate the bidding on one item. Objects of this class are created by indicating (1) a seller process which needs to be notified when the auction is over, (2) a minimal bid, and (3) the date when the auction is to be closed.

The process behavior is defined by its `run` method. That method repeatedly selects (using `receiveWithin`) a message and reacts to it, until the auction is closed, which is signaled by a `TIMEOUT` message. Before finally stopping, it stays active for another period determined by the `timeToShutdown` constant and replies to further offers that the auction is closed.

```

trait AuctionMessage;
case class Offer(bid: int, client: Actor) extends AuctionMessage;
case class Inquire(client: Actor) extends AuctionMessage;

trait AuctionReply;
case class Status(asked: int, expire: Date) extends AuctionReply;
case object BestOffer extends AuctionReply;
case class BeatenOffer(maxBid: int) extends AuctionReply;
case class AuctionConcluded(seller: Actor, client: Actor)
extends AuctionReply;
case object AuctionFailed extends AuctionReply;
case object AuctionOver extends AuctionReply;

```

Listing 1.6. Implementation of an Auction Service

```

class Auction(seller: Actor, minBid: int, closing: Date) extends Actor {
  val timeToShutdown = 36000000; // msec
  val bidIncrement = 10;
  override def run() = {
    var maxBid = minBid - bidIncrement;
    var maxBidder: Actor = _;
    var running = true;
    while (running) {
      receiveWithin ((closing.getTime() - new Date().getTime())) {
        case Offer(bid, client) =>
          if (bid >= maxBid + bidIncrement) {
            if (maxBid >= minBid) maxBidder send BeatenOffer(bid);
            maxBid = bid; maxBidder = client; client send BestOffer;
          } else {
            client send BeatenOffer(maxBid);
          }
        case Inquire(client) =>
          client send Status(maxBid, closing);
        case TIMEOUT =>
          if (maxBid >= minBid) {
            val reply = AuctionConcluded(seller, maxBidder);
            maxBidder send reply; seller send reply;
          } else {
            seller send AuctionFailed;
          }
      }
      receiveWithin(timeToShutdown) {
        case Offer(_, client) => client send AuctionOver
        case TIMEOUT => running = false;
      }
    }
  }
}

```

Listing 1.7. Implementation of an Auction Service

Here are some further explanations of the constructs used in this program:

- The `receiveWithin` method of class `Actor` takes as parameters a time span given in milliseconds and a function that processes messages in the mailbox. The function can be expressed directly as a matching expression. The `receiveWithin` method selects the first message in the mailbox which matches one of these patterns and applies the corresponding action to it.
- The last case of `receiveWithin` is guarded by a `TIMEOUT` pattern. If no other messages are received in the meantime, this pattern is triggered after the time span which is passed as argument to the enclosing `receiveWithin` method. `TIMEOUT` is a particular instance of class `Message`, which is triggered by the `Actor` implementation itself.
- Reply messages are sent using syntax of the form `destination send SomeMessage`, which expands to `destination.send(SomeMessage)` as described before.

All the constructs discussed above are offered as methods in the library class `Actor`. That class is itself implemented in Scala, based on the underlying thread model of the host language (e.g. Java, or .NET). The implementation of all features of class `Actor` used here is given in Section 5.2.

5.1 Mailboxes

Mailboxes are high-level, flexible constructs for process synchronization and communication. They allow sending and receiving of messages. A *message* in this context is an arbitrary object. There is a special message `TIMEOUT` which is used to signal a time-out.

```
case object TIMEOUT;
```

Mailboxes implement the following signature.

```
class MailBox {
  def send(msg: Any): unit;
  def receive[a](f: PartialFunction[Any, a]): a;
  def receiveWithin[a](msec: long)(f: PartialFunction[Any, a]): a;
}
```

The state of a mailbox consists of a multi-set of messages. Messages are added to the mailbox using the `send` method. Messages are removed using the `receive` method, which is passed a message processor `f` as argument. This is a partial function from messages to some result type, which can be implemented as a pattern matching expression. The `receive` method blocks until there is a message in the mailbox for which its message processor is defined. The matching message is then removed from the mailbox and the blocked thread is restarted by applying the message processor to the message.

Here's how the mailbox class can be implemented:

```
class MailBox {
  private abstract class Receiver extends Signal {
    def isDefined(msg: Any): boolean;
    var msg = null;
  }
}
```

We define an internal class for receivers with a test method `isDefined`, which indicates whether the receiver is defined for a given message. The receiver inherits from class `Signal` a `notify`

method which is used to wake up a receiver thread. When the receiver thread is woken up, the message it needs to be applied to is stored in the msg variable of Receiver.

```
private val sent = new LinkedList[Any];
private var lastSent = sent;
private val receivers = new LinkedList[Receiver];
private var lastReceiver = receivers;
```

The mailbox class maintains two linked lists, one for sent but unconsumed messages, the other for waiting receivers.

```
def send(msg: Any): unit = synchronized {
  var r = receivers, r1 = r.next;
  while (r1 != null && !r1.elem.isDefined(msg)) { r = r1; r1 = r1.next }
  if (r1 != null) {
    r.next = r1.next; r1.elem.msg = msg; r1.elem.notify;
  } else {
    lastSent = insert(lastSent, msg);
  } }
}
```

The send method first checks whether a waiting receiver is applicable to the sent message. If yes, the receiver is notified. Otherwise, the message is appended to the linked list of sent messages.

```
def receive[a](f: PartialFunction[Any, a]): a = {
  val msg: Any = synchronized {
    var s = sent, s1 = s.next;
    while (s1 != null && !f.isDefinedAt(s1.elem)) { s = s1; s1 = s1.next }
    if (s1 != null) {
      s.next = s1.next; s1.elem
    } else {
      val r = insert(lastReceiver, new Receiver {
        def isDefined(msg: Any) = f.isDefinedAt(msg);
      });
      lastReceiver = r;
      r.elem.wait();
      r.elem.msg
    } }
  f(msg)
}
```

The receive method first checks whether the message processor function f can be applied to a message that has been sent but not consumed yet. If yes, the thread continues immediately by applying f to the message. Otherwise, a new receiver is created and linked into the receivers list, and the thread waits for a notification on this receiver. Once the thread is woken up again, it continues by applying f to the message that was stored in the receiver. The insert method on linked lists is defined as follows.

```
def insert(l: LinkedList[a], x: a): LinkedList[a] = {
  l.next = new LinkedList[a];
  l.next.elem = x;
  l.next.next = l.next;
  l }
}
```

The mailbox class also offers a method `receiveWithin` which blocks for only a specified maximal amount of time. If no message is received within the specified time interval (given in milliseconds), the message processor argument f will be unblocked with the special `TIMEOUT` message. The implementation of `receiveWithin` is quite similar to `receive`:

```
def receiveWithin[a](msec: long)(f: PartialFunction[Any, a]): a = {
  val msg: Any = synchronized {
    var s = sent, s1 = s.next;
    while (s1 != null && !f.isDefinedAt(s1.elem)) {
      s = s1; s1 = s1.next ;
    }
    if (s1 != null) {
      s.next = s1.next; s1.elem
    } else {
      val r = insert(lastReceiver, new Receiver {
        def isDefined(msg: Any) = f.isDefinedAt(msg);
      });
      lastReceiver = r;
      r.elem.wait(msec);
      if (r.elem.msg == null) r.elem.msg = TIMEOUT;
      r.elem.msg
    }
  }
  f(msg)
} // end MailBox
```

The only differences are the timed call to `wait`, and the statement following it.

5.2 Actors

The auction service was based on high-level actor processes, that work by inspecting messages in their mailbox using pattern matching. An actor is simply a thread whose communication primitives are those of a mailbox. Actors are hence defined as a mixin composition extension of Java's standard `Thread` class with the `MailBox` class.

```
abstract class Actor extends Thread with MailBox;
```

6 Conclusion

We have discussed Scala's facilities for dealing with semistructured data and concurrency. We have shown that the combination of functional and object-oriented programming constructs lets one design expressive high-level libraries that are also easy to use. Two examples of this approach were Scala's libraries for XPath-style XML navigation and Erlang-style actors.

Of course, moving a construct from a language to a library is not a silver bullet by itself. However, the library-based approach has two benefits. First, the core language can be kept simpler and more general. Second, libraries are much easier to upgrade and extend than core language constructs.

In future work, we plan to further pursue the library based approach. Our next targets are a framework for database integration and a web service infrastructure based on continuations. Scala is available under a free software license under <http://scala.epfl.ch>.

Acknowledgments The Scala design and implementation has been a collective effort of many people. Besides the authors, Philippe Altherr, Vincent Cremet, Julian Dragos, Gilles Dubochet, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman and Matthias Zenger have made important contributions. The work was partially supported by grants from the Swiss National Fund under project NFS 21-61825, the European Framework 6 project PalCom, the Swiss National Competence Center for Research MICS, Microsoft Research, and the Hasler Foundation.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
2. D. Atkins, T. Ball, G. Bruns, and K. Cox. Mawl: a domain-specific language for form-based services. *IEEE Trans. Software Eng.*, 25(3):334–346, 1999.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. In *Proc. of the 16th European Conference on Object-oriented programming.*, LNCS 2374, 2002.
4. G. Bierman, E. Meijer, and W. Schulte. The essence of data access in $C\omega$. In *Proc. of the 19th European Conference on Object-Oriented Programming*, LNCS 3586, 2005.
5. C. Brabrand, A. Møller, and M. I. Schwartzbach. The <bigwig> project. *ACM Transactions on Internet Technology*, 2(2):79–114, 2002.
6. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.
7. V. Cremet and M. Odersky. PiLib: A hosted language for pi-calculus style concurrency. In *Dagstuhl proc.: Domain-Specific Program Generation*, 2003.
8. B. Emir. Extending pattern matching with regular tree expressions for XML processing in Scala. Master’s thesis, Rheinisch-Westfälische Technische Hochschule Aachen, 2003.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
10. V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.
11. M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. Xj: facilitating XML processing in Java. In *Proc. of the 14th International Conference on World Wide Web*, 2005.
12. Oasis. RELAX NG. See <http://www.oasis-open.org/>.
13. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical Report IC/2004/64, Ecole Polytechnique Fédérale de Lausanne, 2004.
14. M. Odersky and M. Zenger. Scalable component abstraction. In *Proc. of 20th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, 2005.
15. C. Quiennec. The influence of browsers on evaluators or, continuations to program web servers. In *Proc. of the 5th ACM SIGPLAN international conference on Functional programming*, 2000.
16. World wide web committee. <http://www.w3.org/>
17. M. Wallace and C. Runciman. Haskell and XML: Generic combinators or type-based translation? In *Proc. of the 4th International Conference on Functional Programming*, 1999.