XLambda: A Functional Programming Language with XML Syntax

Ruhsan Onder
Department of Computer Engineering
Cyprus International University
Nicosia, T.R.N.Cyprus
Email: ronder@ciu.edu.tr

Zeki Bayram
Department of Computer Engineering
Eastern Mediterranean University
Famagusta, T.R.N.Cyprus
Email: zeki.bayram@emu.edu.tr

***Abstract*—We describe XLambda, a functional language with XML syntax, and its processor which is implemented fully and completely in XSLT. XLambda has all the basic features of a functional language, such as defining named functions, operations on numbers, passing functions as parameters, constructing arbitrary data structures etc. What sets XLambda apart from the rest of the functional languages is not its feature set though, but rather its syntax and processor which is implemented as an XSLT stylesheet. Since most Web browsers have XSLT processors already built in, XLambda has the potential to be used as a scripting language in Browsers, much in the same way as JavaScript, among other possibilities.**

I. Introduction

Functional programming languages have a clean syntax and well defined semantics. More importantly, control flow is specified implicitly through function application, and the resulting code is much more concise, easy to understand and maintain. Examples of functional languages include Lisp [1], Haskell [2] and CAML [3].

In this paper, we describe XLambda, a functional language with XML syntax. XLambda was developed as part of a project to implement the operational and denotational semantics of programming languages in XSLT [4]. Its original purpose was to serve as the object language in the implementation of the denotational semantics of another XML based language, XIM [5]. It has evolved since then, with the addition of list/tree construction primitives, into a real programming language, with XML syntax. This, together with the fact that its processor is implemented completely in XSLT [6], opens up a whole new world of possibilities for XLambda that are not available with more traditional functional languages. These include using XLambda as a scripting language inside browsers, since most browsers already have XSLT processors in them. A more esoteric usage area would be encoding the meaning of objects as lambda expressions (the standard approach for specifying the meaning of program segments in denotational semantics) and sending the meaning of an object, together with the object, over the Internet to be analyzed by the user of the object. Alternatively, XLambda could be used to encode agents, to be executed on clients browsers, since the environment that the agent requires for execution (i.e. the interpreter for XLambda in the form of an XSLT stylesheet), would travel with it. We have not as yet exploited these areas of possible application for XLambda- they are part of future work. What we describe here is XLambda itself, its syntax, some example programs in XLambda, and a somewhat detailed exposition of its implementation in XSLT.

The remainder of this paper is organized as follows. Section II describes informally the syntax of XLambda. A formal description as an XML Schema [7] could not be included due to space considerations. Operational semantics of XLambda is given in III. Section IV compares XLambda to the very few langauges with similar features, and points out their weaknesses compared to XLambda. Finally, we have the conclusion and future research directions in section V.

II. XLambda Syntax

In XLambda, an expression is either an integer (represented by a <num> element), a λ-variable (represented by a <var> element), an arithmetic expression (represented by an <op> element), a boolean expression (represented by a <boolop> element), a λ-abstraction (represented by a <lambda> element), an application (represented by an <apply> element), or a named function (represented by a <named_function> element). In addition to these expression variants we have <if> elements to represent conditional constructs (*if-then-else*), <pair> elements to represent lists or trees, and also a <nil> element to indicate empty lists/trees.

The <op> and <boolop> elements for representing arithmetic/boolean expressions have two child elements.

The <if> elements for the conditional construct have three children. The first child represents the conditional expression. The second child represents the *then* part, and the third child represents the *else* part. Figure 1 is an example of *if-then-else* construct.

The syntax specification of XLambda is general enough to be able to represent any lambda calculus expression such as (λ f. f 3) (λ x. x+1) whose XLambda representation is depicted in Figure 2. The λ-reducer we implemented and discussed in Section III can evaluate any λ-expression written in XLambda syntax. Normal order evaluation is employed for good termination behavior. α-reduction is used whenever variable capture is a possibility.

Figure 3 depicts a function in XLambda that takes two parameters and returns their sum. Figure 4 shows the definition

```
<if>
   <apply>
      <var name="e"/>
      <var name="s"/>
   </apply>
   <apply>
      <named_function name="add">
      <num value="1"/>
   </apply>
   <apply>
      <named_function name="add">
      <num value="3"/>
   </apply>
</if>
```

Fig. 1. XLambda code for the expression *"if (e s) then (add 1) else (add 3)"*

```
<apply>
    <lambda var="f">
        <apply>
           <var name="f"/>
           <num value="3"/>
        </apply>
    </lambda>
    <lambda var="x">
        <op opname="+">
            <var name="x"/>
            <num value="1"/>
        </op>
    </lambda>
</apply>
```

Fig. 2. XLambda code for the expression (λ f. f 3) (λ x. x+1)

of the well-known recursive *factorial* function.

III. Implementation of XLambda Operational Semantics: λ-Reduction Machine in XSLT

In this section, we describe in some detail the operational semantics implementation of XLambda.

A. Evaluation at the Top-Level

We implemented the operational semantics of lambda calculus to be able to evaluate XLambda code. In our implementation, we used β-reduction and α-conversion together with a combination of lazy and eager evaluation strategies [8]. In lazy evaluation, the leftmost outermost redex is first reduced and evaluation of expressions that are passed as parameters are deferred until their evaluation is required. That is, in function applications of the form (*e1 e2*), *e1* is reduced until an unreducible form is reached (a λ-abstraction) and then β-reduction is carried out. The advantage of lazy evaluation is that it has better termination properties as a result of deferred parameter evaluation and therefore can sometimes produce results when an eager strategy would enter infinite recursion. However, care should be taken to avoid evaluating the same expression more than once. This can happen when a parameter is used more than once in the body of a function. In this case, eager evaluation is used to avoid the re-evaluation.

The top-level template for evaluating λ-expressions matches the root node which encloses the expression and calls template *Evaluate* with its first child as depicted in Figure 5.

```
<define_function name="add">
   <lambda var="x">
      <lambda var="y">
         <op opname="+">
            <var name="x"/>
            <var name="y"/>
         </op>
      </lambda>
   </lambda>
</define_function>
```

Fig. 3. A simple function that adds its two parameters

```
<define_function name="factorial">
   <lambda var="n">
       <if>
          <boolOp test="eq">
             <var name="n"/>
             <num value="1"/>
          </boolOp>

          <num value="1"/>

          <op opname="*">
              <var name="n"/>
              <apply>
                 <named_function name="factorial"/>
                 <op opname="-">
                     <var name="n"/>
                     <num value="1"/>
                 </op>
              </apply>
          </op>
       </if>
   </lambda>
</define_function>
```

Fig. 4. The well-known recursive *factorial* function

B. Template Evaluate of The Reducer

The template *Evaluate* whose algorithm is depicted in Figure 6 uses the *case* construct to determine the type of the incoming expression in parameter *$prNode* to take action accordingly. If an <apply> element is under consideration, it calls itself recursively with the left redex (that is the first child of the <apply> element), and it makes another recursive call with the returned result of the first recursive call. This is to ensure that the leftmost redex is reduced to a λ-abstraction (because the replacement in the next step can only take place if the left redex is a λ-abstraction). Then the unevaluated right redex is replaced in the result of the last recursive call. The template *Replace* is called to perform the reduction. After this replacement another recursive call is performed for evaluating the result of the replacement.

In the case that the incoming expression to *Evaluate* is a

```
<xsl:template match="/">
   <xsl:call-template name="Evaluate">
      <xsl:with-param name="prNode" select="child::*[1]"/>
   </xsl:call-template>
</xsl:template>
```

Fig. 5. The top-level template for evaluating λ-expressions written in XLambda

input parameter:
$prNode (expression to be evaluated)
Case (the incoming parameter *$prNode)* **of**

- **an <apply>** element :
  1) Call *Evaluate* recursively with the 1st child of <apply> to evaluate left redex and store result in variable *$Tex*
  2) Call *Evaluate* recursively with *$Tex* to evaluate result of step 1 and store result in variable *$Expr*
  3) To reduce the right redex in the evaluated left redex call *Reduce* with *$Expr* as parameter *$redex*, 2nd child of <apply> (the right redex) as parameter *$substitute* and the bound variable of *$Expr* as parameter *$bound* and store result in variable *$reduced*
  4) Call *Evaluate* with the result of reduction in step 3 and return result
- **a <lambda>, <num>, <var>, <nil>** element : return the node as result
- **a <named_function>** element : match the corresponding definition of the named λ-abstraction in "auxiliaries.xml" and return it (that is retrieve the <define_function> element having attribute @*name* equal to the attribute @*name* of incoming <named_function> element)
- **an <if>** element : Evaluate 1st child (condition) with a recursive call to *Evaluate*, if the returned result is equal to "1" call *Evaluate* with 2nd child (then part), otherwise call *Evaluate* with 3rd child (else part)
- **an <op> or <boolOp>** element :
  1) Call *Evaluate* with 1st child (left operand) get the attribute @*value* of the returned result (since a <num> element is returned) and store in variable *$O1*
  2) Call *Evaluate* with 2nd child (right operand) get the attribute @*value* of the returned result and store in variable *$O2*
  3) Return the result of performing the corresponding operation (indicated by the attribute @*opname* in case of <op> and @*test* in case of <boolOp>) on *$O1* and *$O2* in a <num> element
- **a <pair>** element:
  Call *Evaluate* with each child and return the result as a <pair>
- **an <is-null>** element :
  Call *Evaluate* with the 1st child and test if the result of evaluation is null. If it is so then return 1, otherwise return 0
- **a <first>** element :
  Call *Evaluate* with the 1st child and return the first child of the result of the evaluation
- **a <rest>** element :
  Call *Evaluate* with the 1st child and return the second child of the result of the evaluation

Fig. 6. Algorithm of the template *Evaluate* of the λ-reducer

<lambda> element (i.e. a λ-abstraction), it is returned back as it is.

When an <if> element is encountered, first *Evaluate* is called with its first child, the condition, then according to the returned result *Evaluate* is called with either one of its remaining two children (that is *then* part or *else* part).

In case of a <named_function> the lambda code for the corresponding named λ-abstraction is obtained from the "auxiliaries.xml" document by the use of the *"document()"* function provided by XSLT [6]. The *"document()"* function provides access to the contents of the document whose path is supplied. The fully qualified name of the file containing named λ-abstractions of the auxiliary functions is supplied as a parameter to the "document()" function and the XPath expression "//define_function[@name= $prNode/@name]/child ::*[1]" is concatenated to this call for matching the corresponding <define_function> element which has the @*name* attribute same as the @*name* attribute of the incoming <named_function> element in *$prNode*.

input parameters:
$redex (expression to be reduced)
$bound (bound variable of the redex)
$substitute (expression which will replace the occurrences of the *$bound*)
$flag (0:no substitution only α-conversion can be performed, 1:substitution is allowed besides α-conversion)
Case (the incoming parameter *$redex*) **of**
a <lambda> element :

1) **If** this inner λ-abstraction binds a variable having the same name as *$bound* (if *$bound*=*$redex/@var*)
   Then no reduction is performed (an inner λ-scope binding the same variable)
   Call *Replace* for all children of *$redex* with *$flag*=0 (to prevent substitution of the variable bound in the local scope and permit only α conversion)
2) **Else if** this inner λ-abstraction binds a variable having the same name with *$substitute* (*$substitute/@name*=*$redex/@var*) or substitute is an expression rather than a single free variable
   Then apply α-conversion with β-reduction
   Rename bound variable of inner λ-expression and all of its occurrences while substituting *$bound* with *$substitute*
   Else (no need for α-conversion)
   Perform substitution and Call *Replace* for all children of *$redex*

an <apply>, <op>, <boolOp>, <if>, <pair>, <is-null>, <first>, <rest> element : Call *Replace* for all children of *$redex*
a <named_function>, <num>, <nil> element : no substitution
a <var> element :

1) **If** <var> has the same name with $bound : **if** (*flag*=1) **then** substitute <var> with *$substitute* **else** no substitution (variable is bound in the inner scope)
2) **Else if** <var> has the same name with *$substitute* or *$substitute* is an expression rather than a single free variable **Then** apply α-conversion (rename <var>)
   Else no substitution, no renaming

Fig. 7. Algorithm of the template *Replace* which performs β-reduction and α-conversion as needed

```
<apply>
   <lambda var="y">
       <lambda var="x">
           <op opname="+">
               <var name="x"/>
               <var name="y"/>
           </op>
       </lambda>
   </lambda>
   <var name="x"/>
</apply>
```

Fig. 8. XLambda code for (λ y.λ x. x+y)x

```
<lambda var="x1">
    <op opname="+">
        <var name="x1"/>
        <var name="y"/>
    </op>
</lambda>
```

Fig. 9. Result of evaluating expression of Figure 8, λ x1. x1+x

```
<apply>
   <lambda var="x">
      <apply>
         <lambda var="y">
            <lambda var="x">
                <apply>
                   <var name="y"/>
                   <var name="x"/>
                </apply>
            </lambda>
         </lambda>
         <lambda var="z">
            <op opname="+">
                <var name="x"/>
                <var name="z"/>
            </op>
         </lambda>
      </apply>
   </lambda>
   <op opname="+">
       <var name="z"/>
       <num value="1"/>
   </op>
</apply>
```

Fig. 10. XLambda code for (λ x.(λ y.λ x. y x)(λ z. x+z)) (z+1)

```
<lambda var="x1">
    <apply>
       <lambda var="z1">
           <op opname="+">
               <op opname="+">
                   <var name="z"/>
                   <num value="1"/>
               </op>
               <var name="z1"/>
           </op>
       </lambda>
       <var name="x1"/>
    </apply>
</lambda>
```

Fig. 11. Result of evaluating expression of Figure 10, λ x1. ((λ z1. z+1+z1) x1)

```
<pair>
   <num value="1"/>
   <pair>
      <num value="2"/>
      <pair>
         <num value="3"/>
         <nil>
      </pair>
   </pair>
</pair>
```

Fig. 12. Representation of list [1,2,3] in XLambda

```
<rest>
   <pair>
      <num value="1"/>
      <pair>
         <num value="2"/>
         <pair>
            <num value="3"/>
            <nil>
         </pair>
      </pair>
   </pair>
 </rest>
```

Fig. 13. Application of *<rest>* on the list of Figure 12

```
<pair>
    <num value="2"/>
    <pair>
       <num value="3"/>
       <nil>
    </pair>
 </pair>
```

Fig. 14. The result of evaluating the expression in Figure 13

For elements <var>, <num> and <nil>, the element itself is returned. For <op> or <boolOp> elements, *Evaluate* is called for each child and the specified mathematical or logical operation is performed on the returned values. The result of an evaluated operation is returned in a <num> element. Therefore, each time the value returned from an evaluation is required, the *@value* attribute of the returned result is taken.

When a <first> element is encountered, the template *Evaluate* is called with its first child and the first child of the result of the evaluation is returned. Likewise, in the processing of a <rest> element, the template *Evaluate* is called with the first child of the <rest> element and the second child of the result of evaluation is returned. If an <is-null> element is the case, the template *Evaluate* is called with its first child to determine whether the result is null or not.

C. Template Replace for Performing β-Reduction and α-Conversion

The template *Replace* whose algorithm is depicted in Figure 7 takes four parameters: the first one is the expression to be reduced (*$redex*), the second one is the name of the bound variable which will be replaced (*$bound*), while the third one is the substitute expression (*$substitute*) that will replace the occurrences of the bound variable. The fourth parameter is *$flag* indicating if substitution will take place. This parameter is necessary in order not to substitute a bound variable of an inner scope which has the same name of the bound variable of an outer one. This template checks all children of the redex to find the bound variables to be replaced and replaces them with the substitute expression. It also considers inner λ-abstractions whose bound variable has the same name with the bound variable that is being replaced. In such a case, this inner λ-abstraction constitutes a local scope and therefore it remains unreduced, only α-conversion is applied when necessary. The conditions requiring α-conversion is considered by the implementation of necessary condition checks to determine if the substitute expression is a single variable having the same name with a bound variable of an inner λ-abstraction. In such a case, all the occurrences of the λ-variable in the inner scope that has the same name will be renamed by concatenating '1' to its name. Another condition is set to check if the substitute expression is not a single variable but a composed expression with free variables in it. In such a case, the occurrences of the bound variable in the inner λ-scope is renamed without further checking to see if there are matching names of the free

```
<first>
   <rest>
      <pair>
         <num value="1"/>
         <pair>
            <num value="2"/>
            <pair>
               <num value="3"/>
               <nil>
            </pair>
         </pair>
      </pair>
    </rest>
 </first>
```

Fig. 15. Nested application of <*first*> and <*rest*> on the list of Figure 12

```
<apply>
   <apply>
      <named_fuction name="append"/>
      <pair>
         <num value="1"/>
         <nil/>
      </pair>
   </apply>
   <pair>
      <num value="2"/>
      <pair>
         <num value="3"/>
         <nil/>
      </pair>
   </pair>
</apply>
```

Fig. 16. XLambda code to append [2,3] to [1]

variables of the substitute expression and the bound variable of the inner scope.

The template *Replace*, uses the *case* structure to determine the type of the node under consideration and uses recursion to check all children of the redex. When the redex incoming within the parameter *$redex* is a <named_function>, a <num>, or a <nil> element representing an identifier, it remains unchanged. In case of a <lambda> element under consideration, the tests discussed above are performed to decide either to leave the λ-abstraction unchanged, or apply substitution of the occurrences of bound variable (*$bound*) and perform rename in case of situations requiring α-conversion. The same tests are applied for the case of the incoming redex is a <var> element that is either a free or a bound variable.

When the incoming redex is one of the elements <apply>, <if>, <op>, <boolOp>, <rest>, <first>, <is-null> or <pair>, the template *Replace* is called for all children recursively.

Figures 8 and 10 show two cases requiring α-conversion (i.e. renaming). Figures 9 and 11 are the result of evaluating these expressions with our λ-reducer.

D. Named Lambda Abstractions

The definitions of some named λ-abstractions such as *append* and *doWhile* are stored in the "auxiliaries.xml" file, whose partial contents is given in Figure 17. Named λ-abstractions are applied on the required parameters by the use of nested <apply> elements.

E. Implementation of the List/Tree Manipulation Functions

A list/tree is implemented using the <pair> element in XLambda. An example of a list is given in Figure 12. The list/tree manipulation functions are *first* to retrieve the first element in a list, *rest* to retrieve the elements other than the first one in a list and *is-null* to check if a given list is empty. Figure 13 shows the application of *rest* on the list given in Figure 12 and Figure 14 shows the result of evaluation. The nested application of *rest* and *first* is given in Figure 15 which yields <num value="2"/> as the result. Figure 16 shows a code fragment to append list [2,3] to another list [1]. The evaluation of this code yields the list [1,2,3] as shown in Figure 12.

IV. Related Work

The distinction of XLambda from the other functional languages in literature such as Haskell [2], Lisp [1] and CAML [3], is that, it combines Web technologies with the merits of a functional language.

Meijer and Shields introduce XMλ in [9] to be used in place of server side languages such as CGI Perl scripts [10], Active Server Pages (ASP) [11] or Java Server Pages (JSP) [12] to generate XML or HTML from query results in case of client applications querying server databases. Their argument is that, mixing static XML Web technologies together with dynamic behavior of server side scripting languages creates a layering up of languages and DOM APIs that are needed to access and manipulate XML documents, and this results in the loss of the connection between the generated documents and their DTDs or Schemas. They suggest the use of a single XML based functional language, namely XMλ, to produce XML or HTML from the query results conforming to the provided DTDs or Schemas. Nevertheless, they have not implemented XMλ yet. On the other hand, XLambda together with its evaluator λ-reducer, is a full-fledged functional language. XLambda can be used in the same way as XMλ is proposed by Meijer and Shields in [9].

Meijer and VanVelzen suggested the use of Haskell for generating dynamic Web content in [13] and named their approach as *Haskell Server Pages* (HSP). They claim to benefit from the functional programming approach of Haskell by implementing HSP as a simple preprocessor to Haskell. However they have problems in pattern matching of XML fragments and in the validation of XML documents against a DTD without resorting to high level languages and APIs. They were planning to remedy these problems by the use of XMλ when it will be implemented [9]. The difference between HSP and XLambda is that HSP is a *server-side* technology, whereas XLambda runs on the *client-side*.

V. Conclusion and Future Work

We described the syntax and operational semantics of a novel functional language XLambda which has XML based syntax. The fact that the operational semantics of XLambda

```
<auxiliaries>
<define_function name="append">
    <lambda var="L1">
        <lambda var="L2">
                <if>
                    <is-null>
                      <var name="L1"/>
                    </is-null>
                    <var name="L2"/>
                    <pair>
                       <first>
                          <var name="L1"/>
                       </first>
                       <apply>
                          <apply>
                             <named_function name="append"/>
                             <rest>
                               <var name="L1"/>
                             </rest>
                          </apply>
                          <var name="L2"/>
                       </apply>
                    </pair>
                </if>
            </lambda>
        </lambda>
 </define_function>

<define_function name="doWhile">
    <lambda var="e">
        <lambda var="b">
            <lambda var="s">
                <if>
                    <apply>
                        <var name="e"/>
                        <var name="s"/>
                    </apply>
                    <apply>
                      <apply>
                        <apply>
                          <named_function name="doWhile"/>
                          <var name="e"/>
                        </apply>
                        <var name="b"/>
                      </apply>
                      <apply>
                          <var name="b"/>
                          <var name="s"/>
                       </apply>
                    </apply>
                    <var name="s"/>
                </if>
            </lambda>
        </lambda>
  </lambda>
</define_function>
</auxiliaries>
```

Fig. 17. The definitions of named abstractions

is implemented completely in XSLT makes XLambda unique among all functional languages and opens up possibilities for XLambda that are not available for other languages. These possibilities include using it as a scripting language in Web browsers without the need for browser add-ons, encoding the meaning of objects as lambda expressions and sending the meaning of an object, together with the object over the Internet to users of the object, and encoding agents that run inside browsers.

For future work, we are planning to explore the implementation of agents that run in client browsers, using XLambda.

References

[1] P. Winston and B. Horn, *Lisp*. Reading, MA: Addison-Wesley, 1989.

[2] Haskell Community. (2009) HaskellWiki. [Online]. Available: http://haskell.org/

[3] INRIA (French National Research Institute For Computer Science). (2009) The CAML Language: Home. [Online]. Available: http://caml.inria.fr/

[4] Ruhsan Onder, "Specification and Implementation of Programming Language Semantics using Extensible Stylesheet Language Transformations," Ph.D. dissertation, Eastern Mediterranean University, Famagusta, North Cyprus, February 2008.

[5] R. Onder and Z. Bayram, "Interpreting Imperative Programming Languages In Extensible Stylesheet Language Transformations (XSLT)," in *Proceedings of the IASTED International Conference on Internet and Multimedia Systems and Applications (EuroIMSA 2005)*, Grindelwald, Switzerland, February 2005, pp. 131–136.

[6] W. W. W. Consortium. (2009) XSL Transformations (XSLT). [Online]. Available: http://www.w3.org/TR/xslt

[7] ——. (2009) W3C XML Schema. [Online]. Available: http://www.w3.org/XML/Schema

[8] H. Barendregt and E. Barendsen, *Introduction to Lambda Calculus*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[9] E. Meijer and M. Shields, "XMλ: A functional language for constructing and manipulating XML documents," 1999, unpublished. [Online]. Available: http://www.cartesianclosed.com/pub/xmlambda/index.html

[10] L. D. Stein. (2009) Official guide to programming with cgi.pm. [Online]. Available: http://www.wiley.com/legacy/compbooks/stein/

[11] S. Walther, *ASP.NET 3.5 Unleashed*. Sams, 2008.

[12] Sun Developer Network. (2009) JavaServer Pages Technology. [Online]. Available: http://java.sun.com/products/jsp/

[13] E. Meijer and D. van Velzen, "Haskell server pages - functional programming and the battle for the middle tier," Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.8202, 2000.