A Lisp Subset Based on MathML

Yuzhen Xie, Stephen Watt, and Luca Padovani
The University of Western Ontario

Abstract

We are interested in an XML representation for programming languages. Could content markup of MathML be used for this purpose, especially in the setting of a functional programming language? In order to address this question we have developed an interpreter for a Scheme-like language encoded in XML. We demonstrate that, with some degree of extension to its core elements, syntax and default semantics of content MathML are quite sufficient to represent Scheme expressions. We also describe the basic architecture of the interpreter, which has been written in Java. Our work suggests that the potential of a web scripting language encoded in XML (possibly based on MathML) is very high. In particular, it offers very natural mechanisms to construct XML expressions dynamically incorporating desired XML subexpressions.

1. Content MathML as the Nucleus of a Functional Programming Language

XML (eXtensible Markup Language) [1] is used to describe structured documents by user-defined markup tags. Due to its focus on structure, XML can be easily used to create a programming language framework where all the programming constructs like control and data flow, functions, format of message passing, are encoded in XML. Programs encoded in this form could immediately have the advantages of XML, in particular standardization of the encoding format, independence of the delivery medium, easiness of parsing by standard software tools.

A natural place to start an investigation into these ideas is with the Lisp family of languages, as Lisp S-Expressions and XML are quite similar. In fact, it has been previously noted that XML documents can be represented as S-Expressions [2]. We address the question of the opposite embedding. To this aim we consider the S-Expressions of the Scheme [3] dialect of Lisp. In another paper [4], Boley has considered content markup in MathML as a sub-language for Lisp-like notation, with a view to develop a markup language for functional relations.

To start with, we investigate the content markup of MathML [5] as an expression language. For example, the Scheme expression `(lambda (x) (+ x x))` can be represented in MathML content markup as:

```
<lambda>
 <bvar><ci>x/<ci>/<bvar>
 <apply>
  <plus/>
  <ci>x</ci>
  <ci>x</ci>
 </apply>
</lambda>
```

Since the intent of content markup in MathML is to provide an explicit encoding of the structure underlying a mathematical expression, it is quite rich in describing lambda constructs and definitions. For instance, the `apply` element corresponds to a complete mathematical expression where an operator (or a function) is applied to its arguments. The `lambda` element is used to construct a function given a list of variables to bind (the functions' arguments) and an expression (the function's body). A `declare` construct associates specific properties or meanings to a mathematical object. Identifiers, numbers and symbols are encoded by `ci`, `cn`, and `csymbol` respectively. `csymbol`, in particular, can be used to extend content MathML when none of the default elements is suitable to represent a specific operator or a specific function.

The characteristics of the container elements in MathML content markup make them adequate to represent expressions, functions, and definitions in a programming language. Container elements are used in a simple, recursive way and constructs can be nested at arbitrary depth. As a more complete example, the Scheme expression `(define reverse-subtract (lambda (x y) (- y x)))` can be encoded as the following MathML content fragment:

```
<declare>
 <ci>reverse-subtract</ci>
 <lambda>
  <bvar><ci>x</ci></bvar>
  <bvar><ci>y</ci></bvar>
  <apply>
   <minus/>
   <ci>y</ci>
   <ci>x</ci>
  </apply>
 </lambda>
</declare>
```

After having established such encoding, the next natural step is the definition of a notion of expression evaluation in MathML. MathML does not encompass any notion of parameter passing. Furthermore, some operators and symbols that are specific to Scheme, such as cons and `'()`, are not part of the core content elements in MathML. But, as we noticed before, content MathML provides a `csymbol` element whose semantics can be defined externally by the user. Such element gives content MathML enough flexibility to be used as a programming language.

There is also an issue about types: MathML types do not correspond directly to the types of Scheme. Although the `cn` element has an attribute `type` that can be used to specify the type of numbers, the data types available are far too simple and not extendible. In particular, there is no way to declare complex data types for pairs or lists in Scheme. Nevertheless, since Scheme is loosely and dynamically typed, data typing can be handled in the XML-Scheme interpreter.

2. XML-Scheme Syntax and Grammar

The XML-Scheme language is described by the content MathML elements with some degree of extension to its elements. The core content elements of MathML are used to represent most of the first class operators, constants and symbols, data flow and control, expressions, and function definitions. As already anticipated, the `csymbol` element is used to create symbols having a particular meaning in Scheme but that are not part of the core content elements in MathML. Even though it would be easier to define tags and semantics for symbols such as <cons>, this approach is not extendible, and treats some functions, like `cons`, differently from other functions like `reverse-subtract`. The core content MathML together with the elements created using `csymbol` form XML-Scheme, which is an XML representation for a Lisp-like subset of functional programming language.

In this study, a few new elements were invented to meet the particular requirements in Scheme. These were represented as a set of csymbols. In practice, these could equally well be elements in a separate namespace. The <xml-scheme> element is used as the root element in a document representing a program written in XML-Scheme. The Scheme constant '() is denoted by the symbol null. The Scheme constructor and destructors for pairs are denoted by the symbols cons, car, and cdr. The Scheme null? operator is denoted by a null? symbol in XML-Scheme.

We define a number of syntactic rules that define expression reduction and parameter passing. An apply construct whose first child is one of the first class operators or lambda construct performs operation on the rest of its children which are operands. An apply construct whose first child is ci indicates that its first child is an identifier to a function, and the rest of its children are the argument list. What follows is a complete example of XML-Scheme program that uses +, -, *, cons, if, lambda, and define. The corresponding Scheme expressions are illustrated in the XML comments.

```
<?xml version="1.0" encoding="UTF-8"?>

<xml-scheme>

 <!-- (define my_pair (cons 1 -1)) -->
 <!-- (define my_abs (lambda (x)
       (if (< x 0) (- x) (+ x)))) -->
 <!-- (define my_double (lambda (x)
       (* 2 x))) -->
 <!-- (define compose-fn (lambda (f1 f2)
       (lambda (n) (f1 (f2 n))))) -->
 <!-- (define my_fun
       (compose-fn my_double my_abs)) -->
 <!-- (my_fun (car my_pair)) -->

 <declare>
  <ci>my_pair</ci>
  <apply>
   <csymbol>cons</csymbol>
   <cn>1</cn>
   <cn>-1</cn>
  </apply>
 </declare>

 <declare>
  <ci>my_abs</ci>
  <lambda>
   <bvar><ci>x</ci></bvar>
   <piecewise>
    <piece>
     <apply>
      <minus/>
      <ci>x</ci>
     </apply>
     <apply>
      <lt/>
      <ci>x</ci>
      <cn>0</cn>
     </apply>
    </piece>
    <otherwise>
     <apply>
```

```
      <plus/>
      <ci>x</ci>
     </apply>
    </otherwise>
   </piecewise>
  </lambda>
 </declare>

 <declare>
  <ci>my_double</ci>
  <lambda>
   <bvar><ci>x</ci></bvar>
   <apply>
    <times/>
    <cn>2</cn>
    <ci>x</ci>
   </apply>
  </lambda>
 </declare>

 <declare>
  <ci>compose-fn</ci>
  <lambda>
   <bvar><ci>f1</ci></bvar>
   <bvar><ci>f2</ci></bvar>
   <lambda>
    <bvar><ci>n</ci></bvar>
    <apply>
     <ci>f1</ci>
     <apply>
      <ci>f2</ci>
      <ci>n</ci>
     </apply>
    </apply>
   </lambda>
  </lambda>
 </declare>

 <declare>
  <ci>my_fun</ci>
  <apply>
   <ci>compose-fn</ci>
   <ci>my_double</ci>
   <ci>my_abs</ci>
  </apply>
 </declare>

 <apply>
  <ci>my_fun</ci>
  <apply>
   <csymbol>car</csymbol>
   <ci>my_pair</ci>
  </apply>
 </apply>

</xml-scheme>
```

The program starts declaring a pair named `my_pair` with value `(1 . -1)` created by the cons operation. Then, a function with the identifier of `my_abs` is defined to calculate the absolute value of a variable. The second function that is named my_double will double a given value. The function that is called compose_fn is composed of two general functions f1 and f2, and a parameter n. The my_fun function is defined by calling the compose_fn with my_double and my_abs as arguments. Finally, the my_fun function is called by passing the car value of my_pair.

3. XML-Scheme Interpreter

The XML-Scheme interpreter was designed in an Object-Oriented model and implemented in Java [6]. Figure 1 shows a diagram with the most important classes of the XML-Scheme interpreter.

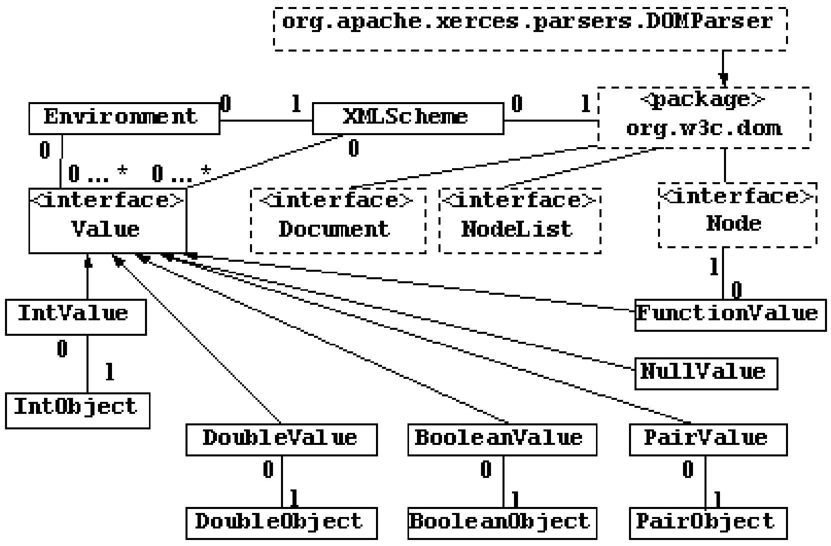

Figure 1 Class Diagrams of XML-Scheme Interpreter

The interpreter parses a program in XML-Scheme and executes it. The Xerces-J DOM parser [7] is used to parse the XML document with embodied Scheme expressions. The result of the parsing step is a DOM Document. The Document is then processed and evaluated in the XMLScheme class. The key operations include value computation, variable binding, and procedure creation and evaluation.

Data types are determined and checked while the evaluation moves on. Typing rules are the same as those in Scheme. Values handled by the interpreter are Java objects implementing the common Value interface. In this program, seventeen Scheme operators were implemented, including +, -, *, /, <, =, define, lambda, if, cons, car, cdr, null?, '(), quote, quasiquote, and unquote. Data types handled by the interpreter include integer, real, Boolean, null and pair.

As procedure are first-class objects in Scheme, there is a class, FunctionValue, for objects representing procedure values. All the MathML operators recognized by the interpreter have a corresponding Java method that implements their semantics. The expression tree associated to define is treated as an identifier corresponding to a FunctionValue. When a procedure is called or an expression is processed, it is recursively evaluated. The Environment class uses a hash table structure to keep track of variable binding by associating each variable to its value. Variables are bound in two circumstances. The first one is in a declaration, which binds a value to a variable. The second one is when a function is applied to one or more arguments. In this case the arguments are evaluated, and their values are bound to the corresponding parameters of the function. It is worth noting that, despite the name "variable", in our interpreter there is no notion of assignment. Variables can be declared multiple times, but the association between a variable and its value is static.

4. Quotation

One of the intriguing possibilities brought out by this work is the potential for using Lisp-style quasi- quoting mechanisms to construct XML documents in a natural, structured manner.

Scripting languages such as PHP [8] provide mechanisms to create document components in an unstructured manner, e.g.

```
<?
   print("<ul>");
   for ($ix = 0; $ix < count($stuff); $ix++)
   {
       print("<li> $ix. $stuff($ix) </li>\n");
   }
?>
```

The above PHP fragment would be included in a web page with some later fragment providing the closing </ul>.

Note that the XHTML of the page would be explicit, and the dynamic content filled in by the print statements in the <? ?> processing directive. On one hand, this provides a familiar model of computation to C programmers. But on the other, it is error-prone and forces programmers and software tools to deal with multiple paradigms.

Another approach would be to adapt the solutions from the "program is data" world view of Lisp. The various dialects of Lisp all have mechanisms for "quoting" data. That is, they provide mechanisms to include data structures in a natural syntax in programs. Most modern Lisps, Scheme included, provide mechanisms for mostly quoted data to include executable code that fills in parts of the structure. This is usually called "quasi quoting." As an example, the following Scheme fragment defines `mylist` to be a quoted list structure with two sub-expressions (preceded by commas) evaluated and spliced in.

```
(set! mylist '(1 2 ,(+ 2 1) 4 ,(* n n) 6))
```

We propose that a useful scripting model for XHTML documents would be to consider a document as quasi-quoted data. Then scripting components would be programs also in XML syntax (e.g. XML- Scheme) with evaluation indicated by an unquoting mechanism exactly analogous to the comma in Scheme. An example is as follows. This document fragment would be part of a larger document, which would be implicitly enclosed in <xs:[ ).

```
<... xmlns:xs="http://www.orcca.on.ca/MathML/XML-Scheme"
        xmlns:m="http://www.w3.org/TR/MathML2">
<ul>
 <li><p>A fragment of quasi-quoted data in XML
format</p></li>
 <li>
  <xs:unquote>
   <m:apply>
    <m:ci>my_fun</m:ci>
    <m:apply>
     <m:csymbol>car</m:csymbol>
     <m:ci>my_pair</m:ci>
    </m:apply>
   </m:apply>
  </xs:unquote>
 </li>
 <li>
  <xs:unquote>
   <m:apply>
    <m:ci>my_fun</m:ci>
    <m:apply>
     <m:csymbol>cdr</m:csymbol>
     <m:ci>my_pair</m:ci>
    </m:apply>
   </m:apply>
  </xs:unquote>
 </li>
</ul>
```

5. Concluding Remarks

The prototype of the XML-Scheme interpreter shows that it is feasible to use content MathML to represent a simple functional programming language. Our XML-Scheme language was defined according to the syntax of MathML content markup. It can describe expressions that are composed of a basic set of Scheme operators, in particular the lambda calculus and definitions. Thanks to the the `csymbol` element, MathML content markup can be extended so to represent a full-featured functional programming language like Scheme. This study indicates that it would be useful to examine the potential uses of programming languages represented in XML. In particular, the quoting/unquoting mechanism described in Section 4 might lead to a new family of scripting languages for Web pages, entirely described in an XML (and possibly MathML) syntax.

References

[1] W3C Recommendation: Extensible Markup Language (XML) 1.0, Feb. 1998.
[2] XML and Scheme, a micro-talk presentation at the Workshop on Scheme and Functional Programming, Sep. 2002.
[3] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme. Feb. 1998.
[4] Harold Boley, Markup Languages for Functional-Logic Programming, Proc. 9th WFLP 2000, Benicassim, Spain, Sep. 2000.
[5] W3C Recommendation: Mathematical Markup Language (MathML) Version 2.0, Feb. 2001.
[6] Ken Arnold, James Gosling, and David Homes. Java^{TM} Programming Language, Addison-Wesley, 2000.
[7] Xerces APACHE: Xerces-J DOM Parser 1.4.4, Jan. 2002.
[8] PHP Manual, Stig Saether Bakken et al, http://www.php.net