A Primer for XOCL

Tony Clark

May 27, 2004

1 Introduction

XMF is an environment for domain specific language design. XMF can be used to define and deploy both diagrammatic languges such as UML and text-based languages. The kernel of XMF is defined in a language called XOCL. XOCL is the language that is used to define XMF and as such is the basic language that XMF developers will use to develop XMF applications including new domain specific languages.

This document is a primer for XOCL. The approach is informal and we aim to introduce features of the language by example, occasionally using features in examples before they are completely introduced.

2 Basic Data Types and Expressions

XOCL provides basic data types for integers and booleans. Operations for arithmetic ($+$, $-$, $/$ and $*$) are supported in addition to the usual relational operators ($<$, $>$, $=$, $<=$ and $>=$). Boolean logic is supported by infix operators `and` and `or` and prefix `not`. The following example shows an operator definition for factorial. The operator is named `fact`, takes a single argument `n` and is defined in the global context `Root` which means that the name `fact` is available everywhere:

```
context Root
  @Operation fact(n)
    if n = 0
    then 1
    else n * fact(n - 1)
    end
  end
```

Another example of a global operation definition is `gcd` below that computes the greatest common divisor for a pair of positive integers. The example shows that operations can optionally have argument and return types:

```
context Root
  @Operation gcd(m:Integer,n:Integer):Integer
    if m = n
    then n
    else
      if m > n
      then gcd(m-n,n)
      else gcd(m,n-m)
      end
    end
  end
```

Integers are represented in 24 bits in XMF. The operators `and` and `or` are overloaded to perform bit comparison when supplied with integers. The operators `lsh` and `rsh` are defined on integers. They take the number of bits to shift left and right respectively. The following operation adds up the number of bits in an integer:

```
context Root
  @Operation addBits(n:Integer):Integer
    if n = 0 or n < 0
    then 0
    else
      (n and 1) + (addBits(n.rsh(1)))
    end
  end
```

In addition, integers support the follosing operators: `mod`, `abs`, `bit`, `max`, `min` and `byte`.

XMF supports floating point numbers. All numeric operators described above are defined for floating point numbers. In general integers and floats can be mixed in expressions and the integer is translated to the equivalent float. You can construct a float using the constructor `Float` so that `3.14` is created by `Float(3,14)`. Floats are translated to integers by the operations `round` (upwards) and `floor` (downwards).

Integer division is performed by the operation `div` and floating point division is performed by the infix operator `/` (translating integers to floats as necessary). The operators `sqrt`, `sin` and `cos` are defined for floats.

All values in XMF support the operation `of` that returns the most specific class of the receiver. A value can be asked whether it is of a given class using the operator `isKindOf`. Classes can be compared using `inheritsFrom` where a class is considered to inherit from itself. We could define `isKindOf` as:

```
context Element
  @Operation isKindOf(type:Classifier):Boolean
    self.of().inheritsFrom(type)
  end
```

The distinguished value `null`, of type `Null`, is special in that it returns `true` when asked whether it `isKindOf` any class. It is used as the default value for all non-basic classes in XMF.

The following operation returns `true` when supplied with a number:

```
context Root
  @Operation isNum(x)
    (x.isKindOf(Integer) or
     x.isKindOf(Float)) and
    x <> null
  end
```

3 Strings, Symbols and Equality

XMF strings are of type `String`. The following operation wraps the string `"The"` and `"."` around a supplied string:

```
context Root
  @Operation makeSentence(noun:String):String
    "The " + noun + "."
  end
```

Strings are sequences of characters indexed starting from 0. Equality of strings is defined by `=` on a character by character comparison. Characters are represented as integer ASCII codes. The following operation checks whether a string starts with an upper case char:

```
context Root
  @Operation startsUpperCase(s:String):Boolean
    if s->size > 0
    then
      let c = s->at(0)
      in "A"->at(0) <= c and c <= "Z"->at(0)
      end
    else false
    end
  end
```

Strings can be compared using `<`, `<=`, `>` and `>=` in which case the usual lexicographic ordering applies.

Since strings are compared on a character by character basis this makes string comparison relatively inefficient when performing many comparisons. Strings are often used as the keys in lookup tables (for example as the names of named elements). In order to speed up comparison, XMF provides a specialization of `String` called `Symbol`. A symbol is the same as a string except that two symbols with the same sequence of characters have the same identity.

Comparison of symbols by identity rather than character by character is much more efficient. A string `s` can be converted into a symbol by `Symbol(s)`.

Any value can be converted into a string using the operation `toString`. To get the string representation of a number for example: `100.toString()`.

4 Tables

A table is used to associate keys with values. A table has operations to add associations between keys and values and lookup a given key. A table is created using the `Table` constructor supplying a single argument that indicates the approximate number of elements to be stored in the table. Suppose that a library maintains records on borrowers:

```
context Root
  @Class Library
    @Attribute borrowers : Table = Table(100) end
  end
```

A new borrower is added by supplying the id, name and address. When the new borrower is added we check that no borrower has been allocated the same id. If the id is not already in use then we register the new borrower by associating the id with a borrower record in the table:

```
context Library
  @Operation newBorrower(id:String,name:String,address:String)
    if not borrowers.hasKey(id)
    then
      let borrower = Borrower(id,name,address)
      in borrowers.put(id,borrower)
      end
    else self.error("Borrower with id = " + id + " already exists.")
    end
  end
```

The library also provides an operation that gets a borrower record:

```
context Library
  @Operation getBorrower(id:String):Borrower
    if borrowers.hasKey(id)
    then borrowers.get(id)
    else self.error("No borrower with id = " + id)
    end
  end
```

Tables provide operations to get all the keys and all the values:

```
context Library
  @Operation idsInUse():Set(String)
```

```
    borrowers.keys()
  end

context Library
  @Operation allBorrowers():Set(Borrower)
    borrowers.values()
  end
```

5 Sets

A set is an unordered collection of elements. The elements of a set need not all be of the same type. When `T` is the type of each element of the set then the set is of type `Set(T)`. Operations are provided to add and remove elements from sets and to combine sets. Sets can be used in iterate expressions. This section describes set-specific operations; iteration is described in section 8.

Sets are created by evaluating a set expression of the form: `Set{x,y,z,...}` where `x`, `y`, `z` etc are element expressions. For example:

```
Set{1,true,Set{"a","b","c"},Borrower("1","Fred","3 The Cuttings")}
```

The expression `Set{}` produces the empty set. A set is unordered. An element can be selected at random from a non-empty set by performing `S->sel`. A set is empty when `S->isEmpty` produces true (or when it is `=` to `Set{}`). An element `e` is added to a set by `S->including(e)` and removed from a set by `S->excluding(e)`. The union of two sets is produced by `S1 + S2` and the difference is constructed by `S1 - S2`. An element is contained in a set when `S->includes(e)`.

Suppose that the set operation `includes` was not provided as part of XOCL. It could be defined by:

```
context Set(Element)
  @Operation includes(e:Element):Boolean
    if self->isEmpty
    then false
    else
      let x = self->sel
      in if x = e
         then true
         else self->excluding(x)->includes(e)
         end
      end
    end
  end
```

6 Sequences

A sequence is an ordered collection of elements. The elements in the sequence need not all be of the same type. When `T` is the type of each element in the sequence then the sequence is of type `Seq(T)`. Sequences can be used in iterate expressions as described in section **??**. This section describes sequence operations.

Sequences are created by evaluating a sequence expression or by translating an existing element into a sequence. Sets, strings, integers and vectors can be translated to sequences of elements, characters, bits and elements respectively by performing `e.asSeq()`.

The following operations are defined on sequences: `+` appends sequences; `asSet` transforms a sequence into a set; `asString` transforms a sequence of character codes into a string; `asVector` transforms a sequence into a vector; `at` takes an index and returns the element at that position in the sequence, it could be defined as:

```
context Seq(Element)
  @Operation at(n:Integer):Element
    if self->size = 0
    then self.error("Seq(Element).at: empty sequence.")
    else if n <= 0
         then self->head
         else self->tail.at(n - 1)
         end
    end
  end
```

The operation `butLast` returns all elements in a sequence but the last element. It could have been defined as follows, note the use of `Seq{head | tail}` to construct a sequence with the given head and tail:

```
context Seq(Element)
  @Operation butLast():Seq(Element)
    if self->size = 0
    then self.error("Seq(Element)::butLast: empty sequence.")
    else if self->size = 1
         then Seq{}
         else Seq{self->head | self->tail->butLast}
         end
    end
  end
```

The operation `contains` returns true when a sequence contains a supplied element; `drop` takes an integer and returns a sequence that is the result of dropping the supplied number of elements; `flatten` maps a sequence of sequences to a sequence:

```
context Seq(Element)
  @Operation flatten():Seq(Element)
    if self->isEmpty
    then self
    else self->head + self->tail->flatten
    end
  end
```

The operation `hasPrefix` takes a sequence as an argument and returns true when the receiver has a prefix that is equal to the argument; `including` takes an element and returns a new sequence, this is the receiver if it contains to the argument or the argument prepended to the receiver; `indexOf` takes an element and returns the index of the argument in the receiver:

```
context SeqOfElement
  @Operation indexOf(element:Element):Integer
    if self = Seq{}
    then -1
    else
      if self->head = element
      then 0
      else self->tail->indexOf(element) + 1
      end
    end
  end
```

The operation `insertAt` takes an element and an index and inserts the element at the given index; `last` returns the last element in a sequence; `hasSuffix` takes a sequence as an argument and returns true when the receiver ends with the argument:

```
context Seq(Element)
  @Operation hasSuffix(suffix):Boolean
    self->reverse->hasPrefix(suffix->reverse)
  end
```

The operation `head` returns the head of a non-empty sequence; `includes` returns true when the argument is included in the receiver; `isEmpty` returns true when the receiver is empty; `isProperSequence` returns true when the final element in the sequence is a pair whose tail is `Seq{}`; `map` applies an operation to each element of a sequence, the definition of `map` shows the *varargs* feature of XOCL operations where the last argument may be preceded by a `.` indicating that any further supplied arguments are bundled up into a sequence and supplied as a single value:

```
context Seq(Element)
  @Operation map(message:String . args:Seq(Element)):Element
    self->collect(x | x.send(message,args))
  end
```

The operation `max` finds the maximum of a sequence of integers; `prepend` adds an element to the head of a sequence; `qsort` takes a binary predicate operation and sorts the receiver into an order that satisfies the predicate using the quicksort algorithm:

```
context Seq(Element)
  @Operation qsort(pred):Seq(Element)
    if self->isEmpty
    then self
    else
      let e = self->head
      in let pre = self->select(x | pred(x,e));
             post = self->select(x | x <> e and not pred(x,e))
         in pre->sort(pred) + Seq{e} + post->sort(pred)
         end
      end
    end
  end
```

The operation `ref` can be used to lookup a namespace path represented as a sequence of strings to the element found at the path. The operation takes a sequence of namespaces as an argument; the namespace arguments are used as the basis for the lookup, for example:

```
Seq{"Root","EMOF","Class","attributes"}->ref(Seq{Root})
```

returns the attribute named `attributes`. Note that namespace `Root` contains itself.

The operation `reverse` reverses the receiver:

```
context Seq(Element)
 @Operation reverse():Seq(Element)
    if self->isEmpty
    then Seq{}
    else self->tail->reverse + Seq{self->head}
  end
```

The operation `separateWith` takes a string as an argument and returns the string that is formed by placing the argument between each element of the receiver after the element is transformed into a string using `toString`.

The operation `subst` takes three arguments: `new`, `old` and `all`; it returns the result of replacing element `old` with `new` in the receiver. If `all` is true then all elements are replaced otherwise just the first element is replaced.

The operation `subSequence` takes a staring and terminating indices and returns the appropriate subsequence; `take` takes an integer argument and returns the prefix of the receiver with that number of elements; `tail` returns the tail of a non-empty sequence.

Sequences have identity in XOCL; the head and tail of a sequence can be updated using:

```
S->tail := e
```

and

```
S->tail := e
```

This makes sequences very flexible and can be used for efficient storage and update of large collections (otherwise each time a sequence was updated it would be necessary to copy the sequence).

7 A-Lists

An *a-list* is a sequence of pairs; each pair has a head that is a key and a tail that is the value associated with the key in the a-list. A-lists are used as simple lookup tables. They are much more lightweight than instances of the class `Table` and have the advantage that the large number of builtin sequence operations apply to a-lists.

The following class shows how an a-list can be used to store the age of a collection of people:

```
context Root
  @Class People
    @Attribute table : Seq(Element) end
    @Operation newPerson(name:String,age:Integer)
      self.table := table->bind(name,age)
    end
    @Operation getAge(name:String):Integer
      table->lookup(name)
    end
    @Operation hasPerson(name:String):Boolean
      table->binds(name)
    end
    @Operation birthday(name:String)
      // Assumes name is in table:
      table->set(name,table->lookup(name) + 1)
    end
  end
```

8 Iteration

Iteration expressions in XOCL allow collections (sets and sequences) to be manipulated in a convenient way. Iteration expressions are a shorthand for higher-order operations that take an operation as an argument and apply the argument operation to each element of the collection in turn. As such, iteration expressions can be viewed as sugar for the invocation of the equivaent higher-order

operations. This section defines the XOCL iteration expressions and shows how they can be defined as higher-order operations.

A collection can be filtered using a `select` expression:

```
S->select(x | e)
```

where `x` is a variable and `e` is a predicate expression. The result is the sub-collection of the receiver where each element `y` in the sub-collection satisfies the predicate when `x` is bound to `y`. This can be defined as follows for sequences:

```
context Seq(Element)
  @Operation select(pred:Operation):Seq(Element)
    if self->isEmpty
    then self
    else
      if pred(self->head)
      then Seq{self->head | self->tail.select(pred)}
      else self->tail.select(pred)
      end
    end
  end
```

The `reject` expression is like `select` except that it produces all elements that fail the predicate; here is the definition of `refject` for sets:

```
context Set(Element)
  @Operation reject(pred:Operation):Set(Element)
    if self->isEmpty
    then self
    else
      let x = self->sel
      in if pred(x)
         then self->excluding(x)->reject(pred)
         else self->excluding(x)->reject(pred)->including(x)
         end
      end
    end
  end
```

The `collect` expression maps a unary operation over a collection:

```
S->collect(x | e)
```

It is defined for sequences as follows:

```
context Seq(Element)
  @Operation collect(map:Operation):Seq(Element)
    if not self->isEmpty
    then
```

```
      let x = self->sel
      in self->excluding(x)->select(map)->including(map(x))
      end
    else self
    end
  end
```

The expression `iterate` has the form:

```
S->iterate(x y = v | e)
```

This expression steps through the elements of `S` and repeatedly sets the value of `y` to be the value of `e` where `e` may refer to `x` and tt y. The initial value of `y` is the value `v`. The following shows an example that adds up the value of a sequence of integers:

```
Seq{1,2,3,4,5}->iterate(i sum = 0 | sum + i)
```

Th definition of `iterate` as a higher order operation on sequences is as follows:

```
context Seq(Element)
  @Operation iterate(y:Element,map:Operation):Element
    if self->isEmpty
    then y
    else self->tail.iterate(map(self->head,y)map)
    end
  end
```

A XOCL Grammar

```
AName ::= Name | Drop.

Apply ::= PathExp Args | KeyArgs .

ArithExp ::= UpdateExp [ ArithOp ArithExp ].

ArithOp ::= '+' | '-' | '*' | '/'.

Args ::= '(' (')' | Exp (',' Exp)* ')'.

AtExp ::= '@' AtPath @ 'end'.

AtPath ::= Name ('::' Name)*.

Atom ::= VarExp | 'self' | Str | Int | IfExp | Bool | LetExp |
         CollExp | AtExp | Drop | Lift | '(' Exp ')' | Throw | Try |
         ImportIn | Float.

AtomicPattern ::= Varp | Constp | Objectp | Consp | Keywordp.
```

```
Binding ::= AName '=' LogicalExp.

Bindings ::= Binding (';' Binding)*.

Bool ::= 'true' | 'false'.

CollExp ::= SetExp | SeqExp.

CompareExp ::= ArithExp [ CompareOp CompareExp ].

CompareOp ::= '='| '<' | '>' | '<>' | '>=' | '<='.

CompilationUnit ::= ParserImport* Import* (Def | TopLevelExp)* EOF.

Consp ::= Pairp | Seqp | Emptyp.

Constp ::=  Int | Str | Bool | Expp.

Def ::= 'context' PathExp Exp.

Drop ::= '<' Exp '>'.

EmptyColl ::= Name '{' '}'.

Emptyp ::= Name '{' '}'.

Exp ::= OrderedExp.

Expp ::= '[' Exp ']'.

Float ::= Int '.' Int.

Import ::= 'import' TopLevelExp.

ImportIn ::= 'import' Exp 'in' Exp 'end'.

ParserImport ::= 'parserImport' Name ('::' Name)* ';' ImportAt.

IfExp ::= 'if'Exp 'then'Exp IfTail.

IfTail ::= 'else' Exp 'end' | 'elseif' Exp 'then' Exp IfTail | 'end'.

KeyArgs ::= '[' (']' | KeyArg (',' KeyArg)* ']'.

KeyArg ::= Name '=' Exp.

Keywordp ::= Name ('::' Name)* '[' Keyps ']'.

Keyps ::= Keyp (',' Keyp)* | .
```

```
Keyp ::= Name '=' Pattern.

Lift ::= '[|' Exp '|]'.

LetBody ::= 'in'Exp| 'then' Bindings LetBody.

LetExp ::= 'let'Bindings LetBody 'end'.

LogicalExp ::= NotExp [ LogicalOp LogicalExp ].

LogicalOp ::= 'and' | 'or' | 'implies'.

NonEmptySeq ::= Name '{' Exp ((',' Exp)* '}' | '|' Exp '}').

NonEmptyColl ::= Name '{' Exp (',' Exp)* '}'.

NotExp ::= CompareExp | 'not' CompareExp.

Objectp ::= Name ('::' Name)* '(' Patterns ')'.

OrderedExp ::= LogicalExp [ ';' OrderedExp ].

OptionallyArgs ::= Args | .

Pairp ::= Name '{' Pattern '|' Pattern '}'.

PathExp ::= Atom [ '::' AName ('::' AName)* ].

Pattern ::= AtomicPattern ('->' Name '(' Pattern ')')* ('when'  Exp  | ).

Patterns ::= Pattern (',' Pattern)* | .

RefExp ::= Apply
    (
      '->'
       (
          'iterate' '(' AName AName '=' Exp '|' Exp ')'
        |
         AName
          (
            OptionallyArgs
          |
            '(' AName '|' Exp ')'
          |

          )
        )

      |
        '.' AName
```

```
      (
        Args
      |

      )
  )*.

Seqp ::=  Name '{' Pattern SeqpTail.

SeqpTail ::= ',' Pattern SeqpTail | '}'.

SeqExp ::= EmptyColl | NonEmptySeq.

SetExp ::= EmptyColl | NonEmptyColl.

Throw ::= 'throw' LogicalExp.

TopLevelExp ::= LogicalExp ';'.

Try ::= 'try' Exp 'catch' '(' Name ')' Exp 'end'.

UpdateExp ::= RefExp (':=' ! LogicalExp | ).

VarExp ::= Name Token.

Varp ::= AName ('=' Pattern | ) (':' Exp | ).
```