Patterns in XMF

Tony Clark

April 27, 2004

1 Introduction

XMF is an environment for domain specific language development. The basic XMF environment is supplied with a language, called XOCL, that is based on the OMG standard OCL language. XOCL is defined using the XMF language development technologies and adds a number of features to OCL including: higher order functions, syntax classes, meta-circularity, slot update, object creation, threads and pattern matching. The purpose of this document is to describe the pattern matching facilities in XOCL.

XOCL aims to support the rapid development of applications using a rich collection of language features. XOCL abstracts away from the details of *how* data is accessed and transformed by providing *pattern matching* language features. Patterns provide an information-centric focus of attention by emphasising *what* is being accessed and transformed rather than *how* the data is navigated and modified. In this sense, XOCL provides an abstraction layer that increases productivity and reduces the likelihood of low-level programming errors.

2 Patterns

A pattern is matched against a value. The pattern match may succeed or fail in a given matching context. A matching context keeps track of any variable bindings generated by the match and maintains choice points for backtracking if the current match fails.

Pattern matching can be viewed as being performed by a *pattern matching machine* that maintains the current pattern matching context as its state. The engine state consists of a stack of patterns to be matched against a stack of values, a collection of variable bindings and a stack of choice points. A choice point is a machine state. At any given time there is a pattern at the head of the pattern stack and a value at the head of the value stack. The machine executes by performing state transitions driven by the head of the pattern stack: if the outer structure of the pattern matches that of the value at the head of the value stack then:

- 0 or more values are bound.

- 0 or more choice points are added to the choice point stack.
- 0 or more component patterns are pushed onto the pattern stack.
- 0 or more component values are pushed onto the value stack.

If the machine fails to match the pattern and value at the head of the respective stacks then the most recently created choice point is popped and becomes the new machine state. Execution continues until either the pattern stack is exhausted or the machine fails when the choice stack is empty.

The rest of this section describes the different categories of pattern. The semantics of matching are defined informally in terms of a general description and example definitions involving the pattern.

Variables A variable pattern consists of a name, optionally another pattern and optionally a type. The simplest form of variable pattern is just a name, for example, the formal parameter `x` is a variable pattern:

```
let add1 = @Operation(x) x + 1 end in ...
```

Matching a simple variable pattern such as that shown above always succeeds and causes the name to be bound to the corresponding value. A variable may be qualified with a type declaration:

```
let add1 = @Operation(x:Integer) x + 1 end in ...
```

which has no effect on pattern matching. A variable may be qualified with a pattern as in `x = <Pattern>` where the pattern must occur before any type declaration. Such a qualified variable matches a value when the pattern also matches the value. Any variables in the pattern *and* `x` are bound in the process.

Constants A constant pattern is either a string, an integer, a boolean or an expression (in the case of an expression the pattern consists of [followed by an expression followed by]). A constant pattern matches a value when the values is equal to the constant (in the case of an expression the matching process evaluates the expression each time the match occurs). For example:

```
let fourArgs = @Operation(1,true,"three",x = [2 + 2]) x end in ...
```

is an operation that succeeds in the case:

```
fourArgs(1,true,"three",4)
```

and returns `4`.

Sequences A sequence pattern consists of either a pair of patterns or a sequence of patterns. In the case of a pair:

```
let head = @Operation(Seq{head | tail}) head end in ...
```

the pattern matches a non-empty sequence whose head must match the head pattern and whose tail must match the tail pattern. In the case of a sequence of patterns:

```
let add3 = @Operation(Seq{x,y,z}) x + y + z end in ...
```

the pattern matches a sequence of exactly the same size where each element matches the corresponding pattern.

Constructors A constructor pattern matches an object. A constructor pattern may be either a by-order-of-arguments constructor pattern (or BOA-constructor pattern) or a keyword constructor pattern.

A BOA-constructor pattern is linked with the constructors of a class. It has the form:

```
let flatten = @Operation(C(x,y,z)) Seq{x,y,z} end in ...
```

where the class `C` must define a 3-argument constructor. A BOA-constructor pattern matches an object when the object is an instance of the class (here `C` but in general defined using a path) and when the object's slot values identified by the constructor of the class with the appropriate arity match the corresponding sub-patterns (here `x`, `y` and `z`).

A keyword constructor pattern has the form:

```
let flatten = @Operation(C[name=y,age=x,address=y]) Seq{x,y,z} end in ...
```

where the names of the slots are explicitly defined in any order (and may be repeated). Such a pattern matches an object when it is an instance of the given class and when the values of the named slots match the appropriate sub-patterns.

Conditions A conditional pattern consists of a pattern and a predicate expression. It matches a value when the value matches the sub-pattern and when the expression evaluates to true in the resulting variable context. For example:

```
let repeat = @Operation(Seq{x,y} when x = y) Seq{x} end in ...
```

Note that the above example will fail (and probably throw an error depending on the context) if it is supplied with a pair whose values are different.

Sets Set patterns consist of an element pattern and a residual pattern. A set matches a pattern when an element can be chosen that matches the element pattern and where the rest of the set matches the residual pattern. For example:

```
let choose = @Operation(S->including(x)) x end in ...
```

which matches any non-empty set and selects a value from it at random.

Set patterns introduce choice into the current context because often there is more than one way to choose a value from the set that matches the element pattern. For example:

```
let chooseBigger = @Operation(S->including(x),y where x > y) x end in ...
```

Pattern matching in `chooseBigger`, for example:

```
chooseBigger(Set{1,2,3},2)
```

starts by selecting an element and binding it to `x` and binding `S` to the rest. In this case suppose that `x = 1` and `S = Set{2,3}`. The pattern `y` matches and binds `2` and then the condition is applied. At this point, in general, there may be choices left in the context due to there being more than one element in the set supplied as the first parameter. If the condition `x > y` fails then the matching process jumps to the most recent choice point (which in this cases causes the next element in the set to be chosen and bound to `x`). Suppose that `3` is chosen this time; the condition is satisfied and the call returns `3`.

3 Pattern Contexts

Patterns may be used in the following contexts:

Operation Parameters Each parameter in an operation definition is a pattern. Parameter patterns are useful when defining an operation that must deconstruct one or more values passed as arguments. Note that if the pattern match fails then the operation invocation will raise an error.

Operations defined in the same class and with the same name are merged into a single operation in which each operation is tried in turn when the operation is called via an instance of the class. Therefore in the following example:

```
@Class P
  @Operation f(Seq{}) 0 end
  @Operation f(Seq{x | t}) x + self.f(t) end
end
```

an instance of `P` has a single operation `f` that adds up all the elements of a sequence.

Case Arms A case expression consists of a number of arms each of which has a sequence of patterns and an expression. A case expression dispatches on a sequence of values and attempts to match them against the corresponding patterns in each arm in turn. For example, suppose we want to calculate the set of duplicated elements in a pair of sets:

```
context Root
  @Operation dups(s1,s2)
    @Case s1,s2 of
      s1->including(x),s2->including(y) when x = y do
        Set{x} + dups(s1,s2)
      end
      s1->including(x),s2 do
        dups(s1,s2)
      end
      s1,s2->including(y) do
        dups(s1,s2)
      end
      Set{},Set{} do
        Set{}
      end
    end
  end
```

Mapping Clauses Mappings contains clauses. A clause is a named case arm and therefore mappings can contain patterns. The following mapping transforms a class to a sequence of all the parent classes including itself:

```
  @Map AllClasses(EMOF::Class)->Seq(EMOF::Class)
    @Clause MapClass
      c = EMOF::Class[parents = P] do
        Seq{c} + S where S = self(P)
    end
    @Clause MapClasses
      P->including(p) do
        self(p) + self(P)
    end
    @Clause MapEmpty
      Set{} do
      Seq{}
    end
  end
```

4 Syntax

The following BNF grammar defines the language of patterns in XOCL. The following conventions have been used: terminals are surrounded by single quotes; non-terminals start with a capital letter; non-terminals Integer, String and Boolean are assumed; $*$ is used postfix to mean 0 or more repetitions; [and] surround optional elements.

```
Pattern ::= AtomicPattern ('->including(' Pattern ')')* [ 'when' Exp ]

AtomicPattern ::= Varp | Constp | Cnstrp | Seqp | Setp

Varp ::= Name [ '=' Pattern ] [ ':' Type ]

Constp ::= Integer | String | Boolean | '[' Exp ']'

Cnstrp ::= BOAp | Keyp

BOAp ::= Path '(' [ Pattern (',' Pattern)* ] ')'

Keyp ::= Path '[' [ Name '=' Pattern (',' Name '=' Pattern ] ']'

Seqp ::= 'Seq{' [ Pattern (',' Pattern)* ] '}' | 'Seq{' Pattern '|' Pattern '}'

Setp ::= 'Set{}'
```