Developing A Command Language Using XMF

Tony Clark

May 23, 2004

1 The Language Design Space

Language is something that we use to convey meaning. A language consists of a collection of syntax rules that determine the form of communication in the language and a collection of semantic rules that associate communications with meaning. In order that a language definition is well-formed, it must have both syntax and semantics.

Syntax rules can take a number of forms depending on the intended phrase structure of the language. Syntax may be *concrete* (human-centric) or *abstract* (machine-centric). Standard technologies have been developed for expressing textual concrete syntax (the BNF family) and to a lesser extent for expressing graphical concrete syntax (such as graph grammars).

Semantic rules offer a greater variety of options depending on the nature of the language. In broad terms, languages are either *operational* or *non-operational*. An operational language is one that controls the execution of some form of machine and as a result has semantics that relate phrases in the language to machine executions. A non-operational language is not restricted in terms of its semantics. This note is concerned with operational languages.

The semantics of an operational language is defined by relating the syntax domain of the language to a semantic domain. A number of different styles can be used for the operational definition. Two extreme examples are:

1. The semantic domain is defined as a set of values produced by performing the syntax phrases. The semantic relationship takes the form of an interpreter written in some other language. This is a very attractive way of defining the operational semantics of a language since the semantics can be implemented directly. A drawback of this approach is that the interpreter tends to be relatively complex and the semantics of the auxiliary language must be included in the language definition.

2. The semantic domain is defined as a set of machine calculations produced by executing the syntax phrases on the intended target machine. This is to be contrasted with writing an interpreter that maps the syntax domain to the semantic domain. The interpreter approach leads to a complex semantic relationship and a simple semantic domain whereas the complexity is reversed using the calculational approach.

This note is concerned with writing interpreters (translators and compilers).

When defining an interpretive semantics for a language we often have a similar language at hand (at least the language we are writing the interpreter in). We then have the choice as to whether the syntax structures are directly interpreted or whether they are translated to syntax phrases in the existing language. The advantage of a translational approach is that the execution engine for the target language already exists and is probably more efficient than any interpreter we could write for the source language. Compilers are translators for languages where the target language is usually highly optimized and very different from the source language. In principle, however, there is no essential difference between a transator and a compiler. A language that uses the translational approach to target a sub-language of itself is often referred to as *desugaring* the source language.

This note describes how XMF technologies can be used to support a translational approach to language development. A simple command language is used as the case study.

2 XCom - A Simple Command Language

XCom is a simple command language with values that are either records or are atomic. An atomic data value is a string, integer or boolean. A record is a collection of named values. XCom is block-structured where blocks contain type definitions and value definitions. XCom has simple control structures: conditional statements and loops. The following is a simple example XCom program:

```
begin
  type Pair is head tail end;
  type Nil is end;
  value length is 100 end;
  value list is new Nil end;
  while length > 0 do
    begin
      value pair is new Pair end;
      pair.head := length;
      pair.tail := list;
      list := pair
    end
end
```

The definition of XCom is structured as a collection of XMF packages. The `Values` package defines the semantic domain for XCom; it contains classes for each type of program value. Executable program phrases in XCom are divided into two categories: `Expressions` and `Statements`. Expressions evaluate to produce XCom values. Statements are used to control the flow of execution and to update values.

```
@Package XCom
  @Package Values end
  @Package Expressions end
  @Package Statements end
end
```

The rest of this section defines the syntax of XCom by giving the basic class definitions and the XBNF grammar rules for the language constructs.

2.1 XCom Values

XCom expressions evaluate to produce XCom values. Values are defined in the `Values` package and which is the *semantic domain* for XCom. Values are either atomic: integers and booleans, or are records. We use a simple representation for records: a sequence of values indexed by names.

XCom records are created by instantiating XCom record types. A record type is a sequence of names. Types raise an interesting design issue: should the types be included as part of the semantic domain since evaluation of certain XCom program phrases give rise to types that are used later in the execution to produce records. The answer to the question involves the phase distinction that occurs between *static* analysis (or execution) and *dynamic* execution. Types are often viewed as occurring only during static analysis; although this is not always the case. We will show how the semantics of XCom can be defined with and without dynamic types.

All values are instances of sub-classes of the class `Value`:

```
context Values
 @Class Value
   @Attribute value : Element end
   @Constructor(value) ! end
 end
```

Atomic values are either booleans or integers. Each class defines operations that the semantic domain provides for manipulating XCom values. The classes below show the structure and a representative sample of operations:

```
context Values
  @Class Bool extends Value
    @Operation binAnd(Bool(b))
      Bool(value and b)
    end
    @Operation binOr(Bool(b))
      Bool(value or b)
    end
  end

context Values
  @Class Int extends Value
```

```
    @Operation binAdd(Int(n))
      Int(value + n)
    end
  end
```

Record types are sequences of names. A type provides a `new` operation that instantiates the type to produce a new record. This operation is only meaningful if we have dynamic types:

```
context Values
  @Class Type extends Value
    @Attribute names : Seq(String) end
    @Constructor(names) ! end
    @Operation new()
      Record(self,names->collect(n | Seq{n | null}))
    end
  end
```

Records are sequences of values indexed by names; the names are found by navigating to the type of the record:

```
context Values
  @Class Record extends Value
    @Attribute type : Type end
    @Attribute fields : Seq(Element) end
    @Constructor(type,fields) ! end
    @Operation lookup(name:String)
      fields->at(type.names->indexOf(name))
    end
    @Operation update(name:String,value:Element)
      fields->setAt(type.names->indexOf(name),value)
    end
  end
```

2.2 XCom Expressions

XCom expressions are program phrases that evaluate to produce XCom values. The following classes define the expression types:

```
context Expressions
  @Class Exp
  end
end
```

A binary expression has a left and right sub-expression and an operation. The name of the operation is represented as a string:

```
context Expressions
```

```
  @Class BinExp extends Exp
    @Attribute op : String end
    @Attribute left : Exp end
    @Attribute right : Exp end
    @Constructor(op,left,right) ! end
  end
```

An atomic constant expression is either an integer or a boolean:

```
context Expressions
  @Class Const extends Exp
    @Attribute value : Element end
    @Constructor(value) ! end
  end
end
```

A new record is produced by performing a `new` expression. The type to instantiate is given as a string. An alternative representation for types in `new` expressions would be to permit an arbitrary expression that *evaluates* to produce a type. This design choice would rule out static typing and force the language to have dynamic types. We wish to use XCom to illustrate the difference between dynamic and static types in semantic definitions so we use strings to name types in `new` expressions:

```
context Expressions
  @Class New extends Exp
    @Attribute type : String end
    @Constructor(type) ! end
  end
end
```

A variable is just a name:

```
context Expressions
  @Class Var extends Exp
    @Attribute name : String end
    @Constructor(name) ! end
  end
```

A record field ref is:

```
context Expressions
  @Class FieldRef extends Exp
    @Attribute value : Exp end
    @Attribute name : String end
    @Constructor(value,name) ! end
  end
```

The concrete syntax of expressions is defined by the XBNF grammar for the class `Exp`. The grammar parses the expression syntax and synthesizes instances of the expression classes:

```
context Exp
 @Grammar
   // Start at Exp. Logical operators bind weakest.
   Exp ::= e = ArithExp [ op = LogicalOp l = Exp { BinExp(op,e,l) } ].
   LogicalOp ::= 'and' { "and" } | 'or' { "or" }.
   // The '.' for field ref binds tighter than '+' etc.
   ArithExp ::= e = FieldRef [ op = ArithOp a = FieldRef { BinExp(op,e,a) } ].
   ArithOp ::= '+' { "+" }.
   // A field reference '.' optionally follows an atomic expression.
   FieldRef ::= e = Atom ('.' n = Name { FieldRef(e,n) } | { e }).
   // Atomic expressions can be arbitrary exps if in ( and ).
   Atom ::= Const | Var | New | '(' Exp ')'.
   Const ::= IntConst | BoolConst.
   IntConst ::= i = Int { Const(i) }.
   BoolConst ::= 'true' { Const(true) } | 'false' { Const(false) }.
   Var ::= n = Name { Var(n) }.
   New ::= 'new' n = Name { New(n) }.
 end
```

2.3 XCom Statements

XCom statements are used to:

- Introduce new names associated with either types or values.
- Control the flow of execution.
- Perform side effects on records.

The following classes define the statement types for XCom:

```
context Statements
  @Class Statement
  end
end
```

A block (as in Pascal or C) contains local definitions. Names introduced in a block are available for the rest of the statements in the block (including sub-blocks) but are not available when control exits from the block:

```
context Statements
  @Class Block extends Statement
    @Attribute statements : Seq(Statement) end
    @Constructor(statements) ! end
  end
end
```

A declaration introduces either a type or a value binding:

```
context Statements
  @Class Declaration isabstract extends Statement
    @Attribute name : String end
  end
end
```

A type declaration associates a type name with a sequence of field names. To keep things simple we don't associate fields with types:

```
context Statements
  @Class TypeDeclaration extends Declaration
    @Attribute names : Seq(String) end
    @Constructor(name,names) ! end
  end
end
```

A value declaration associates a name with a new value. The value is produced by performing an expression at run-time:

```
context Statements
  @Class ValueDeclaration extends Declaration
    @Attribute value : Exp end
    @Constructor(name,value) ! end
  end
end
```

A while statement involves a test and a body:

```
context Statements
  @Class While extends Declaration
    @Attribute test : Exp end
    @Attribute body : Statement end
    @Constructor(test,body) ! end
  end
end
```

An if statement involves a test, a then-part and an else-part:

```
context Statements
  @Class If extends Declaration
    @Attribute test : Exp end
    @Attribute thenPart : Statement end
    @Attribute elsePart : Statement end
    @Constructor(test,elsePart) ! end
  end
end

context Statement
  @Grammar extends Exp.grammar
```

```
  Statement ::= Block | Declaration | While | If.
  Block ::= 'begin' s = Statement* 'end' { Block(s) }.
  Declaration ::= TypeDeclaration | ValueDeclaration.
  TypeDeclaration ::= 'type' n = Name 'is' ns = Name* 'end' {
    TypeDeclaration(n,ns) }.
  ValueDeclaration ::= 'value' n = Name 'is' e = Exp 'end' {
    ValueDeclaration(n,e) }.
  While ::= 'while' e = Exp 'do' s = Statement 'end' {
    While(e,s) }.
  If ::= 'if' e = Exp 'then' s1 = Statement 'else' s2 = Statement 'end' {
    If(e,s1,s2) }.
end
```

3 An Evaluator for XCom

As described in the introducion we are interested in defining XCom operational semantics. We will do this in a number of different ways in the rest of this note. The first, and possibly most straightforward, approach is to define an *interpreter* for XCom in the XOCL language. This involves writing an `eval` operation for each of the XCom syntax classes. The `eval` operation must be parameterized with respect to any context information that is required to perform the evaluation. An XCom program `p` is then evaluated in a context `e` by: `p.eval(e)`.

3.1 Evaluating Expressions

Expression evaluation is defined by adding `eval` operations to each class in `Expressions` as follows:

```
context Exp
  @AbstractOp eval(env:Env):Value
  end
```

Evaluation of a constant produces the appropriate semantic domain value:

```
context Const
  @Operation eval(env)
    @TypeCase(value)
      Boolean do Bool(value) end
      Integer do Int(value) end
    end
  end
```

Evaluation of a variable involves looking up the current value. The value is found in the current context of evaluation: this must contain associations between variable names and their values. This is the only thing required of the XCom evaluation context and therefore we represent the context as an *environment* of variable bindings:

```
context Var
  @Operation eval(env)
    env.lookup(name)
  end
```

Evaluation of a binary expression involves evaluation of the sub-expressions and then selecting an operation based on the operation name. The following shows how XCom semantics is completely based on XOCl semantics since + in XCom is performed by + in XOCL.

```
context BinExp
  @Operation eval(env)
    @Case op of
      "and" do left.eval(env).binAnd(right.eval(env)) end
      "or"  do left.eval(env).binOr(right.eval(env)) end
      "+"   do left.eval(env).binAdd(right.eval(env)) end
    end
  end
```

Creation of new records is performed by evaluaing a `new` expression. The interpreter has dynamic types so the type to instantiate is found by looking up the type name in the current environment:

```
context New
  @Operation eval(env)
    env.lookup(type).new()
  end
```

Field reference is defined as follows:

```
context FieldRef
  @Operation eval(env)
    value.eval(env).lookup(name)
  end
```

3.2 Evaluating Statements

XCom statements are performed in order to introduce new names, control flow or to update a record field. Statements are defined to evaluate in a context and must observe the rules of scope that require variables are local to the block that introduces them. The context of execution is an environment; evaluation of a statement may update the supplied environment, so statement evaluation returns an environment:

```
context Statement
  @AbstractOp eval(env):Env
  end
```

A value declaration evaluates the expression part and then extends the supplied environment with a new binding:

```
context ValueDeclaration
  @Operation eval(env)
    env.bind(name,value.eval(env))
  end
```

A type declaration extends the supplied environment with a new type:

```
context TypeDeclaration
  @Operation eval(env)
    env.bind(name,Type(names))
  end
```

A block must preserve the supplied environment when its evaluation is complete. Each statement in the block is performed in turn and may update the current environment:

```
context Block
  @Operation eval(originalEnv)
    let env = originalEnv
    in @For statement in statements do
         env := statement.eval(env)
       end
    end;
    originalEnv
    end
```

A `while` statement continually performs the body while the test expression returns `true`. A while body is equivalent to a block; so any updates to the supplied environment that are performed by the while body are discarded on exit:

```
context While
  @Operation eval(originalEnv)
    let env = orginalEnv
    in @While test.eval(env).value do
         env := body.eval(env)
       end;
       originalEnv
    end
  end
```

An `if` statement conditionally performs one of its sub-statements:

```
context If
  @Operation eval(env)
    if test.eval(env).value
    then thenPart.eval(env)
    else elsePart.eval(env)
    end
  end
```

4 A Translator for XCom with Run-Time Types

The previous section defines an interpreter for XCom. This is an appealing way to define the operational semantics of a language because the rules of evaluation work directly on the abstract syntax structures. However the resulting interpreter can often be very inefficient. Furthermore, an interpreter can lead to an *evaluation phase distinction*. Suppose that XCom is to be embedded in XOCL. XOCL has its own interpretive mechanism (the XMF VM); at the boundary between XOCL and XCom the XOCL interpretive mechanism must hand over to the XCom interpreter – the XCom code that is performed is a data structure, a completely alien format to the VM. This phase distinction can lead to problems when using standard tools, such as save and load mechanisms, with respect to the new language. For example a mechanism that can save XOCL code to disk cannot be used to save XCom code to disk (it can, however, be used to save the XCom interpreter to disk).

An alternative strategy is to translate the source code of XCom to a language for which we have an efficient implementation. No new interpretive mechanism is required and no phase distinction arises. Translation provides the opportunity for static analysis (since translation is performed prior to executing the program). As we mentioned earlier, static analysis can translate out any type information from XCom programs; the resulting program does not require run-time types. Since static analysis requires a little more work, this section describes a simple translation from XCom to XOCL that results in run-time types; the subsequent section shows how this can be extended to analyse types statically and remove them from the semantic domain.

4.1 Translating Expressions

Translation is defined by adding a new operation `desugar1` to each sbatract syntax class. There is no static analysis, so the operation does not require any arguments. The result of the operation is a value of type `Performable` which is the type of elements that can be executed by the XMF execution engine.

```
context Exp
  @AbstractOp desugar1():Performable
  end
```

An XCom constant is translated to an XOCL constant:

```
context Const
  @Operation desugar1():Performable
    @TypeCase(value)
      Boolean do BoolExp(value) end
      Integer do IntExp(value) end
    end
  end
```

An XCom binary expression is translated to an XOCL binary expression. Note that the sub-expressions are also translated:

```
context BinExp
  @Operation desugar1():Performable
    @Case op of
      "and" do [| <left.desugar1()> and <right.desugar1()> |] end
      "or"  do [| <left.desugar1()> and <right.desugar1()> |] end
      "+"   do [| <left.desugar1()> + <right.desugar1()> |] end
    end
  end
```

An XCom `new` expression involves a type name. Types will be bound to the appropriate variable name in the resulting XOCL program; so the result of translation is just a message `new` sent to the value of the variable whose name is the type name:

```
context New
  @Operation desugar1():Performable
    [| <OCL::Var(type)>.new() |]
  end
```

XCom variables are translated to XOCL variables:

```
context Var
  @Operation desugar1():Performable
    OCL::Var(name)
  end
```

XCom field references are translated to the appropriate call on a record:

```
context FieldRef
  @Operation desugar1():Performable
    [| <value.desugar1()>.ref(<StrExp(name)>) |]
  end
```

4.2 Translating Statements

An XCom statement can involve local blocks. The equivalent XOCL expression that provides local definitions is `let`. A `let` expression consists of a name, a value expression and a body expression. Thus, in order to translate an XCom declaration to an XOCL `let` we need to be passed the body of the `let`. This leads to a translational style for XCom commands called *continuation passing* where each `desugar1` operation is supplied with the XOCL command that will be performed next:

```
context Statement
  @AbstractOp desugar1(next:Performable):Performable
  end
```

A type declaration is translated to a local definition for the type name. Note that the expression `names.lift()` translates the sequence of names to an expression that, when performed, produces the same sequence of names: `list` is a means of performing evaluation in reverse:

```
context TypeDeclaration
  @Operation desugar1(next:Performable):Performable
    [| let <name> = Type(<names.lift()>)
       in <next>
       end
    |]
  end
```

A value declaration is translated to a local decinition:

```
context ValueDeclaration
  @Operation desugar1(next:Performable):Performable
    [| let <name> = <value.desugar1()>
       in <next>
       end
    |]
  end
```

A block requires each sub-statement to be translated in turn. Continuation passing allows us to chain together the sequence of statements and nest the local definitions appropriately. The following auxiliary operation is used to implement block-translation:

```
context Statements
  @Operation desugar1(statements,next:Performable):Performable
    @Case statements of
      Seq{} do
        next
      end
      Seq{statement | statements} do
        statement.desugar1(Statements::desugar1(statements,next))
      end
    end
  end
```

Translation of a block requires that the XOCL local definitions are kept local. Therefore, the sub-statements are translated by chaining them together and with a final continuation of `null`. Placing the result in sequence with `next` ensures that any definitions are local to the block.

```
context Block
  @Operation desugar1(next:Performable):Performable
    [| <Statements::desugar1(statements,[| null |])> ;
```

```
      <next>
    |]
  end
```

A `while` statement is translated to the equivalent expression in XOCL:

```
context While
  @Operation desugar1(next:Performable):Performable
    [| @While <test.desugar1()>.value do
         <body.desugar1([|null|])>
       end;
       <next>
    |]
  end
```

An `if` statement is translated to an equivalent expression in XOCL:

```
context If
  @Operation desugar1(next:Performable):Performable
    [| if <test.desugar1()>.value
       then <thenPart.desugar1(next)>
       else <elsePart.desugar1(next)>
       end
    |]
  end
```

5 A Translator for XCom without Run-Time Types

It is usual for languages to have a static (or *compile time*) phase and a dynamic (or *run time*) phase. Many operational features of the language can be performed statically. This includes type analysis: checking that types are defined before they are used and allocating appropriate structures when instances of types are created. This section shows how the translator for XCom to XOCL from the previous section can be modified so that type analysis is performed and so that types do not occur at run-time.

5.1 Translating Expressions

Since types will no longer occur at run-time we will simplify the semantic domain slightly and represent records as *a-lists*. An a-list is a sequence of pairs, the first element of each pair is a ket and the second element is a value. In this case a record is an a-list where the keys are field name strings. XOCL provides operations defined on sequences that are to be used as a-lists: `l->lookup(key)` and `l->set(key,value)`.

The context for static analysis is a type environment. Types now occur at translation time instead of run-time therefore that portion of the run-time

context that would contain associations between type names and types occurs during translation:

```
context Exp
  @AbstractOp desugar2(typeEnv:Env):Performable
  end
```

Translation of a constant is as for `desugar1`:

```
context Const
  @Operation desugar2(typeEnv:Env):Performable
    self.desugar1()
  end
```

Translation of binary expressions is as for `desugar1` except that all translation is performed by `desugar2`:

```
context BinExp
  @Operation desugar2(typeEnv:Env):Performable
    @Case op of
      "and" do [| <left.desugar2(typeEnv)> and
                  <right.desugar2(typeEnv)> |] end
      "or"  do [| <left.desugar2(typeEnv)> and
                  <right.desugar2(typeEnv)> |] end
      "+"   do [| <left.desugar2(typeEnv)> +
                  <right.desugar2(typeEnv)> |] end
    end
  end
```

Translation of a variable is as before:

```
context Var
  @Operation desugar2(typeEnv:Env):Performable
    self.desugar1()
  end
```

A `new` expression involves a reference to a type name. The types occur at translation time and therefore part of the evaluation of `new` can occur during translation. The type should occur in the supplied type environment; the type contains the sequence of field names. The result of translation is an XOCL expression that constructs an a-list based on the names of the fields in the type. The initial value for each field is `null`:

```
context New
  @Operation desugar2(typeEnv:Env):Performable
    if typeEnv.binds(type)
    then
      let type = typeEnv.lookup(type)
      in type.names->iterate(name exp = [| Seq{} |] |
```

```
          [| <exp>->bind(<StrExp(name)>,null) |])
      end
    else self.error("Unknown type " + type)
    end
  end
```

A field reference expression is translated to an a-list lookup expression:

```
context FieldRef
  @Operation desugar2(typeEnv:Env):Performable
    [| <value.desugar2(typeEnv)>->lookup(<StrExp(name)>) |]
  end
```

5.2 Translating Statements

A statement may contain a local type definition. We have already discussed continuation passing with respect to `desugar1` where the context for translation includes the next XOCL expression to perform. The `desugar2` operation cannot be supplied with the next XOCL expression because this will depend on whether or not the current statement extends the type environment. Therefore, in `desugar2` the continuation is an operation that is awaiting a type environment and produces the next XOCL expression:

```
context Statement
  @AbstractOp desugar2(typeExp:Env,next:Operation):Performable
  end
```

A type declaration binds the type at translation time and supplies the extended type environment to the continuation:

```
context TypeDeclaration
  @Operation desugar2(typeEnv:Env,next:Operation):Performable
    next(typeEnv.bind(name,Type(names)))
  end
```

A value declaration introduces a new local definition; the body is created by supplying the unchanged type environment to the continuation:

```
context ValueDeclaration
  @Operation desugar2(typeEnv:Env,next:Operation):Performable
    [| let <name> = <value.desugar2(typeEnv)>
       in <next(typeEnv)>
       end
    |]
  end
```

Translation of a block involves translation of a sequence of sub-statements. The following auxiliary operation ensures that the continuations are chained together correctly:

```
context Statements
  @Operation desugar2(statements,typeEnv,next):Performable
      @Case statements of
        Seq{} do
          next(typeEnv)
        end
        Seq{statement | statements} do
          statement.desugar2(
            typeEnv,
            @Operation(typeEnv)
              Statements::desugar2(statements,typeEnv,next)
            end)
        end
      end
    end
```

A block is translated to a sequence of statements where local definitions are implemented using nested `let` expressions in XOCL. The locality of the definitions is maintained by sequencing the block statements and the continuation expression:

```
context Block
  @Operation desugar2(typeEnv:Env,next:Operation):Performable
    [| <Statements::desugar2(
         statements,
         typeEnv,
         @Operation(ignore)
           [| null |]
         end)>;
       <next(typeEnv)>
    |]
  end
```

A `while` statement is translated so that the XOCL expression is in sequence with the expression produced by the contintuation:

```
context While
  @Operation desugar2(typeEnv:Env,next:Operation):Performable
    @While <test.desugar2(typeEnv)>.value do
      <body.desugar2(typeEnv,@Operation(typeEnv) [| null |] end)>
    end;
    <next(typeEnv)>
  end
```

The `if` statement is translated to an equivalent XOCL expression:

```
context If
  @Operation desugar2(typeEnv:Env,next:Operation):Performable
```

```
    if <test.desugar2(typeEnv)>.value
    then <thenPart.desugar2(typeEnv,next)>
    else <elsePart.desugar2(typeEnv,next)>
  end
```

6 Compiling XCom

The previous section shows how to perform static type anslysis while translating XCom to XOCL. XOCL is then translated to XMF VM instructions by the XOCL compiler (another translation process). The result is that XCom cannot to anything that XOCL cannot do. Whilst this is not a serious restriction, there may be times where a new language wishes to translate directly to the XMF VM without going through an existing XMF language. This may be in order to produce highly efficient code, or because the language has some unusual control constructs that XOCL does not support. This section shows how XCom can be translated directly to XMF VM instructions.

6.1 Compiling Expressions

```
context Exp
  @AbstractOp compile(typeEnv:Env,valueEnv:Seq(String)):Seq(Instr)
  end

context Const
  @Operation compile(typeEnv,valueEnv)
    @TypeCase(value)
      Boolean do
        if value
        then Seq{PushTrue()}
        else Seq{PushFalse()}
        end
      end
      Integer do
        Seq{PushInteger(value)}
      end
    end
  end

context Var
  @Operation compile(typeEnv,valueEnv)
    let index = valueEnv->indexOf(name)
    in if index < 0
       then self.error("Unbound variable " + name)
       else Seq{LocalRef(index)}
       end
    end
```

```
  end

context BinExp
  @Operation compile(typeEnv,valueEnv):Seq(Instr)
    left.compile(typeEnv,valueEnv) +
    right.compile(typeEnv,valueEnv) +
    @Case op of
      "and" do Seq{And()} end
      "or"  do Seq{Or()} end
      "+"   do Seq{Add()} end
    end
  end

context New
  @Operation compile(typeEnv,valueEnv):Seq(Instr)
    self.desugar2(typeEnv).compile()
  end

context FieldRef
  @Operation compile(typeEnv,valueEnv):Seq(Instr)
    Seq{StartCall(),
        PushStr(name)}
    value.compile(typeExp,valueExp) +
    Seq{Send("lookup",1)}
  end
```

6.2 Compiling Statements

```
context Statement
  @AbstractOp compile(typeEnv:Env,varEnv:Seq(String),next:Operation):Seq(Instr)
  end

context TypeDeclaration
  @Operation compile(typeEnv,varEnv,next)
    next(typeEnv.bind(name,Type(names)),varEnv)
  end

context ValueDeclaration
  @Operation compile(typeEnv,varEnv,next)
    value.compile(typeEnv,varEnv) +
    Seq{SetLocal(name,varEnv->size),
        Pop()} +
    next(typeEnv,varEnv + Seq{name})
  end

context Statements
  @Operation compile(statements,typeEnv,varEnv,next)
```

```
    @Case statements of
      Seq{} do
        next(typeEnv,varEnv)
      end
      Seq{statement | statements} do
        statement.compile(
          typeEnv,
          varEnv,
          @Operation(typeEnv,varEnv)
            Statements::compile(statements,typeEnv,varEnv,next)
          end)
      end
    end
  end

context Block
  @Operation compile(typeEnv,varEnv,next)
    Statements::compile(
      statements,
      typeEnv,
      varEnv,
      @Operation(localTypeEnv,localVarEnv)
        next(typeEnv,varEnv)
        end)
  end

context While
  @Operation compile(typeEnv,varEnv,next)
     Seq{Noop("START")} +
     test.compile(typeEnv,varEnv) +
     Seq{SkipFalse("END")} +
     body.compile(typeEnv,varEnv,
       @Operation(typeEnv,varEnv)
         Seq{}
       end) +
     Seq{Skip("START")} +
     Seq{Noop("END")} +
     next(typeEnv,varEnv)
  end

context If
  @Operation compile(typeEnv,varEnv,next)
     test.compile(typeEnv,varEnv) +
     Seq{SkipFalse("ELSE")} +
     thenPart.compile(typeEnv,varEnv,
       @Operation(typeEnv,varEnv)
         Seq{Skip("END")}
```

```
      end) +
    Seq{Noop("ELSE")} +
    elsePart.compile(typeEnv,varEnv,
      @Operation(typeEnv,varEnv)
        Seq{Skip("END")}
      end) +
    Seq{Noop("END")} +
    next(typeEnv,varEnv)
end
```

7 Conclusions

This note has shown how XMF can be used to define the operational semantics of languages. We have shown how to implement an interpreter for a simple language and how to translate the language to existing XMF languages. We have discussed a number of different issues relating to language translation, in particular how much work is performed statically and how much is left to run-time.