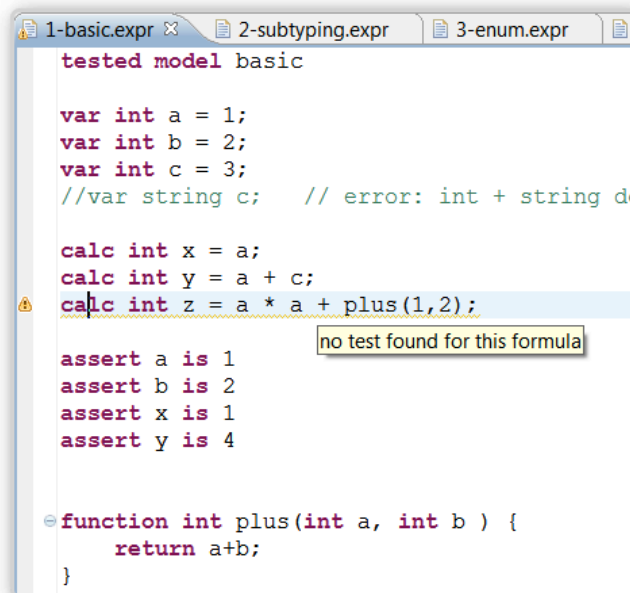# Xtext/Interpreter

Markus Völter
(voelter@acm.org)

Introduction

This document explains how to write your own simple interpreter for Xtext-based languages. A small supporting framework is available and will be used in this documentation. The language we use as an example in this documentation is an extended version of the example language introduced in the Xtext typesystem documentation, so please take a look.
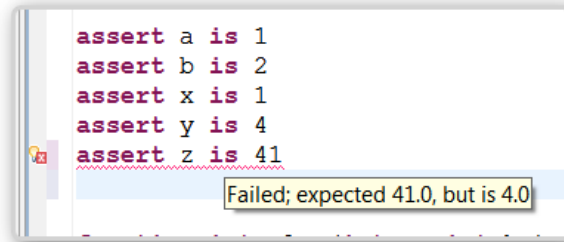
What is an interpreter?

An interpreter is a program that traverses a model and performs actions as the model is traversed. These actions usually constitute an execution of the program scrapped by the model. Take a look at the following example. The warning shown in the picture is a simple constraint. It can be evaluated by simply testing the structure of the model. If there is no assert statement where the actual value planes to the calculation at hand, then the warning is executed.

```
1-basic.expr ⊠    2-subtyping.expr    3-enum.expr
    tested model basic

    var int a = 1;
    var int b = 2;
    var int c = 3;
    //var string c;    // error: int + string do

    calc int x = a;
    calc int y = a + c;
⚠  calc int z = a * a + plus(1,2);
                    no test found for this formula
    assert a is 1
    assert b is 2
    assert x is 1
    assert y is 4


  function int plus(int a, int b ) {
        return a+b;
    }
```

Now let's take a look at the example below. To show this error, the program shown above actually has to be executed, the calculations expressed by the code have to be performed. Of course, you can generate code (for example the unit test) and run it. However, in the example below

we have integrated an interpreter into the validation framework for the language to perform the calculation and report the error.

```
assert a is 1
assert b is 2
assert x is 1
assert y is 4
assert z is 41
           Failed; expected 41.0, but is 4.0
```

In the rest of the document, we used this example to explain how to build an interpreter. Of course, the interpreter itself varies based on the language you want to build the interpreter for. I recently built an interpreter for a state machine, where asynchronous, event-based behavior has to be simulated. Many languages, however, have notions of expressions and statements. The reusable framework addresses these specifically.

Implementing the interpreter

The dispatch problem

If you design your language right the structure of expressions is recursive. We have already exploited this characteristic in the way the type system is built. We also make use of this fact in the interpreter. However, in case of the interpreter we cannot just use EMF reflection to work our way through the model. We actually have to write Java code. So we basically have to recursively step through the expression tree and evaluate expressions. This entails a lot of *instanceof* checks, or some kind of polymorphic dispatch (e.g. using the extract polymorphic dispatcher). For the interpreter framework we have decided on another approach, though: we generate a dispatcher that calls methods that are specific to evaluating the respective language concept. The following piece of code shows what this results in. For every expression concept you get an eval method you can implement.

```java
@Override
protected Object evalNumberLiteral(NumberLiteral expr,
            LogEntry log) {
    return new Double( expr.getValue() );
}

@Override
protected Object evalStringLiteral(StringLiteral expr,
            LogEntry log) {
    return expr.getValue();
}

@Override
protected Object evalMulti(Multi expr,
            LogEntry log) {
    return doubleTimesDouble(expr.getLeft(),
            expr.getRight(), log);
```

```
}

@Override
protected Object evalPlus(Plus expr, LogEntry log) {
        return doublePlusDouble(expr.getLeft(),
             expr.getRight(), log);
}
```

These methods are called through the generated dispatcher. You can get this dispatcher generated automatically by adding a new fragment to the MWE2 file that builds your language:

```
// interpreter
fragment = de.itemis.interpreter.generator.InterpreterGenerator {
    statementRootClassName = "Element"
    expressionRootClassName = "Expr"
}
```

This fragment can be put anywhere in the workflow, as long as it is after the fragment that generates the Ecorse file from the grammar.

The fragment takes two arguments: the base class from which all expressions derive (i.e. language concepts that evaluate to a value) and the base class for all the statements (i.e. language concepts that may do something, but don't have a value). You have to put the names of your language concepts (metaclasses) to make this work.

As a consequence, you will get two new classes generated into the *src-gen* directory of your language:
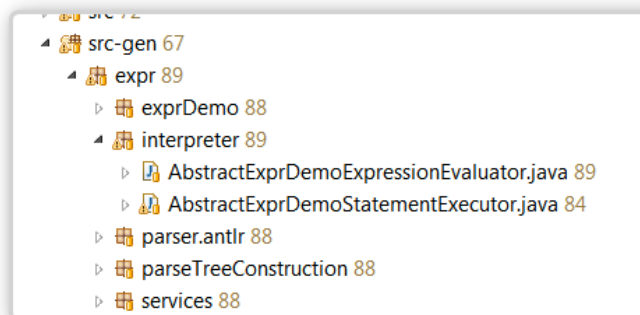
Setting up the infrastructure

Let us now implement the interpreter itself. The following class ties it all together:

```
public class ExprModelInterpreter {

 public MessageList runModel( Model m, ITypesystem ts )
             throws InterpreterException {
   ExecutionContext ctx = new ExecutionContext(ts);
   LogEntry log = LogEntry.root("running model "+m.getName());
   LogEntry.setMostRecentRoot(log);
   new Evaluator(ctx);
```

```
    new StatementExecutor(ctx).execute(m.getElements(), log);
    return ctx.messages;
  }

}
```

We first create an execution context. It contains among other things symbol table. We then create a *LogEntry*, which is used for capturing a trace of the overall execution for debugging purposes. We then register the log entry we just created globally for use in the trace view, which we will talk about later. We then create an evaluator and a statement executor. These two classes are manually written, and subclass the two generated classes shown above. These two classes contain the actual implementation.

Notice how the *runModel* method returns the set of messages that contain warnings and errors accumulated during the execution. These can then be shown as error markers on the elements that failed. The interpreter is called from a validation method:

```
@Check()
public void runAssertStatements( Model m ) {
  if ( !m.isIsTested() ) return;
  try {
    MessageList errors =
          new ExprModelInterpreter().runModel(m, ts);
    for (MessageList.MessageItem o: errors.getMessages()) {
      error( o.message, o.element, -1, INTERPRETERFAILED );
    }
  } catch (InterpreterException e) {
    if ( e.getFailedObject() != null ) {
      error( e.getMessage(), e.getFailedObject(),
          -1, INTERPRETERFAILED );
    } else {
      error( e.getMessage(), m, -1, INTERPRETERFAILED );
      e.printStackTrace();
    }
  }
}
```

Let us now take a look at the actual implementation code of the interpreter. Let us start wth the statements.

Executing Statements

The class *StatementExecutor* extends *AbstractExprDemoStatementExecutor*, which is the generated base class for used for executing statements. We implement the methods for the various subtypes of *Element* (the name oft he top level Statement-like class is *Element*). Let u stake a look at the various methods.

A variable declaration may have an init expression that defines the value of the variable. If an init expression is given, we

- first evaluate the expression. The *evalCheckNullLog* method evaluates the expression, throws an *UnexpectedNullException* if the value is null and creates a log entry (for debugging purposes, see below).

- then we put a new symbol-value pair into the symbol table. The symbol is the variable, the value is the result of the evaluation. So in some sense, we "assign" the value of the evaluated expression to the variable.

```java
@Override
protected void executeVarDecl(VarDecl s, LogEntry log) {
  Expr init = ((VarDecl) s).getInit();
  if ( init != null) {
    ctx.environment.put(s, evalCheckNullLog( init, log ));
  }
}
```

For a formula, we basically do the same thing (a variable with an init expression is kind of the same thing as a formula anyway). So we don't show the code here.

Next we take a look at the assert statement. It serves to compare two values and report an error if they are not the same.

```java
protected void executeAssert(Assert s, LogEntry log) {
   Object expected =
      evalCheckNullLog( s.getExpected(), log );
   Object actual =
      evalCheckNullLog( s.getActual(), log );
   if ( !expected.equals(actual) ) {
     ctx.messages.addError(s,
           "Failed; expected "+expected+
           ", but is "+actual );
   }
}
```

We first evaluate the two arguments (expected and actual). If they are not equal, we output an error message. The messages object in the context has a method to output errors and warnings. As we have seen above, these will be rendered as error/warning markers in the editor.

The last kind of statement is the return statement. This can only be understood once we discussed function calls. Since functions have return values, they are evaluated, not executed, and we explain it below. Functions themselves are not executed as a statement either. They are only executed once they are "called" from the evaluator.

Evaluating Expressions

Let us start with the simple stuff. The value of a number literal is the numeric representation of the literal text, i.e.:

```java
protected Object evalNumberLiteral(NumberLiteral expr,
                LogEntry log) {
      return new Double( expr.getValue() );
}
```

For a string literal, the code is basically the same.

Multiplication and addition are similar to each other. They recursively evaluate their left and right arguments, multiply/add them, and then return a new number. What makes it a bit non-trivial in real-world languages is that numeric values can have different types: *int* and *double*,

for example. And the addition function has to downcast to these types correspondingly. Since this is annoying, and basically the same in every interpreter, we have provided utility methods for this (take a look at the class *AbstractExpressionEvaluator* in the framework to see all the utility methods; these will become more over time). In the end, we can implement the methods like this:

```
@Override
protected Object evalMulti(Multi expr, LogEntry log) {
        return doubleTimesDouble(expr.getLeft(),
                        expr.getRight(), log);
}


@Override
protected Object evalPlus(Plus expr, LogEntry log) {
        return doublePlusDouble(expr.getLeft(),
                        expr.getRight(), log);
}
```

Evaluating a symbol reference is more interesting, since, as a consequence of how we had to design the grammar, it can reference various different things. A symbol reference also acts as a function call (it may then have arguments; checked by the validator).

So here are the first couple of lines of the method:

```
protected Object evalSymbolRef(SymbolRef expr, LogEntry log) {
  Symbol symbol = expr.getSymbol();
  if ( symbol instanceof VarDecl ) {
     return log( symbol,
         ctx.environment.getCheckNull(symbol), log);
  }
```

If the referenced symbol is a variable declaration, then we go to the symbol table and try to lookup the value. We wrap this in a call to *log* to update the execution log. If we don't find a value, the *getCheckNull* throws an appropriate exception and the interpreter stops. We could check explicitly, and output an nicer message (using *ctx.messages.addError(..)*).

The code for a reference to a formula is similar. We also use the same code for a reference to a *Parameter* from inside a function. This raises the question of how function parameters get their values… So let's take a look at the code for the function call, i.e. a symbol reference that references a function.

```
if ( symbol instanceof FunctionDeclaration ) {
  FunctionDeclaration fd = (FunctionDeclaration) symbol;
  return callAndReturnWithPositionalArgs("calling "+
          fd.getName(), fd.getParams(), expr.getActuals(),
          fd.getElements(), RETURN_SYMBOL, log);
}
```

 We simply call a utility function *callAndReturnWithPositionalArgs*. Let's undertstand what it does:

- First it pushes the symbol table. The symbol table is a stack of tables, and a the execution of a function has it's own copy of the symbol table. After the execution of the function, the symbol table stack is popped, restoring the same symbolic environment as before the function call.

- On the new symbol table, the utility function adds associations between each formal parameter (*fd.getParams()*) and the value passed in as an actual argument (*expr.getActuals()*).

- The utility function then executes the statements in the body of the function (*fd.getElements()*). As we have seen above, we look up the value via the symbol reference to parameters in the "pushed" symbol table.

- Finally, when we execute the return statement, we put the value of the return expression into a symbol table (under a special name, *RETURN_SYMBOL*). After executing the function body, the utility function grabs that value from the symbol table and returns it as the result of the call – which we then return further, as the value of the function call.

Please take a look at the implementation of the *callAndReturnWithPositionalArgs* method:

```
protected Object callAndReturnWithPositionalArgs(String name,
        EList<? extends EObject> formals,
        EList<? extends EObject> actuals,
        EList<? extends EObject> bodyStatements,
        Object returnSymbol, LogEntry log) {

  ctx.environment.push(name);
  for( int i=0; i<actuals.size(); i++ ) {
      EObject actual = actuals.get(i);
      EObject formal = formals.get(i);
      ctx.environment.put(formal,
              evalCheckNullLog(actual, log));
  }
  ctx.getExecutor().execute( bodyStatements, log );
  Object res = ctx.environment.get(returnSymbol);
  ctx.environment.pop();
  return res;
}
```

We can now take a look at – and make sense of – the code for executing the return statement:

```
protected void executeReturn(Return s, LogEntry log) {
  ctx.environment.put(Evaluator.RETURN_SYMBOL,
        evalCheckNullLog(s.getExpr(), log));
}
```

Logging the execution

Debugging the execution of the program can either be done on the level od the DSL/model (currently not possible) or it can be done on the level of the interpreter source (possible, but annoying). Another alternative, especially for small programs, is logging the execution trace of the program and making it accessible to the users.

In our example here we have integrated the log view as a quick fix action. So if an assertion fails, you can press Ctrl-1 and select Show Log. This is how the result look (cuurently):

```
assert a is 1
assert b is 2
assert x is 1
assert y is 4
assert z is 41
```

| Problems | @ Javadoc | Declaration | Interpreter Execution Log ⊠ |

Time	Kind	Element	Message	Exce
▶ 13:42:05.480	eval	SymbolRef	evaluating SymbolRef result: 2.0	
▼ 13:42:05.480	debug	Assert	executing Assert	
13:42:05.480	eval	NumberLiteral	evaluating NumberLiteral result: 1.0	
▶ 13:42:05.480	eval	SymbolRef	evaluating SymbolRef result: 1.0	
▼ 13:42:05.480	debug	Assert	executing Assert	
13:42:05.480	eval	NumberLiteral	evaluating NumberLiteral result: 4.0	
▼ 13:42:05.480	eval	SymbolRef	evaluating SymbolRef result: 4.0	
▼ 13:42:05.480	eval	Plus	evaluating Plus result: 4.0	
▶ 13:42:05.480	eval	SymbolRef	evaluating SymbolRef result: 1.0	
▶ 13:42:05.480	eval	SymbolRef	evaluating SymbolRef result: 3.0	

You can inspect all the log objects created during execution. A number of objects are created by the framework, others you have to create by hand. Each log object carries the object to which the log message relates. You can double click on the respective line in the table to select the corresponding object/line in the program.