Extending JML
By New Data Types

Peter. H. Schmitt

May 8, 2016

These notes explain the changes to the existing code of the KeY system that are necessary to add a new data type to JML. If the new data type is only used in JML specifications then only the changes described in Section 1 need to be done. If you want to use ghost variables or ghost fields, in particular assignments to ghost variables and fields of the new adt then in addition the changes described in Section 2 need to be done.

These notes do not aim to provide an understanding of the existing code they just tell the user what to do by mimicking what is already there. We take the data type *Seq* of finite sequences as a model.

In the following we use `Basepath` as an abbreviation for the path

`gitKeY/key/key/key.core/src/de/uka/ilkd/key/`

1 Extension Of The JML Parser

At the moment the following instructions do not cover the addition of new variable binder symbols.

Change file `KeYJMLLexer.g` in `Basepath/speclang/jml/translation`
Here are some existing entries:

```
SEQ       : '\\seq';  //KeY extension, not official JML
SEQCONCAT : '\\seq_concat'; //KeY extension, not official JML
SEQEMPTY  : '\\seq_empty';  //KeY extension, not official JML
```

that you change to fit the purpose. It is important to note that it is here that you establish the syntax that is to be used in the code for the data type name and its operations. In the present case that is `\seq` for the data type of finite sequences and `\seq_concat` for the concatenation operation and

`\seq_empty` for the empty sequence. Note, the addition escape character `\`. At the same time you also establish a name for the corresponding nonterminal grammar symbol.

Personal note: I am suprised that `seqLen` does not occur here.

Change file `KeYJMLParser.g` `Basepath/speclang/jml/translation`

Change 1 Look for the existing grammar rule for sequences which looks roughly like this:

```
sequence returns [SLExpression ret = null] throws SLTranslationException
@init {
    ImmutableList<Term> tlist = null;
    KeYJavaType typ;
    Term t, t2;
    Token tk = null;
    Pair<KeYJavaType,ImmutableList<LogicVariable>> declVars = null;
}
@after { ret = result; }
:
 SEQEMPTY
   {
     result=new SLExpression(tb.seqEmpty());
    }
 |
 .
 .
 .
 |  (tk2=SEQCONCAT{tk=tk2;} | tk3=SEQGET{tk=tk3;} | tk4=INDEXOF{tk=tk4;})
      LPAREN e1=expression COMMA e2=expression RPAREN
  {
    result=translator.translate(tk.getText(),
                                SLExpression.class,
                                services,e1,e2);
  }
;
```

Add a new rule replacing the name `sequences` and adapt what follows after the colon `:`.

Change 2 In the `jmlprimary` rule take the entry

```
| (SEQEMPTY
   |
   .
   .
   .
   | SEQCONCAT
   | SEQGET
   | INDEXOF)
        => result = sequence
```

as a model. Of course you replace `sequence` in this example by the rule name you chose in the first change.
Do not forget to add an equivalent for your new data type for the line

```
| (LPAREN (SEQDEF | SEQ) quantifiedvardecls SEMI)
```

This is neacessary fo parsing quantifier variables of the new data type.

Change 3 In the `builtintype` rule add a new entry for your data type mimicking

```
 |   SEQ
     {
        type = javaInfo.getKeYJavaType(PrimitiveType.JAVA_SEQ);
      }
```

Change file `TermBuilder.java` in `Basepath/logic`
Look for the section starting with the comment lines

```
//-------------------------
//sequence operators
//-------------------------
```

add a suitably adapted section for the new data type. You will notice that you have to use the method names introduced in the public interface section in `OrdLDT`.

Create files For each nonterminal `NT` for an operator as declared in `KeYJMLLexer.g` create a class file `NameNT.java` in the directory `Basepath/java/expression/operator/adt` following the examples
Take `SeqConcat.java` or `SeqLength.java` as an example.
It is a good idea to pick a name `NameNT` that somehwo resembles `NT`.

Constants are treated as literals, not as operators, and their corresponding files go into a different directory. See below.

Calls to the constructors of these new classes are passed on via `super` to the constructors of the class it extends, `Operator.java` or `BinaryOperator.java`. So here there is nothing else to do but renaming. You should also get the `getArity()` right.
I also set the result of the `getPrecedence()` method to 0. I am not sure if that is always correct.

Work needs to be done for the last two methods `visit(Visitor v)` and `prettyPrint(PrettyPrinter p)` as will be detailed in the next steps.

Change file `Visitor.java` in directory `Basepath/java/visitor`.

This class is an interface. All you have to do here is to add a line that adapts the examples you see, e.g.

```
void performActionOnSeqConcat(SeqConcat x);
```

Change file `CreatingASTVisitor.java` in `Basepath/java/visitor`.

The empty method specifications from `Visitor.java` are overwritten here. Just copy what you see, e.g. for `performActionOnSeqConcat` and do the appropriate renaming.

Change file `JavaASTVisitor.java` in directory `Basepath/java/visitor`.

This is an abstract class. You need to add a default method implementation. See `performActionOnSeqConcat(SeqConcat x)` for a model.

Always when you edit a file check the import clauses. You need to add an adaption of

```
import de.uka.ilkd.key.java.expression.operator.adt.SeqConcat;
```

in all three files you edit in the last three steps. You replace of course `SeqConcat` by the name of the file you created in step 2.

The files for literals go into the directory `Basepath/java/expression/literal` instead.

Change file `PrettyPrinter.java` in directory `Basepath/java`.

You need to add a method `printnewOp` for your operator `newOp` in the data type `newAdt`. tylke your leads e.g. from `printSeqConcat`. Of course you enter here the string you want to see printer for your operator.

Create files in directory `Basepath/java/expression/literals`
This parallels the creation of files like `SeqConcat.java` for the operation `SeqConcat`, but now for the literals declared in `KeYJMLLexer.g`. Note, that these files go into a different directory. The following changes also parallel those for operators except that for literals `JavaASTVisitor.java` is not affected.

Change file `Visitor.java` in directory `Basepath/java/visitor`.

Change file `JavaASTVisitor.java` in directory `Basepath/java/visitor`.

Change file `PrettyPrinter.java` in directory `Basepath/java`.

Create file in directory `Basepath/ldt`
Look at the existing file `SeqLDT.java` as a model for the new file to be created. You will note that for every literal and operation class created in `java.expression.literal` and `java.expression.operator.adt` a field of type `Function` is declared. To choose a name for this field best follow the pattern you find in `SeqLDT.java`. The link between the classes in `java.expression.operator.adt` and the fields are effected in method `getFunctionFor`. The link between the classes in `java.expression.literal` and the fields are effected in method `translateTerm`.

Also note that in the constructor `SeqLDT(TermServices services)` e.g. in the line

```
seqConcat    = addFunction(services, "seqConcat");
```

the string, here `"seqConcat"`, must match the declarations in the .key file, here in `seq.key`.

Make sure to include all the files created in the previous two create steps in the import statements.

This is a lot of work.

Change file `LDT.java` in directory `Basepath/ldt/`
In the body of the method `getNewLDTInstances(Services s)` add aline for the new data type mimicking the existing lines, e.g.

```
ret.put(SeqLDT.NAME, new SeqLDT(s));
```

Change file `TypeConverter.java` in `Basepath/java`
Add a line for the new data type taking the following line for the sequence data type as a model:

```
 public SeqLDT getSeqLDT() {
return (SeqLDT) getLDT(SeqLDT.NAME);
    }
```

and, as always, do not forget the necessary import statement.

2 Extension Of The Recoder To KeY Translation

Change file `LDT.java` in directory `Basepath/ldt`

Add a line for the new data type mimicking e.g. the existing line for the Seq data type

```
  ret.put(SeqLDT.NAME, new SeqLDT(s));
```

Change file `ProofJavaParser.jj` in `Basepath/parser/proofjava/`

Change 1 In section `/* RESERVED WORDS AND LITERALS */` mimick the line

```
| < SEQ: "\\seq" >
```

The string `SEQ` for the nonterminal of the grammar and the name `\seq` for the new data type are the same as in the grammar file `KeYJMLLexer.g` in `Basepath/speclang/jml/translation`. Note, the additional escape character `\`. The notation you substitute for `\seq` will be used in the annotated Java code.

Change 2 Below `TypeReference PrimitiveType() :` add a line by adapting

```
 | "\\seq"
```

Change 3 Look for `ADTGetter() :` and add all getter symbols of the new data type Just mimick what you see there for operators starting with `\seq_`. Here is an example:

```
|
  "\\seq_concat" "(" expr=Expression() "," result=Expression()")"
  {
      result = new SeqConcat(expr, result)
      setPrefixInfo(result);
   }
|
```

enclosed in the disjunctive separator symbol `|` of the grammar syntax. The name of the operator in place of `\seq_concat` you take from the grammar file `KeYJMLLexer.g` in `Basepath/speclang/jml/translation`. The result in the above code fragment refers to a constructor, `SeqConcat(expr,result)`, for the class `SeqConcat`. It is your job to add these classes, one for each operator, getter, constructor or literal. See below.

Change 4 Look for `ADTConstructor() :` and add all constructor symbols for the new data type. Here is a guiding example from the seq data type:

```
|
  "\\seq_concat" "(" expr = Expression() "," result = Expression() ")"
  {
      result = new SeqConcat(expr, result)
      setPrefixInfo(result);
  }
|
```

Change 5 Constants are not considered as constructors. You have to add them as literals. Here is a simple example that suffices for the simplest case of just adding one literal.

```
EmptySeqLiteral EmptySeqLiteral() :
{
    EmptySeqLiteral result;
}
{
  "\\seq_empty"
```

```
  {
      result = EmptySeqLiteral.INSTANCE;
      setPrefixInfo(result);
      return result;
  }
}
```

Things get complicated if you want families of literals as e.g. in `bigint`. I did not investigate this.

Change file `Recoder2KeYConverter.java` in directory `Basepath/java`

convert methods needs to be added. Again look for the line

`public SeqConcat convert(...adt.SeqConcat e)`

and do the appropriate changes.

Do not forget to add the necessary import statements.

Change file `PrimitiveType.java` in directory `Basepath/java/abstraction`

Here is the entry for the data type of finite sequence that you may take as a model:

```
public static final PrimitiveType JAVA_SEQ =
   new PrimitiveType("\\seq", EmptySeqLiteral.INSTANCE, SeqLDT.NAME);
```

The second argument refers to the default element for the new data type. See below.

Create files in the directory `Basepath/java/recoderext/adt/` for each operator and each literal introduced in `KeYJMLLexer.g` in directory `Basepath/speclang/jml/translation`.

Look at `EmptySeqLiteral.java`, `SeqLength.java`, and `SeqConcat.java` as examples for literals, unary and binary operators.

In these files you find a field `private static final long serialVersionUID` which you can safely set to 0.

Change file `RecoderModelTransformer.java` in `Basepath/java/recoderext`

In this file the default element of the new data tpye is handled. Mimic the line

```
} else if ("\\seq".equals(type.getName())) {
       return EmptySeqLiteral.INSTANCE;
```

and include the file that replaces `EmptySeqLiteral` in the import statements.

Change file `KeYCrossReferenceSourceInfo.java` in directory

`gitKeY/key/key/key.core/src/recoder/service`

First include all the new files you created in the directory `Basepath/java.recoderext.adt` in the import statements. Then extend the method `public Type getType(Expression expr)` appropriately. It seems that only constructor symbols need to be included.

3 Find the Error

To find the possible error when KeY fails to load I have added the following piece of code

```
System.out.println("reached reportError with message "+message);
System.out.println("throwable is "+ t);
java.io.StringWriter sw = new java.io.StringWriter();
java.io.PrintWriter pw = new java.io.PrintWriter(sw);
t.printStackTrace(pw);
System.out.println(sw.toString());
```

at the beginning of the method body for `reportError(String message, Throwable t)` in file `Recoder2KeY.java` in directory `Basepath/java`.
This prints the error trace of the thrown exception `t` and might give you a hint what could be the problem.