Katja: Generating Immutable Java Classes for Term and Position Types

Patrick Michel [1]

University of Kaiserslautern
Germany

1 Introduction

Todays most commonly used programming languages like Java or C++ lack proper support for immutable and recursive datatypes. As there are many applications which are based on complex recursive data structures and whose design would profit from immutability, it is desirable to introduce such datatypes to the Java programming language.

Katja is a small and light-weight specification language for order-sorted recursive term and position datatypes for Java. Specifications are concise, expressing what is really necessary and thereby avoiding the downsides of language integrating approaches. These are especially critical when it comes to verification of program properties which Katja facilitates by providing a concise formalisation of the generated types, as described in [2].

Though beeing a separate language, Katja does not lose the close relationship to the host language, as it is generating Java classes representing the order-sorted type structure. It is thereby possible to utilize the complete Java language with added immutable and recursive datatypes, whose interfacing is easy to learn yet powerful. The ability to generate classes allows the integration of powerful methods and commonly needed functionalities into the datatypes. Katja provides enhanced capabilities to express properties on occurrences of terms in their greater context by providing term-position types for the specification.

This tool demo gives an overview of Katja by providing an introductory example showing the usage as well as demonstrating it's capabilities.

2 Specifying with Katja

Consider the following task: You want to be able to process terms of a predicate calculus to check context conditions and normalize them with respect to

[1] eMail: p_michel@informatik.uni-kl.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

a base of operators. Let $F, \Rightarrow$ and $\forall$ be the base of the considered calculus and $\exists$ an additional operator which is only provided for convenience. The Katja specification to support formulas of the calculus looks like this:

```
package formula

Formula     ( Expression top )
Expression  = False      ( )
            | Implies    ( Expression left, Expression right )
            | Not        ( Expression expr )
            | Predicate  ( String name, Parameters vars )
            | Quantifier
Quantifier  = Forall     ( Variable vari, Expression expr )
            | Exists     ( Variable vari, Expression expr )
Parameters  * Variable
Variable    ( Integer index )
```

Katja reads the specification file, generates Java classes for all types and puts them together with some auxiliary classes in the specified package named `formula`. This package can be included in other Java programs, which is all one has to do to utilize all Katja features.

The Katja specification language consists only of three production types; the tuple, list and variant productions. These have the general form:

```
<tupletype>   ( <type1> <selector1>, ..., <typen> <selectorn> )
<listtype>    * <anytype>
<varianttype> = <type(def)1> | ... | <type(def)n>
```

Selectors for tuples are optional but can greatly increase readability and type-safety. If a tuple type is used in a variant it can be defined within the variant, as it is done in the example above. The definition of term types is pretty much the same as in functional languages like ML, except for the fact that variant productions take types as parameter not constructor names, what results in an oder-sorted type structure.

The term classes Katja generates are immutable, their interface is easy to learn, consisting primarily of typing and subterm methods. Base types which can be used in productions are the types in the `java.lang` package as well as any type which is imported into the specification. These types cannot be used in variant productions as the variant would be another supertype which cannot be added anymore.

To actually get a `Formula` in Java, one can for example define a scanner combined with a parser, to build up an abstract syntax tree of some input language, using the generated term constructors in the grammar rules. However it is sufficient for our small example to show the construction of the tautology $(\exists x_0 \forall x_1\ p(x_0, x_1)) \Rightarrow (\forall x_1 \exists x_0\ p(x_0, x_1))$ in Java syntax:

```
Formula formula = Formula(Implies(
  Exists(Variable(0), Forall(Variable(1),
    Predicate("p", Parameters(Variable(0), Variable(1))))),
  Forall(Variable(1), Exists(Variable(0),
    Predicate("p", Parameters(Variable(0), Variable(1)))))));
```

3 Support for Positions in Terms

Katja does not only generate term types from the specification file, but also generates position types representing occurrences of terms in the context of a root term. These types have the same name as the term type plus the "Pos" suffix and a type parameter denoting the type of the root term. The latter is omitted in the examples as it is not needed in this application.

The position interface is much more complex, since it consists of many navigation methods, e.g. `parent`, `rsib`, `lsib`, `preOrder`, `postOrder`, etc. To get a position one has to invoke the `pos` method on a term, which returns the position of the term in the context of itself. Invoking the `root` method on any position yields the root-term of the position structure.

Working with Java representations like the one of our example `Formula`, we can now check if all variables are bound by a quantifier. To do this we define a mapping from `Variable` occurrences to `Quantifier` occurrences:

```
public static QuantifierPos boundTo(VariablePos variablePos) {
  for(KatjaTermPos pos = variablePos;; pos = pos.parent()) {
    if(pos == null) return null; /* unbound Variable */
    if(pos.is(QuantifierPos.sort) &&
            ((QuantifierPos) pos).vari().termEquals(variablePos))
      return (QuantifierPos)pos; /* binding is found */
  }
}
```

Using positions of subterms instead of subterms themselves enables us to distinguish variables with the same name and navigate through their context. The `boundTo` method takes advantage of this feature by walking *upwards* in the term position structure, searching for an appropriate `Quantifier` position. The comparison of the variable and the quantifier variables has to be done on terms since their positions surely differ.

To check that all variables are bound we have to walk through the term and look at all variable bindings. Again this can be expressed very naturally without using recursion or stacks. It is possible to descend in the tree without losing the information of the upper context:

```
public static void printAllBindings(FormulaPos formulaPos) {
  for(KatjaTermPos pos = formulaPos; pos != null; pos = pos.preOrder()) {
    if(pos.is(VariablePos.sort) && !pos.parent().is(QuantifierPos.sort))
      System.out.println(pos+" is bound to "+boundTo((VariablePos) pos));
  }
}
```

This method simply checks the sort of positions when passing by and invokes the `boundTo` method whenever the *parent* of the variable is not a quantifier. When invoked on our example tautology we get the output:

```
Variable@Formula(0, 0, 1, 1, 1, 0) is bound to Exists@Formula(0, 0)
Variable@Formula(0, 0, 1, 1, 1, 1) is bound to Forall@Formula(0, 0, 1)
Variable@Formula(0, 1, 1, 1, 1, 0) is bound to Exists@Formula(0, 1, 1)
Variable@Formula(0, 1, 1, 1, 1, 1) is bound to Forall@Formula(0, 1)
```

The information we get is what we already expected: All occurrences of variables are bound by a quantifier occurrence. One should note that without the use of occurrences for the mapping to quantifiers the `boundTo` method would be useless, as there are equal quantifiers which get their individual meaning primarily through their position.

4 Transformations on Terms

It is in many cases desirable to have the ability to change parts of a term, e.g. to simplify an abstract syntax tree, eleminate redundancies, etc. To maintain immutability of terms one has to create a new term reflecting the change on each step. The elementary operation is the replacement of a subterm in a specific position with regard to a root term. Katja therefore provides the powerfull `replace` method on positions, which generates a new root term by replacing the term at the current position by a new one. It then returns the position of the new subterm in the same context as the original.

To normalize a `Formula` to the mentioned base of operators we need this function for our example calculus::

```
public static FormulaPos normalize(FormulaPos formulaPos) {
  KatjaTermPos last = null;
  for(KatjaTermPos pos = formulaPos.postOrderStart(); pos != null;
                  last = pos, pos = pos.postOrder()) {
    if(pos.is(ExistsPos.sort)) {
      Exists exists = ((ExistsPos) pos).term();
      pos = pos.replace(Implies(Forall(exists.vari(),
            Implies(exists.expr(), False())), False()));
    }
  }
  return ((FormulaPos) last);
}
```

We do this by walking through the tree in post-order, replacing all `Exist` positions as we get to them. The `postOrder` method can be applied though the position structure has been changed as it does not depend on any deprecated internal state.

5 Conclusion

Katja is a light-weight specification language which enables developers to work with the powerfull Java language but concentrate on the essential. Katja not only generates functional term types, but also allows the formulation of properties above positions which have a rich, yet easy and usefull interface.

At present state of the project Katja already facilitates attribution of term trees by identifying positions of term occurrences. Current and future developement focuses on the specification of attributes and functions which are defined within the Katja specification and evaluated by the Katja framework

at runtime. Such specifications will be as concise as sort definitions are now by providing a pattern matching facility similar to [1]

References

[1] Poetzsch-Heffter, A., *Prototyping realistic programming languages based on formal specifications*, Acta Informatica **34** (1997), pp. 737–772.

[2] Poetzsch-Heffter, A. and N. Rauch, *Application and formal specification of sorted term-position algebras*, in: J. L. Fiadeiro, editor, *17th International Workshop on Recent Trends in Algebraic Development Techniques, WADT 2004, Barcelona, Spain*, Lecture Notes in Computer Science (2004).