Automatic renovation of Java programs using ReRAGs — examples and ideas

Torbjörn Ekman and Görel Hedin

Department of Computer Science, Lund University, Sweden
`(torbjorn|gorel)@cs.lth.se`

Abstract. When new constructs are added to a language there is a need for renovating existing programs to make use of the new constructs. We discuss how this can be done using object-oriented ReRAGs (Rewritable Reference Attributed Grammars), and exemplify with the transition from Java 1.4 to Java 1.5.

1 Introduction

When a language evolves and new features are added, it is usually desirable to renovate existing programs to make use of the new features. Some new features could make old programs illegal, and make renovation necessary. Other new features may allow old idioms to be replaced by clearer code.

The evolution of the Java [GJSB00] language contains many such examples. In the evolution from Java 1.3 to Java 1.4, the addition of the assert construct included adding a new keyword to the language. Old programs that happened to use identifiers with the name "assert" were incompatible with Java 1.4, and needed to be changed. In the evolution to Java 1.5, a number of new language constructs are added, allowing many programs to be simplified. An example is the new for loop that allows many iterations to be expressed in a simpler way. Other Java 1.5 examples include static imports, automatic boxing and unboxing, and the use of generics. The evolution need not always concern the language as such. It could as well concern the evolution of standard frameworks, for example by adding new operations that should be used in favor over others that are deprecated.

How can automated support of such renovation be implemented? In simple cases a text replacement tool might do the job. In other cases, more sophisticated techniques are needed that analyze the program in order to identify exactly which parts of the code are relevant to replace, and what to replace them with. For example, the new Java 1.5 *for* construct works only for iterators over types implementing the *java.lang.Iterable* interface, and not over arbitrary types. Another problem may be that many different idioms may be used in the program, and a tool that identifies some standard predefined idioms may not be sufficient.

In this paper we outline how these kinds of problems can be handled using Rewritable Reference Attributed Grammars (ReRAGs), an object-oriented translation technique that we have recently developed [EH04]. In ReRAGs, a program is represented as an object-oriented abstract syntax tree (AST). Computations on the AST, e.g., to support name analysis, type checking, etc., are easy to express in ReRAGs, and can be used by

conditional rewrite rules that can transform the AST to a suitable new form. In our previous work, we have used ReRAGs to implement a Java compiler. We are now looking at how ReRAGs can additionally support program renovation.

ReRAGs have similarities to HAGs (higher-ordered attribute grammars) [VSK89] [Sar99] and to other grammar-based transformation systems such as TXL [CHHP91], ASF+SDF [vdBHdJ+01], and Stratego [V01], in its support of AST-based transformations. A major difference is, however, the object-oriented basis of ReRAGs where ASTs are represented as trees of objects with properties represented as methods and reference-valued fields. In contrast, TXL and HAGs have a basis in functional programming, and ASF+SDF and Stratego have an algebraic basis, viewing the ASTs as functional tree-structured values.

2 The Java 1.5 iterating for loop

The standard *for* construct in Java is extended in Java 1.5 with a second form specifically designed to iterate over collections and arrays. The following example, iterating over all elements in a collection, illustrates the extended functionality.

```
Collection c = ... ;

for(Object o : c) {
  System.out.print(o);
}
```

The new loop construct iterates over all elements in *c* and the current element is bound through the variable *o* of type *Object*. The construct is even more powerful in combination with generics where the type of the collection and current element can be parameterized and need thus not be the generic type Object. Generics are not further discussed in this paper. To be able to iterate over the expression *c* it must implement a new interface, *java.lang.Iterable*, that defines how to access the iterator. The standard class libraries are extended to implement this new interface where appropriate.

Two different straight forward implementations of the new loop construct using Java 1.4 constructs are shown below. The first idiom for iterating over a collection is actually used more than 60 times in the JDK1.4.2 standard libraries. The second, less common version, occurs more than 10 times.

```
Collection c = ... ;

for(Iterator i = c.iterator(); i.hasNext(); ) {
  System.out.print(i.next());
}

Collection c = ... ;

Iterator i = c.iterator();
while(i.hasNext()) {
  System.out.print(i.next());
}
```

Source code implemented using the above described idioms to iterate over collections can automatically be renovated to use the new loop constructs. It is, however, important that this automatic process is safe in that the program semantics are preserved. We will now describe how the first idiom can be detected and renovated to use the new *for* construct.

A for statement starts with an *InitPart*, Iterator i = c.iterator() in the example above, followed by a *Condition*, i.hasNext() above, and finally an *IncrPart*, empty in the example. There is also a *Body* that contains the code for each step in the iteration. The *InitPart* is evaluated before the iteration starts. The iteration repeatedly executes the *Body* as long as the *Condition* is true. After each step in the iteration the *IncrPart* is executed. We define the following pre-conditions to detect uses of the first idiom that can safely be transformed:

– *InitPart* declares and initializes a single reference *v* of type *java.util.Iterator*
– initialization invokes the *iterator()* method defined in the *java.lang.Iterable* interface
– *Condition* is an expression that invokes the *hasNext()* method on *v*
– *IncrPart* is empty
– There is a single access to the reference *v* in the statement body and that access is a method invocation of the *next()* method.

The new for statement starts with an *InitPart* followed by an expression, *Collection*, that defines the collection to iterate over and finally a statement, *Body*, that is repeated for each element. The transformation from the old statement to the new statement can then be divided into the following steps:

– The *InitPart* contains a variable declaration re-using the name *v* from the old version. That name can be used since the iterator will not be used any longer.
– The *Collection* is created from the initialization part of *v* in the old *InitPart* except that the invokation of *iterator()* is removed.
– The *Body* from the old statement is transformed by removing the next invocation from the reference *v*.

3 Expressing the change using ReRAGs

This section describes how the pre-conditions and transformation can be implemented using ReRAGs and our existing grammar for Java. The statements are modelled using the following abstract syntax tree (AST) definitions:

```
ast ForStmt extends Stmt ::= VariableDecl initPart, [Expr condition],
    Stmt incrPart*, Stmt body;
ast IterateStmt extends Stmt ::= VariableDecl initPart,
    Expr collection, Stmt body;
```

The ForStmt node extends the Stmt node, initPart is a VariableDecl, condition is an optional Expr, incrPart a list of Stmts, and the body is a single Stmt. Node types with accessor methods are automatically generated and the interface for the ForStmt is shown below:

```
class ForStmt extends Stmt {
  int numInitPart() { ... };
  VariableDecl initPart(int index) { ... };

  boolean hasCondition() { ... };
  Expr condition() { ... };

  int numIncrPart() { ... };
  Stmt incrPart(int index) { ... };

  Stmt body() { ... };
}
```

Definitions of AST-node declarations, attributes, and methods are grouped in modules where each module describes a certain aspect of the system. An aspect that transforms uses of the old collection iteration idiom into an iteration using the new for construct is show below. Four boolean methods checking one pre-condition each are woven into the ForStmt class using a notation similar to static introduction in AspectJ[KHH[+]01]. The first method, *checkInitPart*, uses the existing name-binding framework to lookup the *java.lang.Iterable* interface. The type of the *initPart* is then verified to be an instance of that particular interface. The second method, *checkCondition*, is verifying that the condition is a method access and that the left hand side of the invokation is referencing the variable declared in the *initPart* and that the method invoked is named *next*. The third method, *checkIncrPart*, then verifies that there are no *incrStmts*. The body is verified using a generic traveler that verifies that there is a single access to the variable declared in the *initPart* similar to *checkCondition*.

   The methods are used for pre-condition checking when rewriting ForStmts. The conditional rewrite declaration, *rewrite ForStmt*, states that each emphForStmt node is to be rewritten to an *IterateStmt* when *checkInitPart*, *checkCondition*, *checkIncrPart*, and *checkBody* are all. The rewrite declaration then rewrites the old statement into the new one. The *Body* is first transformed by invoking the *rewriteBlock* method that is a generic tree traveler that rewrites the body top-down. When the body is rewritten the type of the initPart is changed to java.lang.Object and the initializer is used as a collection expression. A new IterateStmt is finally created taking the modified initPart, collection, and block as parameters.

```
aspect RenovateIteration {
  boolean ForStmt.checkInitPart() {
    TypeDecl iterable = lookupType("java.lang.Iterable");
    return variableDecl().type().instanceOf(iterable) &&
      variableDecl().hasInit();
  }

  boolean ForStmt.checkCondition() {
    if(condition() instanceof MethodAccess) {
      MethodAccess m = (MethodAccess)condition();
      if(m.reference().decl() == variableDecl() &&
          m.name().equals("next"))
        return true;
```

```
    }
    return false;
  }

  boolean ForStmt.checkIncrPart() = numIncrPart() == 0;

  boolean ForStmt.checkBody() { ... }

  rewrite ForStmt {
    when (checkInitPart() && checkCondition() && checkIncrPart()
        && checkBody())
    to IterateStmt {
      body().rewriteBlock(initPart());

      initPart().setType(new TypeAccess("java.lang.Object");

      Expr collection =
        ((MethodAccess)initPart().assignmentInit()).reference();

      return new IterateStmt(initPart(), collection, body());
    }
  }

  void ASTNode.rewriteBlock(VariableDecl varDecl) {
    for(int i = 0; i < numChild(); i++) {
      if(child(i) instanceof MethodAccess) {
        MethodAccess m = (MethodAccess)child(i);
        if(m.reference().decl() == varDecl
            && name().equals("next")) {
          setChild(i, m.reference());
        }
      }
      child(i).rewriteBlock(varDecl);
    }
  }
}
```

From the specification above, our ReRAG tool JastAdd II generates a Java implementation that automatically carries out the renovation of input programs.

4 Conclusions

In this paper we have discussed the need for program renovation when languages evolve, and exemplified how such renovation can be supported by ReRAGs. We have implemented a full Java 1.4 grammar in ReRAGs, and are currently extending our implementation to Java 1.5. Within this work, we are also experimenting with how to renovate existing Java 1.4 programs to exploit the new constructs in Java 1.5. In our experience so far, we have found several features of ReRAGs to be important for this task.

- The object-oriented basis of ReRAGs gives a natural representation of the program, familiar to people with an object-oriented background.
- The attribution specified in the Java grammar provides a framework that is easy to use in specifying the renovation transformations. For example, the AST can be easily navigated, both along the AST hierarchy and along other dependences such as use-declaration links and sub-superclass links. The attribution also includes properties of the different AST nodes, e.g., type information, that is readily available in order to define the conditions for the rewrite.
- The renovation rewrites are expressed as aspects (similar to AspectJ static introduction) that allows them to be expressed in a modular way, separate from the Java grammar, but allowing access to the AST framework.

As a result, transformations can be expressed in a both modular and natural way, and with a small effort. We intend to continue this work by exploring additional examples, both concerning new or changed language constructs, and concerning changed framework APIs.

References

[CHHP91]   James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. Txl: a rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[EH04]   Torbjörn Ekman and Görel Hedin. Rewritable Reference Attributed Grammars. In *Proceedings of ECOOP 2004*, 2004. Accepted for publication.

[GJSB00]   James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[KHH+01]   Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[Sar99]   Joao Saraiva. *Purely functional implementation of attribute grammars*. PhD thesis, Utrecht University, The Netherlands, 1999.

[vdBHdJ+01] Mark van den Brand, Jan Heering, Hayco de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter Olivier, Jeroen Scheerder, Jurgen Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proceedings of Compiler Construction 2001*, LNCS. Springer, 2001.

[VSK89]   H. H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the SIGPLAN '89 Conference on Programming language design and implementation*, pages 131–145. ACM Press, 1989.