Extending Java with the Exponentiation and Safe Navigation Operators

Emma Asklund
D14, Lund University, Sweden
dat14eas@student.lu.se

Victor Winberg
D14, Lund University, Sweden
dat14vwi@student.lu.se

Abstract

This report describes the implementation of the exponentiation and safe navigation operators as extensions to the Java compiler ExtendJ. Our implementations of these operators in ExtendJ is functional and can be used to compile existing Java programs. Bytecode generation is also supported. The implementation of these operators is small, thanks to mapping to pre-existing language constructs.

We have evaluated the safe navigation operator by looking into benefits of using the save navigation operator in three Java and three Groovy open source projects. The conclusion after looking into these six projects is that the safe navigation operator is not as useful as we thought it would be.

1 Introduction

Exponentiation and safe navigation are two common operators available in, for example, the Groovy programming language. Exponentiation is useful in mathematical expressions, and safe navigation is used to simplify code navigating through nullable refernces. These operators do not exist in the Java programming language, as of Java version 11. Especially the safe navigation operator would be useful in Java code, which often deals with null references.

In this project, we investigate how the exponentiation and safe navigation operators can be implemented as extensions to the extensible Java compiler ExtendJ. We implemented the operators by using desugaring, a technique which allows us to simplify code generation for the new operators.

The implementation process was mostly straightforward, but we did encounter a couple of challenges which we discuss in this report. Adding features to a language does require new code to be written but depending on how you implement features to a language the difficulty can differ. ExtendJ has the advantage of using static aspects and *Reference Attribute Grammars* (RAGs) for describing its semantics [1].

Our research question is:

RQ1. Can Java code be simplified using the safe navigation operator?

The rest of this paper is organized as follows: Section 3 briefly describes the ExtendJ compiler. Section 4 explains the implementation for both operators with code snippets from the implementation. Section 5 gives an evaluation of our research question and section 6 discusses related work.

Course paper, EDAN70, Lund University, Sweden
January 12, 2019.

2 Background

The exponentiation operator takes the first argument to the power of the second one. This operator exist in several other programming languages, e.g. Groovy [2], Python [3] and Javascript [4].

The safe navigation operator is a newer operator and is used in some programming languages, like Groovy [2] and C#. The safe navigation operator works similar to a regular dot expression, but it performs a null-check on the value before the operator and if not null performs the regular dot operation.

If we take a simple example of using tree structured nodes we could create a class called `Node` that have the variables left and right that are both nodes, see below:

```
public class Node {
  public Node left;
  public Node right;

  public Node() {}
  public Node(Node left, Node right) {
    this.left = left;
    this.right = right;
  }
}
```

If we would like to reach a node at the depth three at the most left hand side we could reach it by calling left three times on the original node with a normal dot expression, like this:

```
Node found = original.left.left.left;
```

However, this could cause a null pointer exception. To make the code more robust, we would need to check for null at each level, e.g:

```
Node found = (original != null
           && original.left != null
           && original.left.left != null)
            ? original.left.left.left : null;
```

This is unnecessarily long code for simple null safe navigation checks. With the safe navigation operator, the code above could instead be almost as short as the original non-null safe statement, that is:

```
Node found = original?.left?.left?.left;
```

3 ExtendJ

In this section we give an brief overview of the Java compiler ExtendJ and its metacompiler JastAdd [1].

ExtendJ is open source, implemented using JastAdd [5], a metacompiler supporting Reference Attribute Grammars [6], has support for Java 8, and was build with the goal of being easily extendable. The compiler has an architecture that is possible for extensions to add, remove and combine features [7].

4 Implementation

In this section we will discuss the technical parts of the project by showing and explaining code snippets from our implementation of the two operators.

ExtendJ extensions can extend the Java language with new constructs and new behaviour. New constructs are added by new tokens in the lexical analyzer and new parser rules. The behaviour of the language is extended with attribute equations.

We have followed a similar structure when implementing both operators. First, the parser is modified, then the scanner and after that, we added new attribute equations to analyze uses of the new operators for type errors, and to generate bytecode.

4.1 The exponentiation operator

The exponentiation operator can be used for raising an expression to the power of another. For example the math expression 3^x is equal to the expression `3**x`.
In Java, this can be seen as syntactic sugar for `Math.pow(3, x)`. Using a lexical analyzer generator we could extend the ExtendJ compiler simply by adding the ”**” string to the language with the code snippet below:

```
<YYINITIAL> {
  "**"  { return sym(Terminals.POW); }
}
```

The exponentiation operator is detected by our lexical analyzer. Then we need to proceed by using a LALR parser generator that will extend the language by using the new terminal `POW` by the (simplified) EBNF parsing rule below:

```
Expr mul_expr =
    expr.e1 POW expr.e2
    {: return new PowExpr(e1, e2); :}
  ;
```

This adds a new production for the multiplicative expressions to a new `PowExpr`. The `PowExpr` will be defined with desugaring in the JastAdd language.

The exponentiation operator is now both detected and parsed to a *PowExpr* class. First we defined the JastAdd class of the *PowExpr* using the abstract grammar syntax below:

```
PowExpr : MultiplicativeExpr ;
```

In the code snippet above we define an abstract grammar rule with the *PowExpr*-class as a subclass of *MultiplicativeExpr*.

To implement *PowExpr* we use a higher-order attribute to compute the desugared form of the expression. The attribute is defined by the following JastAdd code:

```
syn nta Expr PowExpr.desugared() {
  args = new List(left.tree, right.tree);
  type = new TypeAccess("java.lang", "Math");
  method = new MethodAccess("pow", args);
  return new Dot(type, method);
}
```

This defines a synthesized nonterminal attribute *desugared* on PowExpr. To know which nodes to use in the desugared AST, we wrote a simple test program that worked as we wanted our new function to work, then we printed the AST for the hand-constructed program to use as template for the desugared AST. This generated a tree showing which nodes we wanted to desugar. We wrote the small program:

```
int x = 4;
x = (int) (x ** 5);
```

Then printed the AST (simplified):

```
VarDeclStmt
  PrimitiveTypeAccess ID="int"
    VariableDeclarator ID="x"
        IntegerLiteral LITERAL="4"
ExprStmt
  AssignSimpleExpr
    VarAccess ID="x"
    CastExpr
      PrimitiveTypeAccess ID="int"
        PowExpr
          VarAccess ID="x"
          IntegerLiteral LITERAL="5"
```

The desugared expression is used to generate code for exponentiation expressions by adding the following code snippet:

```
public void PowExpr.createBCode(CodeGen gen) {
  desugared().createBCode(gen);
}
```

4.2 The safe navigation operator

The safe navigation operator is used to simplify code navigating through nullable refernces. The syntax for the operator is a question mark followed by a dot. We added the following code to the scanner specifications to recognize "?." as a new operator:

```
<YYINITIAL> {
  "?."  { return sym(Terminals.NULLSAFEDOT); }
}
```

This generates a new NULLSAFEDOT terminal that will be used in the parser. To handle parsing, we added a new parser rule similarly to the exponentiation operator.

For the parsing aspect we added the *NULLSAFEDOT* to the parser's precedence file and that the the parser should create a *NullSafeDot* when a *qualified_name* was used.

Similar to how we did to figure out which tree structure to use in the exponentiation operator for desugaring, we used a simple program here as well, starting with:

```
if(a != null){
  a.b;
}
```

However this statement does not work because the AST is expecting an expression and the code shown above is a statement. To fix this we changed our program into using a conditional expression:

```
a != null ? a.b : null;
```

However, with this conditional expression we now have the problem that when checking multiple variables we could receive side effects. This due to each null check chained with one or multiple methods values could change values of internal variables. For example:

```
scanner.next()?.value
```

Would thereby call method *next* twice, once in check and once in return:

```
scanner.next() != null ?
  scanner.next().value : null;
```

We worked around this issue by using an anonymous class. By instantiating the Supplier interface with the Node class and storing the value in the get-method, it resulted in something like the code below:

```
Node left = new Supplier<Node>() {
  public Node get() {
    Node right = node.right;
    return right.left;
  }
}.get();
```

We first tried to generate the bytecode similar to how we did in the exponentiation operator using the desugaring pattern but found out that it did not work for all cases. Instead, we used a JastAdd AST rewriting mechanism. The difference between desugaring and rewriting is that with desugaring you can only add to the AST while rewrite can make modifications in it.

We used *rewrite* when getting bytecode to the safe navigation operator because we needed to add the null check before the dot operator. The code snippet below shows how we used the rewrite mechanism to perform the conditional expression check if null.

```
rewrite NullSafeDot {
  to Expr {
    NullLiteral nil = new NullLiteral("null");
    NEExpr neexpr = new NEExpr(
      getLeft().treeCopy(), nil);

    Dot dot = new Dot(getLeft().treeCopy(),
      getRight().treeCopy());
    return new ConditionalExpr(neexpr, dot, nil);
  }
}
```

5 Evaluation

Here we will evaluate our research question RQ1, if the operators could save a lot of time and lines of code if it is used in the right way. We will compare projects regarding their uses of null safe occurrences with different languages and sizes, by using resources online.

Comparing the two examples using the Node class from the background section, we can see that a null safe navigation operator is useful in some specific cases. However, to determine the benefit of adding a null safe navigation operator to a language we think it is important to compare the use of the null safe navigation operator in real examples and not only artifical examples. Therefore, we looked for examples form large GitHub repositories.

We used the following projects in our evaluation: *TheAlgorithms - Java* (8k lines of code) [8], *java-design-patterns* (25k lines of code) [9] and *spring-boot* (250k lines of code) [10]. In the first repository (TheAlgorithms) we found 13 cases where simple null safe navigation operator could be used to replace a simple null safe check:

Project	Algorithms	Patterns	Spring
Lines of code	8k	25k	250k
Null safe occ	13	30	200

Table 1. Three big Java projects from GitHub compared

Project	Geb [11]	Grails [12]	Nextflow [13]
Lines of code	24k	37k	76k
Null safe occ	70	200	734

Table 2. Three big Groovy projects from GitHub compared

```
if (right != null) {
  right.traverse(visitor);
}
```

Could be converted into:

```
right?.traverse(visitor);
```

Given the thousands lines of code in table 1 it is just a very small amount of instances where the safe navigation operator could be used. For the more complex cases of nested null checks we could not find any instances of nested null checks as described earlier.

Instead of using null checks for navigation we found that most of the null checks were used to check whether or not the object sent as a parameter to a method was null or not. Perhaps those cases could be some sort of indirect case were the null check would appear ”further down” in the code with a safe navigation operator, e.g:

```
if (added != null) {
  list.insert(added);
}
```

The other simple null checks were used to determine if the method was done or not fetching all the required variables, e.g:

```
if (user != null) {
  return user;
}
user = dbConn.get(username);
user.setActive();
return user;
```

Although we could create artificial examples where safe navigation makes a major difference, we couldn't find any real examples where it could decrease complexity. Furthermore, we compared projects in the language Groovy that has the safe navigation operator. Similarly to what we concluded that safe navigation operator is not used a lot in big Groovy projects either, see table 2.

We assumed that safe navigation would save a lot of lines of code. Simultaneously as we have proven that using safe navigation could have cases where the code would be much shorter and more readable, we haven't found any real example that complex as we showed earlier. Furthermore, the amount of code saved in real life projects we found is pretty small. Also, since we couldn't properly succeed implementing this operator completely, we do not see the operator to be easily implemented either. In conclusion, we have not found enough motivation for implementing the safe navigation operator. However, it could still be an useful operator if there is some proofs from real projects that contradicts what we found and instead benefits from using safe navigation operator. It is also about the balance between having too many and too few operators and what benefits they add compared to the complexity it adds.

6 Related work

Previous extensions to ExtendJ have added operators to the compiler similarly to how we implemented the exponentiation and safe navigation operators, e.g. Chonewics and Stenström [14] who implemented the print-assign operator and partially implemented the spread-operator. The print-assign operator is a new operator that prints the value when assigning a value to some variable, which is mainly used for debugging code. The spread operator is a newer operator but also used in some programming languages, e.g. Groovy [2].

Another project that extended Java with ExtendJ was Hjelm's and Olsson's project [15]. In their report, they explain the changes made to support Java 9 and how their changes impact the existing compiler. Both the mentioned projects are done under the same circumstances as our project.

7 Conclusion

In this report, we have described our implementation of the exponentiation operator and the safe navigation operator as extensions to Java with ExtendJ.

The safe navigation operator is an operator that can make the code more easy to read if a null check needs to be done before navigating through, or calling a method on, a nullable reference. However, after investigating different Java projects, we have shown that safe checks is used in a very small amount.

We managed to implement a simple safe navigation operator used for variables only with a small amount of lines of code.

Our conclusion to our research question is that safe navigation saves at most a very small amount of code.

Future work could investigate the differences in implementing new operators, and extensions, in different compilers.

Acknowledgments

We would like to say thank you to our supervisor, the maintainer of ExtendJ, Jesper Öqvist, for the help with our implementations and valuable feedback on this report.

References

[1] G. Hedin and E. Magnusson, "Jastadd—an aspect-oriented compiler construction system," *Science of Computer Programming*, vol. 47, no. 1, pp. 37–58, 2003.

[2] "Groovy operators," Apache Groovy, 2018. [Online]. Available: http://groovy-lang.org/operators.html

[3] "operator — standard operators as functions," Python Software Foundation, 2018. [Online]. Available: https://docs.python.org/3.4/library/operator.html

[4] "Ecmascript 2016," Ecma International, 2016. [Online]. Available: https://www.ecma-international.org/ecma-262/7.0/#sec-exp-operator

[5] G. Hedin, "Reference attributed grammars," *Informatica (Slovenia)*, vol. 24, no. 3, pp. 301–317, 2000.

[6] G. Hedin, *An Introductory Tutorial on JastAdd Attribute Grammars*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 166–200. [Online]. Available: https://doi.org/10.1007/978-3-642-18023-1_4

[7] "Extendj - the jastadd extensible java compiler," 2018. [Online]. Available: https://extendj.org/index.html

[8] "All algorithms implemented in java," The Algorithms, 2018. [Online]. Available: https://github.com/TheAlgorithms/Java

[9] I. Seppälä, "Design patterns implemented in java," 2018. [Online]. Available: https://github.com/iluwatar/java-design-patterns

[10] "Spring boot," Spring, 2018. [Online]. Available: https://github.com/spring-projects/spring-boot

[11] "Very groovy browser automation," Geb, 2018. [Online]. Available: https://github.com/geb/geb

[12] "Gorm - groovy object mapping," Grails, 2018. [Online]. Available: https://github.com/grails/grails-data-mapping

[13] "A dsl for data-driven computational pipelines," nextflow-io, 2018. [Online]. Available: https://github.com/nextflow-io/nextflow

[14] W. Chonewics and F. Stenström, "Extending java with new operators using extendj," *Project in Computer Science, Lund University, 2017*, 2017.

[15] S. Hjelm and M. Olsson, "Extending the extendj java compiler with java 9 support," *Project in Computer Science, Lund University, 2017*, 2017.