Extending Java with new operators

EDAN70 - Project in Computer Science

Hans Bjerndell
Lund University
dat12hbj@student.lu.se

Linus Lexfors
Lund University
bas11lle@student.lu.se

Abstract

We have explored the possibilities of extending the syntax of the Java programming language with a new operator, known as the spread operator, using the extensible Java compiler ExtendJ[1]. The spread operator is a binary operator that accesses a field or calls a method on each object in a collection, a feature we believe to be highly useful. Our extension to ExtendJ consists of new parsing rules, static analysis and a new node type in the abstract syntax tree (AST). Bytecode generation is left as future work, the extension we have developed serves as a basis for further development. In this report we describe the process of developing the extension and provide examples of the spread operator in use. Finally we evaluate the extended compiler, measuring compile-time and comparing the results with the ExtendJ base compiler.

Keywords compiler, extension

1. Introduction

In this project we have implemented the spread operator, developing an extension module to ExtendJ[1], an extensible Java compiler. In the background section we overview the different concepts ExtendJ works upon, as well as JastAdd which ExtendJ is implemented using, along with a description of the spread operator as it is implemented in the Groovy programming language[2].

The motivating examples section exhibits use cases of this operator as well as a desired side effect: reducing source lines of code (SLOC), improving readability and elegance of the code. In the implementation section we give an overview of the different parts of the extension, followed by a more detailed description of these parts, in the form of code excerpts. The result is a compiler supporting the spread operator through new parsing rules, extended abstract grammar and static analysis. Bytecode generation was not implemented due to time constraints.

General purpose programming languages that are widely used today, such as Java, are often extended and abstracted in order to implement domain-specific concepts. There are two very different ways to add functionality to a programming language; using libraries that are compiled together with the source code, or by extending the compiler with new syntax and behavior.

Examples of libraries implementing domain-specific concepts in the Java language are Pattern, from the Java standard library, implementing regular expressions to recognize patterns in strings or the Java Database Connectivity library which implements database handling. While offering helpful abstractions, a library does not change the language at its core, if a new keyword or operator is needed, the compiler has to be modified.
Syntax extension differs radically from using libraries to extend functionality. The compiler, which defines the language, needs to accept new keywords and/or new operators and new parser rules for the added syntax must be defined. Semantic analysis also has to be implemented so that the extensions provide the expected behavior and meaning. The spread operator serves as a good candidate since extending the compiler to support it involves all the different steps of compiling mentioned above.

Many different extensible compiler solutions exists, using different approaches to let the programmer extend functionality and modify syntax in an existing language, and we have briefly looked at a few of them. SugarJ is an extensible language built on top of Java that uses a library-centered approach to modify syntax[6]. Maya[4] uses generic functions as grammar productions, with multimethods to implement semantic actions, transforming the abstract syntax tree(AST). Polyglot[9] is similar to ExtendJ in that it uses the AST as the only structure being operated on, however it does not employ aspect oriented programming to define new AST nodes attributes and behavior, using a delegation mechanism instead.

2. Background

Our main inspiration for the spread operator comes from the Groovy programming language, which uses the Java platform[2]. In Groovy, the spread operator works on collections of objects, accessing a field or calling a method on each of the objects and returning the result in a new collection[3]. We find this functionality useful as well as intuitive: it works as syntactic sugar for what would require looping through the objects and invoking an action on each of them explicitly in Java. Groovy implements other interesting uses of the spread operator, such as using it with a list to inline the contents into another list, which are beyond the scope of this project.

2.1 JastAdd

JastAdd[7] is a meta-compilation system based on Reference Attribute Grammars. JastAdd uses an internal structure of an abstract syntax tree (AST), which is a tree representation of the abstract syntactic structure of the source code and as such each node in the tree represent a construct present in the code. To add attributes to AST nodes, JastAdd makes use of aspects which declare attributes and equations or ordinary fields and methods.

An excerpt of an aspect used in our implementation is shown below:

```
aspect Type {
  syn TypeDecl SpreadExpr.collectionOf() {
    return getCollection().type().iterableElementType();
  }
eq SpreadExpr.getAccess().lookupVariable(String name) {
  return collectionOf().memberFields(name);
  }
}
```

Above we see an example of a *synthesized* attribute, which means that it is defined by an equation in the node itself, in this case the node SpreadExpr. We also see an example of an equation which defines the parameterized attribute *lookupVariable(String name)* located in the child node Access in relation to SpreadExpr.

2.2 ExtendJ

ExtendJ[1] (previously JastAddJ[5]) is an extensible compiler implemented with the aforementioned meta-compilation system, JastAdd, and allows developers to extend Java with new functionality in a modular way. It has a foundation which consists of the abstract grammar, parsing, type analysis etc. of ordinary Java and as of writing this report it has support for up to Java 8, and is still being developed and maintained at Lund University. The developer can create modules consisting of additions to the grammar, parsing, scanning, and aspects to define the added functionality. When generating the compiler, the new module(s) are woven into the code foundation and the result is essentially a new language.

3. Motivating examples

The spread operator we have implemented is used for accessing fields and calling methods on collections of objects. This leads to concise statements using fewer SLOC compared to the equivalent program in Java, improving readability of the code as seen in the following examples.

With the spread operator we avoid the for-loop when we would like to call the method D on all objects of type A in the collection B:

```
B*.D();
```

The equivalent in ordinary Java (which does not support the spread operator):

```
for (A a : B) {
    a.D();
}
```

Aggregating fields in a collection of objects is accomplished in a similar fashion:

```
A = B*.D;
```

The resulting collection A is of the same collection type as B but with the template type of the field D. As we can see, we also avoid explicit construction of the collection before the assignment. This is generally necessary in ordinary Java, as shown in the equivalent example below:

```
A a = new A();
for (C c : B) {
    a.add(c.D);
}
```

[1] http://www.extendj.org

Lets look at a more involved example:

```
public class Person {
    private int age = 5;
    public int getAge() {
      return age;
    }
}

ArrayList<Person> persons = new ArrayList<>();
for (int i=0; i<10; i++) {
    persons.add(new Person());
}

for (Integer i : persons*.getAge()) {
    System.out.println(i);
}
```

The Person class has a private field and a getter method. In the final loop, the spread operator applies the method *getAge* to every element (Person object) in the list *persons*. The result of this is a list containing the corresponding return values (all of value 5), which is then iterated over and printed.

4. Implementation

Developing an extension to the ExtendJ compiler can be comprised of several different additions to all the steps involved in compiling, complementing the base scanner, parser and abstract grammar of ExtendJ. Implementing the spread operator involved making extensions to all of these parts. We start with an overview of the parts involved that needs modification and then venture into further detail on each of them.

Starting with the scanning step, a new token defining the spread operator (***.**) was introduced. The new token was then introduced into the grammar of the language, by introducing new productions involving the operator. A new AST node was added to the abstract grammar, modeling the expression containing the spread operator, and the grammar productions defined to create an instance of this node.

4.1 Parsing and scanning

JFlex is used to generate ExtendJ's scanner. Below we show the addition needed to accept the token for the spread operator:

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

The spread operator can now be referred to in the grammar as shown in the next example. Beaver is used to generate ExtendJ's parser. Below is an excerpt of how we extend the grammar with new productions:

```
%left SPREAD;
Access field_access =
 primary.c SPREAD simple_name.f
 | simple_name.c SPREAD simple_name.f
 {: return new SpreadExpr(c, f); :}
 | simple_name.c SPREAD method_invocation.f
 {: return new SpreadExpr(c, f); :}
 ;
```

Here the spread expression is defined to be left associative and is introduced into the grammar. Note that the *field_access* nonterminal already exists in the original ExtendJ grammar, and is simply extended with more productions. The semantic action builds a new AST-node type to represent the spread operator.

The SpreadExpr node is defined by the following abstract grammar declaration:

```
SpreadExpr : Expr ::= Collection:Expr Access;
```

SpreadExpr is a defined to be a subclass of Expr so that it can be used as a normal expression. The child *Collection* refers to the Java collection that the spread operator is used on. The Access node refers to a field or method in the object contained in *Collection*. These attributes are used for variable lookup, for example finding a declaration for the *Collection* operand, and for type analysis.

4.2 Analysis and error collection

We added attributes, equations and error collection contributions to define when the spread operator is used incorrectly.
Below is an example of a contribution to a collection attribute. The attributes uses methods and other attributes from the ExtendJ API. The target variable that the *SpreadExpr* is assigned to has to be iterable (such as ArrayList or HashSet) when returning a collection. The iterable requirement is implemented with a boolean attribute in the SpreadExpr node:

```
aspect Type {
  syn boolean SpreadExpr.targetIsIterable() {
    return targetType().isIterable();
  }
}
```

After the *targetIsIterable* attribute is defined, we use it to define a compilation error, by making *SpreadExpr* contribute an error to a error collection attribute in the *CompilationUnit* node:

```
SpreadExpr contributes error("Illegal use of spread
operator, target type is non-iterable:
" + targetType().typeName())
    when !targetIsIterable()
    to CompilationUnit.problems();
```

5. Evaluation

Since we have only implemented static analysis for the spread operator we only evaluate static analysis performance.
First by performing static analysis on a number of Java programs of varying size. Note that these programs are ordinary Java programs not using the spread operator. We measured the compilation time of the programs, using both the ExtendJ base compiler and our extended compiler and compared the measurements. Our expectation was that a project not using the added operator would not have an increased compilation time when using our generated compiler compared to an unmodified ExtendJ-compiler.

Secondly we performed static analysis on smaller pairs of equivalent programs, where only one of the programs uses the spread operator, and compared the compile-time.

We also aimed to evaluate the validity of our extension by testing multiple scenarios in which the use of the operator is expected to work as if only native Java code was used. For example to create the correct resulting collection when a spread operator application is passed as a parameter.

5.1 Measurement methodology

For each project/program we performed a compilation 30 times, measured the total execution time and used the results to calculate the mean. We then calculated a 95% confidence interval using a normal distribution for each measurement, which are presented in the figures as a vertical interval at the top of each bar.

5.2 ExtendJ only

We began by, for each project, running the compilation once to warm up the JVM, in order to obtain a fair measurement for the following compilations. We used two projects consisting of 1081 and 4792 lines of code respectively and the resulting compilation times are shown in figure 1.

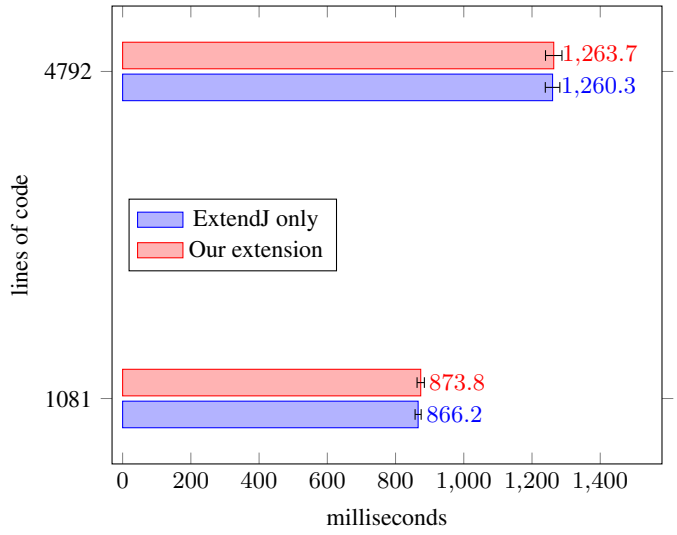

Figure 1. Total execution time.

5.3 ExtendJ with our extension

As with the previous compilations, we began by warming up the JVM before measuring the compilation time. We used the same projects as before and the resulting mean compilation times and confidence intervals are also shown in figure 1.

Compared to the results in figure 1, there was a slight increase in compilation time when we used the compiler with our extension, although it's not statistically significant and is likely not noticeable from a user standpoint.

5.4 Java

Next, we evaluated the compilation times for two simple programs in native Java, using our compiler extension. For this part, we used smaller projects in which we could easily find statements and/or expressions for which we would be able to use the spread operator instead, in order to achieve a fair comparison between the measurements. The resulting measurements are shown in figure 2.

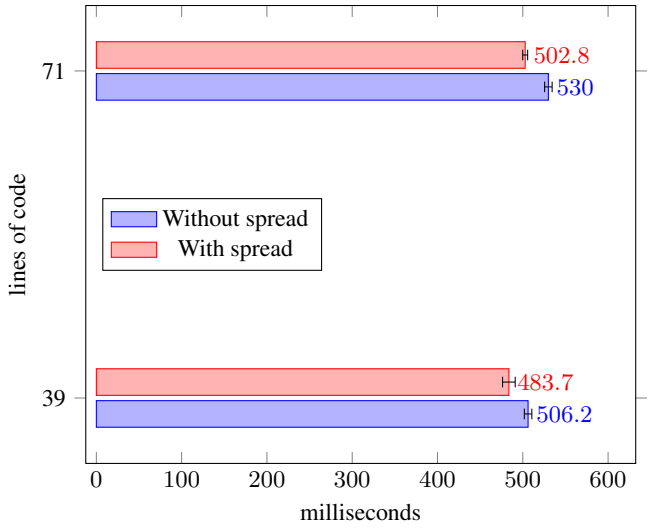

Figure 2. Total execution time.

5.5 Java using the spread operator

Finally, we evaluated the programs from the previous section, although modified to use the spread operator where applicable. The projects are equivalent to the previous ones in all but the syntax. The resulting compilation times are shown in figure 2.

We can see a small decrease in compilation time when using our spread operator. It is likely due to fewer SLOC and that the resulting AST:s has fewer nodes, about 130 fewer in both examples, compared to that of the programs using native Java, which is an important metric to keep in mind as the compiler only performs a static analysis and is not generating bytecode.

5.6 Test cases

During development we created a suite of 25 test cases, to test the correctness of the operator. The tests are divided into two parts, covering field access and method access application of the spread operator. The operator is used in a variety of expressions covering all use cases we could think of. Below we demonstrate some of the more complicated test cases we have considered. The typical error is a mismatch between the field or method type accessed by the operator, and the field/method in the objects in *Collection*.

Ex 1, using the spread operator inside methods.

```
public ArrayList<Integer> getAges
          (ArrayList<Person> persons) {
    return persons*.age;
  }
// Below: correct usage, type match.
ArrayList<Integer> ages = getAges(lista);
// Below: compiler error: type mismatch.
ArrayList<Double> ages = getAges(lista);
```

Ex 2, nested spread application. Although not fully supported, our implementation handles cases of nested spread expressions where no assignment is being made. In the following example, the *Person* objects contained in the *persons* collection has a *Car* object as a private field. The *Car* objects has a void-method *printInfo*, which prints the year and the model of the car:

```
persons*.getCar()*.printInfo();
```

As the spread operator is left associative, the leftmost spread operator returns a collection of *Car*, which is then passed to the second spread operator. In the next section we display the abstract syntax tree(AST) for this nested spread case.

5.7 Dr AST

Dr AST[8] is a attribute debugger tool developed at Lund University. Dr AST allows developers to inspect the AST created by the compiler for a specific program. We have used Dr AST extensively in development to debug our compiler. All the test cases have been loaded into Dr AST, where one can easily compute the attributes of nodes in the AST and get a clear picture of the structure of the program.

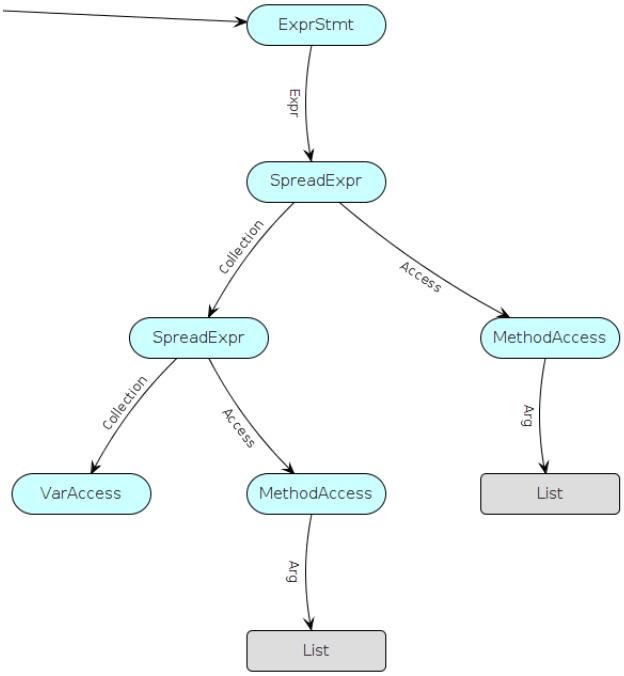

Figure 3. AST for Ex 2 under test cases

6. Conclusion

We developed a compiler extension on top of ExtendJ, implementing the spread operator. We added new parser rules and grammar productions and introduced a new AST-node to the abstract grammar. Static analysis was implemented using the reference attribute grammar system of JastAdd. We have evaluated the compiler-extension through compile-time measurements and test cases. The result is a new compiler capable of building a correct abstract syntax tree for a program using the spread operator, along with suitable error messages when the operator is used incorrectly. Our solution is limited to static analysis as bytecode generation proved too time consuming within the scope of the project. We however believe this project can serve as a basis for further development.

During the project we learned a lot about extensible compilers and the process of developing an extension of our own. This included further insight into the JastAdd system of synthesized and inherited attributes, as well as working with a large, perhaps somewhat outdated, API documentation. We underestimated the amount of work required and the difficulty involved in developing the extension, this perhaps was the most important lesson learned. Our background in the field comes solely from the Compilers course (EDAN65) and we have a new appreciation of the difficulty in static analysis when introducing more complicated structures such as collections and generics to a programming language.

6.1 Working with ExtendJ

It has been an interesting experience working with ExtendJ, although not entirely without difficulty. The main problem we encountered was that the API documentation was incorrect or outdated for certain classes, for instance there could be one or more classes called something similar to each other in the API but something slightly different in the working repository. This caused unnecessarily long periods of trying find and then match the actual classes to the API in order to be certain of correct usage. We also encountered constructor and method signatures that did not correspond to the actual classes, causing originally quite simple additions to take significantly more time to implement due to both build errors caused by (unknowingly) incorrect usage and the ensuing time it took to debug. It may sound like trivial problems but taken our unfamiliarity with ExtendJ and its API into account it amounts to a significant threshold to overcome, and it is likely that new developers may encounter the same.

6.2 Future work

We aimed to create an extension as complete as possible but unfortunately some aspects of the work proved to be more time consuming and complex than we first anticipated. The main aspect of the extension that we did not manage to implement was bytecode generation, which would have made it quite a bit more interesting in terms of usability and for our own part; evaluation.

Our implementation currently does not fully support nested application of the operator, such as:

```
ArrayList<String> strings = list*.getAges()*.toString();
```

Our extension to the grammar does not allow expressions of this kind. The parser rules could be modified to support this while retaining current properties.

The compiler currently does not generate an error when a private field is being accessed through the spread operator. Correctly implementing visibility constraints would be crucial to a complete implementation.

Bytecode generation is the most important feature not developed in this project. This would require a general approach capable of generating the correct bytecode in the multiple different expressions where the spread operator can be used.

Acknowledgments

We have benefited greatly from our meetings with supervisor Jesper Öqvist, the maintainer of ExtendJ. He has given us advice on everything from implementation to writing this report.

References

[1] Extendj homepage. `http://extendj.org/`. [Online, accessed 30-november-2016].

[2] Groovy programming language, . URL `http://groovy-lang.org/`.

[3] Spread operator in groovy, . URL `http://docs.groovy-lang.org/latest/html/documentation/index.html#_spread_operator`.

[4] J. Baker and W. C. Hsieh. Maya: Multiple-dispatch syntax extension in java. *SIGPLAN Not.*, 37(5):270–281, May 2002. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/543552.512562`.

[5] T. Ekman and G. Hedin. The jastadd extensible java compiler. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 1–18. ACM, 2007. ISBN 978-1-59593-786-5. . URL `http://doi.acm.org/10.1145/1297027.1297029`.

[6] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, 46(10):391–406, Oct. 2011. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2076021.2048099`.

[7] G. Hedin and E. Magnusson. Jastadd - a java-based system for implementing front ends. *Electr. Notes Theor. Comput. Sci.*, 44(2):59–78, 2001. . URL `http://dx.doi.org/10.1016/S1571-0661(04)80920-4`.

[8] J. Lindholm, J. Thorsberg, and G. Hedin. Drast: an inspection tool for attributed syntax trees (tool demo). In T. van der Storm, E. Balland, and D. Varró, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*, pages 176–180. ACM, 2016. ISBN 978-1-4503-4447-0. . URL `http://dl.acm.org/citation.cfm?id=2997378`.

[9] A. C. M. Nathaniel Nystrom, Michael R. Clarkson. Polyglot: An extensible compiler framework for java. *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, 2003. URL `https://www.cs.cornell.edu/andru/papers/polyglot.pdf`.