Extending Java with new operators using ExtendJ

Wawrzyn Chonewicz
E13, Lund University, Sweden
elt13wch@student.lu.se

Filip Stenström
E13, Lund University, Sweden
elt13fst@student.lu.se

Abstract

We have used the extensible Java compiler ExtendJ to add two new operators to the java language: print-assign and a simplified spread operator. The print-assign operator extends the assignment operator by also printing, and the spread operator creates a new array by applying a method to each element of an array. In this paper we describe the implementation, which consists of both static analysis and bytecode generation, and highlight some interesting problems.

We evaluate the applicability of our operators by scanning a few open-source projects for use cases. We also evaluate the performance of our extension by compiling a few open-source projects, and we conclude that there is no noticeable difference in compilation time with or without our extension.

1 Introduction

In this report we describe how we added two new Java operators to the extensible Java compiler ExtendJ.

In our extension, the new operators we implemented are: print-assign (.=) and spread (*.). These operators are meant to work as syntactic sugar for common code patterns.

The print-assign operator is a new operator we have not seen before. It works like the ordinary assignment operators, except that it also prints the name and runtime value of the assigned variable. We designed this operator to be a convenient tool for debugging.

We implemented a simplified version of the Spread operator from the Groovy language [4]. That is, the operator calls a method on each element in a collection and returns the result as a collection. However, our simplified version only allows spreading of arrays. Our implementation of the spread operator does not allow several spreads, qualified access or method calls. More functionality was initially planned, but was cut due to the project's limited time.

We will mainly focus on the spread operator in this paper, since it proved to be much more complex to implement compared to print-assign. We will highlight some interesting problems we had during the implementation.

We evaluate our extensions by comparing compile times, and by showing to what extent our operators could replace code used in existing open-source projects.

This project is a part of the *Project in Computer Science* course at Faculty of Engineering LTH, and there have been previous efforts in implementing operators using ExtendJ [1]. However, this project is starting from fresh, and is therefore independent of previous ones.

To evaluate our implementation we scanned open-source projects for use cases of the operators. The result was that there were few cases where the spread operator could be applied, since the operator only supported arrays and not `Iterables`. For print-assign we found no use cases, however it is meant to be a debugging tool. This means that its usage never never makes it to public repositories.

There exist other extensible compilers. One such example is ableJ [3] which much like ExtendJ uses grammar attributes. Our implementation is limited to ExtendJ though, and we have therefore not made an comparison to other extensible compilers.

2 Background

In this section we give an overview of ExtendJ and how to generate bytecode by using a method called desugaring.

2.1 ExtendJ overview

ExtendJ is an implementation of a Java compiler using attribute grammars. ExtendJ is built with JastAdd, a meta compilation framework supporting reference attribute grammars. [2]

Static code analysis is performed by attributes, and code is generated by traversing the abstract syntax tree (AST) while accessing attributes to compute the bytecode. ExtendJ allows for modification and extension of the scanner, parser, AST attributes, and abstract grammar specifications. The attributes can be modified by using aspect oriented programming with JastAdd.

Extensions to ExtendJ are developed as modules. Modules consist of aspects, attributes, scanner and parser additions, and AST type declarations. ExtendJ has a base implementation of Java 1.4 and modules for each new version of Java, up to Java 8. Each module depends on the base and the earlier versions.

2.2 Desugaring

A simple way to avoid writing custom bytecode is to use a method called desugaring. Desugaring means that a complex language construct is translated into a normalized form using simpler language constructs. Desugaring is analogous to finding the code in question and then replacing it with code that is written using existing language features, and then compiling that code instead. We use this method to implement both the spread and the print-assign operator.

Course paper, EDAN70, Lund University, Sweden
January 13, 2018.

Instead of writing a bytecode generation method for the new language construct, the existing ones are used on the generated sub tree.

By using desugaring, the implementation challenge is to construct an AST translations for a new language feature. The upside of this is that the language developer isn't required to have the domain knowledge in order to write Java bytecode.

The downside is that the solution might not be as optimal as it could have been had it been written in bytecode. This is because desugared code performs certain known operations on specific data. In many cases the state of the virtual machine as well as some things about the data operated on are known, which allows for writing bytecode that doesn't have to work for cases that don't conform with that knowledge, hence allowing for more optimal code.

3 Implementation

In this section we describe the technical aspects of the project. In order to make the code snippets understandable, some parts might be left out, or be written in pseudo-code. A link to the source code can be found in Appendix B

For both operators, we first extended the scanner, parser and abstract grammar. After that, we implemented static analysis followed by code generation.

3.1 Parsing

In order for the compiler to accept the new operator syntax, the language grammar had to be extended with new productions. This was made easy by the ExtendJ framework which combined the parser specification for Java itself with the additions from our project.

The changes in order to add the print-assign operator were to simply specify the operators' grammar. The only difference from normal assignment was the operator token itself. The production for an existing non-terminal, `assignment`, was appended with our own production, allowing for the usage of the new operator the same way the normal assignment operator is used:

```
Expr assignment =
    postfix_expression PRINTASSIGN assignment_expression
  ;
```

We also allowed to use the print-assign operator during initialization of new variables. In order to add this, the `variable_declarator` production had to be appended with a construction that, like in the previous case, only differed from normal assignment by the operator token itself:

```
VariableDeclarator variable_declarator =
    IDENTIFIER PRINTASSIGN initialiser
  ;
```

The parser extension for the spread operator was considerably more involved. Because there are many things that could go wrong, the implementation was done incrementally during the course of the project. This way, we could make sure that the small additions were working correctly before adding the next one. In case something worked incorrectly, there was a good chance that the anomaly originated from the feature currently being added.

The language extension for spread was appended to the existing non-terminal `primary_no_new_array`, in order to conform with the precedence rules in Groovy. The specification was as follows:

```
Expr spread_dot =
    simple_name SPREAD_DOT method_invocation
  ;

Expr primary_no_new_array =
    spread_dot
  ;
```

With this production, spread could be used with an identifier and an unqualified method invocation syntax (also defined in the parser specification), for example as follows:

```
System.out.prinln(array *. methodName());
```

Later, the grammar was extended in order to allow spread expressions to be standalone statements by adding spread to the `statement_expression` production:

```
array *. methodName(); //Result is being discarded
```

Finally, in preparation for allowing to chain spread expressions, spread operators production was extended in order to accept most primary expressions on the left-hand side, as for example a method call or even a spread expression itself:

```
getStringArray() *. toString() *. toLowerCase();
```

This however required major refactoring of the grammar in order to avoid shift-reduce conflicts. First, a new production, `primary_no_lambda` was created, containing a copy of the productions from `primary` except for `lambda_expression`. Then `primary_no_new_array` was overwritten, allowing it to be:

```
Expr primary_no_new_array :=
    lambda_expression
  | spread_dot
  | primary_no_lambda
  ;
```

Finally, the spread grammar was modified to be either accept either an identifier on the left-hand side or a primary expression with the exception of `lambda_expression`:

```
Expr spread_dot =
    simple_name SPREAD_DOT method_invocation
  | primary_no_lambda SPREAD_DOT method_invocation
  ;
```

These parser rules effectively disallow for a lambda expression to be on the left-hand side of the spread expression, while not completely removing lambdas from the language. The reason for this can be explained with the following example:

```
arr -> arr *. someMethod();
```

The above code could construct either an AST where the lambda expression `arr -> arr` is being spread, or an AST where a lambda containing the spread expression `arr *. someMethod()`.

We decided to not include spread chaining in the release due to lack of time to solve an issue with bytecode generation. The grammar was further constrained to disallow that by removing spread production from the expressions allowed to be on the left-hand side. The ability to use richer language constructs like method invocations and array accesses on the left-hand side still made it to the release however.

3.2 Reuse of old constructs

In order to avoid reinventing the wheel, operators in this project extend other, already existing operators in the java language. The print-assign operator, which as mentioned before, technically consists of two separate AST classes, `PrintAssign` and `PrintAssignVariableDeclarator`, extend `AssignSimpleExpr` and `VariableDeclarator` respectively. Because of this, all of the static analysis like type and name checking is inherited without modification.

The spread operator extends the dot operator. The built-in dot operator is used for qualified access in Java. Because the behavior of the dot operator is not desired in all cases in our spread operator, some things needed to be tweaked, like the method lookup and type analysis. Example usages of the Dot-operator can bee seen in the code sample below:

```
this.a = 10; //Member access
x.m(); //Method access
new A().foo(); //Method access
```

3.3 Static analysis

The print-assign operator did not require any type of static analysis that was not already inherited from the assignment operator and variable declarator.

For the spread operator, the first and obvious analysis that is performed is type checking, to make sure that the left-hand side is an array. This was done using already existing methods provided by the ExtendJ framework:

```
syn Collection<Problem> SpreadDot.typeProblems() {
    if(!getLeft().type().isArrayDecl()){
        //Error
    }
    ...
}
```

Another issue that could occur is when the programmer tries to spread an array using a method that is not a member of the objects contained in the array that is being spread. This case is handled in a different manner. Instead of explicitly checking if the method is a member, we override an attribute equation responsible for finding methods.

Because the spread operator extends the dot-operator, by default, the `lookupMethod` equation checks if the method is a member of the left-hand side object. This is wrong because the method is actually a member of the elements of the left-hand side. This error was corrected as follows:

```
eq Spread.getRight().lookupMethod(String name) {
    ArrayDecl var = getLeft().type();
    TypeDecl elemType = var.elementType();

    return elemType.memberMethods(name);
}
```

The above code will be automatically used by name analysis in the ExtendJ framework to check if the elements actually have a member method with given name, and at the same time, the reference stored in the Spread class will point to the correct method declaration. Because of that last part, no additional work had to be done in order to get a correct reference.

Type analysis for expressions is performed with attribute named `type()`. To ensure the correct result, the `type()` equation was overriden to return the correct type. The type of the result from the spread expression is an array of the return

type of the method used to spread the left-hand side array:

```
eq type() {
    if (getRight().type().isVoid()) {
        return typeVoid();
    } else {
        return getRight().type().arrayType();
    }
}
```

As can also be seen from the above code, our implementation of spread also allows for using void methods for spreading. In that case, the result is void, since it is pointless and impossible to create an array of type void.

3.4 Bytecode generation

We chose to use desugaring for both operators. This also made sense since both operators are essentially nothing more than syntactic sugar. However, despite the same method being used for both operators, the desugaring of the spread operator was significantly more challenging.

In order to build a correct AST, we used a class called JavaDumpTree, that is a part of ExtendJ. Instead of compiling Java code into a binary, it only parses the code and constructs an AST, after which it prints it in a human-readable format on standard output. Using this tool, it was considerably easier to write the parts of AST used for desugaring. We used it to determine the AST structure we needed to build for desugaring into a desired form.

We also used DrAST[7] which is a graphical AST-inspection tool. DrAST uses a compiler built with JastAdd and a source file as input and shows the graphical representation of the AST and each node's attribute values.

The base for our desugared spread is a `for`-loop. We decided to use `for`-loop over Java streams, since the latter would need different types of method calls depending on whether the return type was a primitive or not.

Some problems we had during bytecode generation were related to using JastAdd and creating non-terminal attributes (NTAs). One problem was that we used an NTA inside another NTA, when one requirement is that NTAs only may contain freshly created nodes (nodes that already exist in the AST may not be referenced).

Another problem was that we declared a node in the method for creating an NTA, and then used an NTA belonging to this new node. The problem was that the NTA inside the NTA used an inherited method from the node. Since the node did not yet exist in the AST, but was a temporary object in the method, no inherited attributes were created. This resulted in unexpected `NullPointerExceptions`. Without sufficient experience in using JastAdd, these kind of problems were hard to debug.

3.4.1 Print-assign

Desugaring of the print-assign operator was relatively straight forward. One reason for this is because the print-assign statement can simply be turned into an assignment and a standard output printout. The only difficulty in implementing this operator is that it is technically two separate operators, not one. One of them is used when an already existing variable is assigned a value. The second case is when a new variable is declared with an initial value.

The idea is to convert the following statements:

```
int a .= 10;
a .= 15;
```

into the following:

```
int a = 10;
System.out.println("a = " + a);
a = 15;
System.out.println("a = " + a);
```

In order to add this to the compiler, we simply override a method called `createBCode`, which is responsible for bytecode generation for our operator, and make it generate the bytecode for the assignment first and the printout second. Because our operator class extends the assignment operator, we did not have to handle the assignment part of the print-assign code generation. We instead only had to generate code for printing the result:

```
public void PrintAssignExpr.createBCode(CodeGeneration gen) {
    super.createBCode(gen);
    desugared().createBCode(gen);
}
```

Of course, analogous code was written for variable declaration with initialization value. The `desugared()` method constructs a tree with the printout statement. For the sake of brevity, we will omit the code that does this, as it gets large for even small trees.

On top of printing the variable contents, print-assign will also detect if the variable to be printed is an array, and if so, it will print the contents of the array using `Arrays.toString` or `Arrays.deepToString` for arrays with more than one dimension. This was done by checking if the variable is an array, and if so, constructing an AST for the following code instead:

```
System.out.println("a=" + Arrays.toString(a));
```

3.4.2 Spread

The process of desugaring, despite requiring considerably more work, is essentially the same as desugaring of the print-assign operator. There are however some differences which need to be discussed.

To give an example, to desugar the following code:

```
arr *. method(arg1, arg2)
```

The first step, as before, is to decide to what code the operator should be translated to. In the example, `arr` is an array of objects that have a method called `method(...)` which returns type `String`. The result of desugaring in this case should be:

```
String[] newArr = new String[arr.length]
for (int i = 0; i < arr.length; i++) {
    newArr[i] = arr[i].method(arg1, arg2);
}
(return newArr;)
```

It is the compilers task to find and replace the name of the array on the left-hand side, the method signature and the return type in the general case.

The last return statement is written in parenthesis because technically it is not present in desugared code. This is one of the big differences from the print-assign operator. Spread is an expression, meaning that its result should be left on the stack of the JVM for something else to use. In contrast, print-assign does its job and the stack remains unchanged.

When the compiler generates the bytecode, it computes the type of resulting array, the array to spread and the method to use for spreading using static analysis, as explained in previous sections. Because the operator also works on void methods, there are two different AST trees that can be generated. Which one is chosen depends of the type of the spread expression.

In cases where there is a resulting array, it needs to be left on the stack. This is done by generating bytecode for a `VarAccess` to the resulting array.

4 Evaluation

In this chapter we evaluate to what extent our new operators are applicable in a few open-source projects. We also evaluate the performance of the extensions by comparing compilation times with and without the extensions for a few other open-source projects.

4.1 Performance

In order to make sure that our additions didn't cause any unnecessary delays or errors, we used both an unmodified ExtendJ compiler as well as our compiler on open source projects from Qualitas Corpus. The averages of five are presented in Table 1.

Project	No additions	With additions
ant-1.8.4	28.62	27.89
argouml-0.34	54.47	54.95
azureus-4.8.1.2	93.33	89.88
cayenne-3.0.1	43.10	43.01
cobertura-1.9.4.1	10.89	11.26
antlr-4.0	6.47	6.48
aspectj-1.6.9	99.29	96.59
castor-1.3.1	46.84	43.64
checkstyle-5.1	15.41	10.02

Table 1. Average time it takes to compile a project. Figures are in seconds.

As can be seen in the results, it appears that the additions of the operators had no effect on other parts of the ExtendJ compiler.

4.2 Applicability

We scanned large open-source projects, see Table 2, to find possible applications for our operators. For spread we found few applications, while we found none at all for print-assign.

The tool we used for scanning was pcregrep. For print-assign, we scanned for uses of `System.out.println`. For spread with arrays only, we scanned for indexed `for`-loops that only invoked a single method call on each element in an array. When we scanned for Iterable, the pattern instead included a `for-each`-loop. For the regular expressions, see appendix A.

For the print-assign operator it was expected to find no uses, since it is intended use was during development and debugging as a quick replacement for `System.out.println`. Based on experience, we believe that when the code has been cleaned up and is ready for production, there are few times when the desired output is a variable name and its `String` representation. However, we didn't consider logging frameworks when we scanned for potential uses. The print-assign operator was not designed to compete with the more thorough loggers: it was designed to be a convenient solution in the absence of a real logger.

We think that the reason why we found few applications for the spread operator is that our implementation is limited to arrays. It is therefore of limited use since collections that implement `Iterable` are almost always preferred over basic arrays. We confirmed this, by also scanning for possible applications of spread on `Iterable` objects. The result is shown in Table 2. The table also includes the total number of `for`-loops used in each project.

Project	Array	Iterable	for-loops
Spring Boot	1	55	370
Elasticsearch	20	233	3291
Guava	2	84	2204
RxJava (ReactiveX)	0	15	601

Table 2. Number of opportunities to use spread operator instead of for-loops in open-source projects. Total amount of for-loops is also included.

5 Related work

There exist other compilers than ExtendJ that can be used for creating language extensions to Java.

One of them is Polyglot [8] which is a source-to-source compiler.

Another extensible compiler framework is ableJ [3]. This framework only implements Java 1.4 though, and there are few extensions, including some for Java 5. Like ExtendJ, ableJ is also built using attribute grammars. However, ableJ is written in an attribute grammar language called Silver [10] instead of JastAdd. There is also a corresponding version for the language C: ableC [6].

Although it would be possible to implement the operators in a non-extensible compiler such as javac, the benefit with using an extensible compiler such as ExtendJ is that all modifications are done in the new modules instead of the original source files. Creating extensions and maintaining them is therefore also comparatively easier.

One way to interact with the compiler in Java is by using predefined annotations or creating annotation processors [9]. By using annotation processors, static analysis and new source files can be created before compile time. Like extensible compilers, annotation processors add extended functionality to the compiler. However, they can't be used to create new operators since they can't modify the language grammar.

6 Future work

One improvement for print-assign would be to add a flag to the compiler such that the printing can be enabled or disabled. This way debug code could be kept in the release and re-enabled without the necessity of recompiling the project.

For our spread operator to behave like Groovy's, there is still a lot of work to do. The two main parts that need to be done is to allow spread of `Iterables`, and to allow expressions consisting of multiple spreads, qualified accesses and method calls.

Since our implementation of spread only applies to arrays at the moment, it will also be necessary to consider the return type when spreading `Iterables`. One intermediate step would be to implement for `Lists` first, and return an `ArrayList` when nothing else is explicitly stated.

Unlike Groovy's spread operator, ours cannot handle expression consisting of multiple spreads, qualified accesses and method calls. For these constructions to work, further work needs to be done with both parsing and bytecode generation.

The Groovy implementation also handles the use of spread on collections where the elements may be `null` by returning `null`. Our implementation instead throws a `NullPointerException`. One possible solution to mimic this behaviour would be to first implement a null-safe operator (like the one in Groovy[5]) and utilize it in the implementation.

7 Conclusion

In this paper we have discussed our implementation of the operators print-assign and simplified spread as an extension to ExtendJ.

We scanned open-source projects for applicability of our new operators and concluded that print-assign has no use in finished code. We also concluded that spread, limited to arrays, had few to no uses in open-source projects' code. However, we also showed that the applicability would have increased if we had allowed spreading of `Iterables`.

Future work can be done for the spread operator by implementing the same functionality as for the spread in Groovy. A compiler switch can also be added for turning on and off printing with print-assign.

Acknowledgments

We'd like to thank Jesper Öqvist, our supervisor and the maintainer of ExtendJ, for helping us with tricky parts of parsing and code generation in ExtendJ, in addition to the feedback on this paper.

References

[1] Hans Bjerndell and Linus Lexfors. 2016. Extending Java with new operators. (2016). http://fileadmin.cs.lth.se/cs/Education/EDAN70/CompilerProjects/2016/Reports/BjerndellLexfors.pdf

[2] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *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*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 1–18. https://doi.org/10.1145/1297027.1297029

[3] Erik Ernst (Ed.). 2007. *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*. Lecture Notes in Computer Science, Vol. 4609. Springer. https://doi.org/10.1007/978-3-540-73589-2

[4] Apache Groovy. 2017. Groovy Documentation. (2017). http://docs.groovy-lang.org/latest/html/documentation/#_spread_operator

[5] Apache Groovy. 2017. Groovy Documentation. (2017). http://groovy-lang.org/operators.html#_safe_navigation_operator

[6] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. 2017. Reliable and automatic composition of language extensions to C: the ableC extensible language framework. *PACMPL* 1, OOPSLA (2017), 98:1–98:29. https://doi.org/10.1145/3138224

[7] Joel Lindholm, Johan Thorsberg, and Görel Hedin. 2016. DrAST: an inspection tool for attributed syntax trees (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering, Amsterdam, The Netherlands, October 31 - November 1, 2016*. 176–180. http://dl.acm.org/citation.cfm?id=2997378

[8] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings (Lecture Notes in Computer Science)*, Görel Hedin (Ed.), Vol. 2622. Springer, 138–152. https://doi.org/10.1007/3-540-36579-6_11

[9] Oracle. 2017. Lesson: Annotations. (2017). https://docs.oracle.com/javase/tutorial/java/annotations/

[10] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Sci. Comput. Program.* 75, 1-2 (2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

Appendix A

Regular expressions used when scanning open-source projects for use of spread operator:

Array:

```
for\s\(int\s([a-z]+\d*)\s=\s\d+;\s?\1\s?[=!<>]{1,2}\s?
(\w+)\.length;\s?((\+\+\1)|(\1\+\+))\)\s\{\n[\s\t]*
[^\+\-\*\/\|\n]*=\s[^\n\(]*\2\[\1\][^\n]*\n[\s\t]*\}
```

Iterable:

```
for\s\([A-Z]\w+\s([a-z]+[0-9]*)\s:\s.*\)\s\{\s?\n
[\t\s]*(.*\s=\s)?\1\.\w+\(.*\)\;.*\n[\t\s]*\}
```

Any for-loop:

```
for\s\(.*\)
```

Appendix B

The source repository for the project can be found at git@bitbucket.org:extendj/spreadpa.git