Weave time macros

Vladimir Oliveira Di Iorio
Univ. Federal de Viçosa
Viçosa - Brazil
vladimir@dpi.ufv.br

Leonardo V. S. Reis
Univ. Federal de Ouro Preto
Ouro Preto - Brazil
leo@decea.ufop.br

Roberto S. Bigonha,
Marco Túlio O. Valente
Univ. Federal de Minas Gerais
Belo Horizonte - Brazil
{bigonha,mtov}@dcc.ufmg.br

ABSTRACT

This ongoing work presents a methodology to extend the pointcut language of AspectJ based on macro definitions. The main features of the proposed approach are: syntax extension in a very flexible way; arguments for new pointcuts are defined by other pointcuts; the semantics of new pointcuts is given by a translation to pure AspectJ, defining precisely the code to be executed at weave time and at runtime. One of the main goals of this methodology is to provide an efficient implementation of the extension mechanism.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Extensible languages

General Terms

Languages

Keywords

Aspect-Oriented Programming, Extensible Languages

1. INTRODUCTION

The expressiveness of the language used to describe pointcuts is one of the most important factors affecting the expressiveness of aspect-oriented languages. Some attempts to define more expressive pointcut languages use functional [7] or logical query languages [8]. Others rely on imperative metaprograms that work on the abstract syntax tree representing the base program [3, 4].

This ongoing work presents a new metaprogramming approach for extending the pointcut language of AspectJ, based on macro definitions. We call it *weave time macros*, for its similarity with standard macros in programming languages. Our approach allows the syntactic extension of any component of AspectJ, including the pointcut language, by extending the language grammar with new production rules. The semantics of the user-defined constructs is established with generative metaprogramming, working on the abstract syntax tree of the source program. Our approach privileges a fine-grained control of program resources, although the complexity may restrict its use to expert programmers. When new pointcuts are defined, it is possible to specify precisely the code that is processed at weave time and at runtime.

This paper is organized as follows. Section 2 shows a simple example that serves as a motivation for the creation of tools that extend the AspectJ pointcut language. Section 3 presents concepts that were combined to define weave time macros. Details of a preliminary implementation are discussed in Section 4. The contribution of this paper is compared with related work in Section 5. Conclusions and future work are presented in Section 6.

2. MOTIVATION

As a motivation for the methodology presented in this paper, we use an example borrowed from [4]. Although it is very simple, it represents a problem that cannot be directly solved using the standard AspectJ pointcut language: an user-defined pointcut that generates weave time and runtime code depending on the joinpoint matched.

Suppose a user wants to define a pointcut `hasType` with two parameters, that works as a predicate satisfied when the type of the first parameter is a subtype of the second parameter. Suppose also that `hasType` is able to explore static type information, identifying situations when the predicate is satisfied or not satisfied without requiring a runtime test. It could be used as follows:

```
...  args(x) && (x hasType T)
```

Using AspectJ and standard pointcut designator syntax, it is possible to create a named pointcut that works in a way similar to the one shown above, implemented with a conditional (`if`) pointcut. But in this case, it is not possible to explore static type information. An AspectJ `if` pointcut always generates a runtime test for every joinpoint matched.

This example is more general than the one presented in [4]. The original problem discusses the implementation of a pointcut that is able to verify only the type of the first parameter of a method. Besides infix syntax, the proposed `hasType` pointcut should be able to analyze the type of any expression, including results from binding variables using pointcuts like `args`, `this` or `target`. Note that the semantics of these pointcuts has to be extended, collecting not only values but also static information about the fragments of code matched by variables.

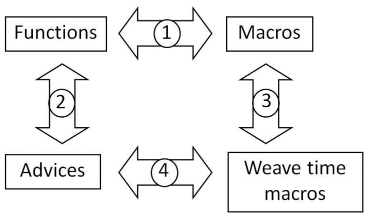

Figure 1: Combining advices and macros.

3. THE PROPOSED APPROACH

We propose that the semantics of user-defined pointcuts may be given by a construct similar to an advice, so that arguments may be collected by AspectJ pointcuts. Like macros, it must treat arguments as program fragments, in order to have full access to the static information of the source program. Combining the concepts of advices and macros, we have coined the term "weave time macros". Figure 1 represents a comparison between functions, advices, macros and weave time macros. The numbers are used to guide the discussion below:

① Functions and macros can be used to extend the functionalities of a program. Macros may go further, allowing syntax extension. In functions, arguments passed as parameters represent values of a program, while in macros, arguments are fragments of code. Macros may define code to be processed in two stages: at macro expansion time and at runtime.

② Functions and advices define parameterized code to be executed at runtime. A function call is defined by the use of the function name, and arguments are passed as expressions. On the other hand, pointcuts define the places where a call to an advice must be inserted, and arguments are defined by special pointcuts that bind values to local variables (in AspectJ, pointcuts like `this`, `target` and `args`).

③ Weave time macros inherit from macros the properties of receiving fragments of code as arguments, and defining code to be executed in two stages: at weave time and at runtime.

④ Weave time macros inherit from advices the property of being executed in places defined by pointcuts. Additionally, they can be parameterized and their arguments can be defined in the same way arguments to advices are defined.

Like standard AspectJ pointcuts, a weave time macro is used together with other pointcuts, which can determine the point in the source code that is affected and are responsible to collect values for the arguments.

4. IMPLEMENTATION

We have developed an implementation using the XAJ language, an extension of AspectJ which allows its own concrete syntax to be modified [5]. In XAJ, *syntax classes* are units that encapsulate the specification of language extensions, offering features for the definition of syntax and semantics,

```
 1 public syntaxclass HasType {
 2
 3   @grammar extends basic_pointcut_expr {
 4     HasType -> "(" expr=expression
 5           "hasType" typeName=name ")"
 6   }
 7
 8   public AST onWeaving(Context ctx) {
 9     Type t = ctx.typeOf(expr);
10     Type t1 = ctx.typeOf(typeName);
11     if (t.subTypeOf(t1)) return '[if (true)];
12     if (! t1.subTypeOf(t)) return '[if (false)];
13     return '[if (#expr instanceof #typeName)];
14   }
15 }
```

Figure 2: A XAJ syntax class.

and also serving as a representation for AST nodes. The semantics is given by a translation to pure AspectJ code, using generative programming, inside a method named *onWeaving*, if the translation occurs at weave time.

Figure 2 shows a XAJ syntax class that extends AspectJ with a `hasType` pointcut designator that works as described in Section 2. In lines 3-6, symbols `basic_pointcut_expr`, `expression` and `name` are nonterminals of an AspectJ grammar. Identifiers `expr` and `typeName` are used as local variables that capture the abstract syntax trees derived from nonterminals `expression` and `name`. The following production is inserted in the grammar:

```
basic_pointcut_expr ->
  "(" expression "hasType" name ")"
```

The method with name "onWeaving" represents a weave time macro, giving the semantics of the new pointcut designator. At weave-time, this method is executed for every join point of the base program. The value returned by `onWeaving` is an abstract syntax tree that replaces the pointcut, each time this method is executed. Expressions of the form `'[...]` represent an abstract syntax tree built using generative programming, written in quasi-quotation syntax. The symbol `#` represents anti-quotation.

If it is possible to statically decide that the type of `expr` is a subtype of `typeName` or cannot be a subtype of `typeName`, the pointcut is replaced by a conditional `if` pointcut that always evaluate to true or false (lines 11 and 12). We rely on optimizations of the AspectJ compiler to delete conditional pointcuts that always evaluate to true or false, generating no residue test.

A first implementation of XAJ has been carried out using the AspectBench Compiler (abc) [2]. In [6], a specific implementation of the `hasType` pointcut designator is described, using the resources of XAJ for the syntax definition and manually modifying the code of the AspectBench compiler to implement the desired semantics. General weave time macros are not implemented in XAJ yet.

5. RELATED WORK

Some works propose the use of a declarative approach to implement extensible pointcut languages. They argue that it makes the pointcut definitions clearer and more flexible. In [8], a very expressive logic-based pointcut language is used, but the base language (ALPHA) is an AO extension of a toy OO core language. In [7], the functional language

XQuery is used as a pointcut language. Our approach is very different from the ones that apply declarative pointcut languages. Weave time macros are defined by generative metaprogramming, using an imperative language (Java).

Josh [4] uses an approach similar to ours. In Josh, the semantics of a user-defined pointcut is given by a static method, whose code is executed at weave time. To insert code to be executed at runtime, a special method must be explicitly invoked. Weave time macros give more flexibility to programmers, allowing user-defined pointcuts to use variables defined by other pointcuts, and allowing the definition of new syntax.

Joinpoint selectors [3] are defined as an extension mechanism for enriching pointcut languages with constructs that play the role of "new primitive pointcuts". They are similar to weave time macros because they can operate either at weave time or at run time. Instead of using metaprogramming, selectors may have a weave time and a runtime version, such that the latter will be called if there is not enough information to decide the selection at weave time.

An important feature of weave time macros and its implementation in XAJ is that AspectJ is the extended language. Because of this, we believe that it may have a larger number of potential users, when completely implemented. On the other hand, Josh is a language inspired in AspectJ, and the current implementation of join point selectors is an extension to the JBoss AOP framework.

SCoPE [1] is another approach that supports the definition of user-defined pointcuts. Similarly to weave time macros, AspectJ is the target language. But instead of extending the AspectJ grammar, analysis-based pointcuts are defined as conditional (`if`) pointcuts, and the compiler of SCoPE generates woven code without runtime tests whenever possible.

6. CONCLUSION AND FUTURE WORK

This paper has presented an ongoing work whose goal is to design and implement a methodology for extension of pointcut languages based on macros. The current implementation is carried out using AspectJ as target language. We believe that choosing the most popular AO language can make the proposed features available for a larger number of potential users.

In order to evaluate the impact of our proposal, we can analyze three main features: the possibility of extending syntax in a flexible way; arguments to weave time macros bound by other pointcuts; use of generative metaprogramming to clearly separate code that is executed at weave time and at runtime. The first feature listed above has already been implemented in the XAJ language. Most works that extend pointcut languages are restricted to the definition of new pointcut designators, without flexible syntax extension. The second feature has been partially implemented, modifying the AspectBench compiler, as shown in [6]. We believe that it represents a clearer way to specify arguments to user-defined pointcuts, when compared to similar works. Finally, the third feature remains unimplemented. We believe that it is also a clearer way to define code that is executed at weave time and at runtime, when compared to similar works. But a drawback of generating arbitrary code at weave time is that it may slow down the compilation process.

When compared to works with a declarative approach, our methodology represents an approach with lower level of abstraction. Defining the semantics of new pointcuts in Java, we expect to develop an efficient implementation. Although restricted to advanced users, defining a new pointcut with weave time macros is evidently much simpler than modifying a real AspectJ compiler, and we hope it can be as efficient as this alternative.

The first implementation of weave time macros, inside the XAJ language, was developed as an extension of the AspectBench compiler. This compiler has not evolved in the last years, so our plans include the development of a completely new implementation with the ajc compiler. It will be important to meet our goal of providing extensibility of the pointcut language to a larger number of users. Our future plans include also investigating the usefulness of generative metaprogramming to define code that is executed at weave time and at runtime, evaluating the performance of compilation and possibly proposing more efficient alternatives.

7. ACKNOWLEDGMENTS

The authors would like to thank Fapemig for funds supporting this work (project APQ-00205-08).

8. REFERENCES

[1] T. Aotani and H. Masuhara. Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 161–172, New York, NY, USA, 2007. ACM.

[2] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[3] C. Breuel and F. Reverbel. User-defined join point selectors – an extension mechanism for pointcut languages. *Journal of Object Technology*, 7(9):5–24, 2008.

[4] S. Chiba and K. Nakagawa. Josh: an open aspectj-like language. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 102–111, New York, NY, USA, 2004. ACM.

[5] V. O. Di Iorio, L. V. d. S. Reis, R. d. S. Bigonha, and M. A. d. S. Bigonha. A proposal for extensible AspectJ. In *DSAL '09: Proceedings of the 4th Workshop on Domain-Specific Aspect Languages*, pages 21–24, New York, NY, USA, 2009. ACM.

[6] V. O. Di Iorio, L. V. d. S. Reis, C. Trevenzoli, and L. E. d. S. Amorim. Implementation of user-defined pointcuts in the XAJ language. In *Proceedings of the IV Latin American Workshop on Aspect-Oriented Software Development*, volume 9, pages 43–48, 2010.

[7] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *APLAS*, pages 366–381, 2004.

[8] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In S. LNCS 3586, editor, *19th European Conference on Object-Oriented Programming (ECOOP)*, 2005.