Generating AspectJ Programs with Meta-AspectJ

David Zook, Shan Shan Huang, Yannis Smaragdakis

College of Computing, Georgia Institute of Technology
Atlanta, GA 30332, USA
{dzook|ssh|yannis}@cc.gatech.edu

Abstract. Meta-AspectJ (MAJ) is a language tool for generating AspectJ programs using code templates. MAJ itself is an extension of Java, so users can interleave arbitrary Java code with AspectJ code templates. MAJ is a structured meta-programming tool: a well-typed generator implies a syntactically correct generated program. MAJ promotes a methodology that combines aspect-oriented and generative programming. Potential applications range from implementing domain-specific languages with AspectJ as a back-end to enhancing AspectJ with more powerful general-purpose constructs. In addition to its practical value, MAJ offers valuable insights to meta-programming tool designers. It is a mature meta-programming tool for AspectJ (and, by extension, Java): a lot of emphasis has been placed on context-sensitive parsing and error-reporting. As a result, MAJ minimizes the number of meta-programming (quote/unquote) operators and uses type inference to reduce the need to remember type names for syntactic entities.

1 Introduction

Meta-programming is the act of writing programs that generate other programs. Powerful meta-programming is essential for approaches to automating software development. In this paper we present Meta-AspectJ (MAJ): a meta-programming language tool extending Java with support for generating AspectJ [9] programs. (AspectJ is a Java extension that allows to separately specify dispersed parts of an application's code and to compose these parts with the main code body.) MAJ offers a convenient syntax, while explicitly representing the syntactic structure of the generated program during the generation process. This allows MAJ to guarantee that a well-typed generator will result in a syntactically correct generated program. This is the hallmark property of *structured* meta-programming tools, as opposed to lexical or text-based tools. Structured meta-programming is desirable because it means that a generator can be released with some confidence that it will create reasonable programs regardless of its inputs.

Why should anyone generate AspectJ programs, however? We believe that combining generative techniques with aspect-oriented programming results in significant advantages compared to using either approach alone. MAJ can be

used for two general kinds of tasks: to implement generators using AspectJ and to implement general-purpose aspect languages using generation. Specifically, MAJ can be used to implement domain-specific languages (i.e., to implement a generator) by translating domain-specific abstractions into AspectJ code. MAJ can also be used to implement general-purpose extensions of AspectJ (e.g., extensions that would recognize different kinds of joinpoints). Thus, MAJ enables the use of AspectJ as an aspect-oriented "assembly language" [13] to simplify what would otherwise be tedious tasks of recognizing patterns in an existing program and rewriting them. A representative of this approach is our prior work on GOTECH [18]: a system that adds distributed capabilities to an existing program by generating AspectJ code using text templates.

The value and novelty of Meta-AspectJ can be described in two axes: its application value (i.e., the big-picture value for potential users) and its technical contributions (i.e., smaller reusable lessons for other researchers working on meta-programming tools). In terms of application value, MAJ is a useful meta-programming tool, not just for AspectJ but also for Java in general. Specifically:

- MAJ is the only tool for structured generation of AspectJ programs that we are aware of. Thus, to combine the benefits of generative programming and AspectJ, one needs to either use MAJ, or to use a text-based approach.
- Compared to plain Java programs that output text, generators written in MAJ are simpler because MAJ allows writing complex code templates using quote/unquote operators.
- Generators written in MAJ are safer than any text-based approach because the syntax of the generated code is represented explicitly in a typed structure.

In terms of technical value, MAJ offers several improvements over prior meta-programming tools for Java. These translate to ease of use for the MAJ user, while the MAJ language design offers insights for meta-programming researchers:

- MAJ shows how to minimize the number of different quote/unquote operators compared to past tools, due to the MAJ mechanism for inferring the syntactic type (e.g., expression, declaration, statement, etc.) of a fragment of generated code. This property requires context-sensitive parsing of quoted code: the type of an unquoted variable dictates how quoted code should be parsed. As a result, the MAJ implementation is quite sophisticated and not just a naive precompiler. A benefit of this approach is that MAJ emits its own error messages, independently from the Java compiler that is used in its back-end.
- When storing fragments of generated code in variables, the user does not need to specify the types of these variables (e.g., whether they are statements, expressions, etc.). Instead, a special `infer` type can be used.

The above points are important because they isolate the user from low-level representation issues and allow meta-programming at the template level.

We next present an introduction to the MAJ language design (Section 2), discuss examples and applications (Section 3), describe in more depth the individual interesting technical points of MAJ (Section 4), and discuss related and future work (Section 5).

2 Meta-AspectJ Introduction

2.1 Background: AspectJ

Aspect-oriented programming (AOP) is a methodology that advocates decomposing software by aspects of functionality. These aspects can be "cross-cutting": they span multiple functional units (functions, classes, modules, etc.) of the software application. Tool support for aspect-oriented programming consists of machinery for specifying such cross-cutting aspects separately from the main application code and subsequently composing them with that code.

AspectJ [9] is a general purpose, high-level, aspect-oriented tool for Java. AspectJ allows the user to define aspects as well as ways that these aspects should be merged (weaved) with the rest of the application code. The power of AspectJ comes from the variety of changes it allows to existing Java code. With AspectJ, the user can add superclasses and interfaces to existing classes and can interpose arbitrary code to method executions, field references, exception throwing, and more. Complex enabling predicates can be used to determine whether code should be interposed at a certain point. Such predicates can include, for instance, information on the identity of the caller and callee, whether a call to a method is made while a call to a certain different method is on the stack, etc. For a simple example of the syntax of AspectJ, consider the code below:

```
aspect CaptureUpdateCallsToA {
  static int num_updates = 0;

  pointcut updates(A a): target(a) && call(public * update*(..));

  after(A a): updates(a) {        // advice
    num_updates++;                // update was just performed
  }
}
```

The above code defines an aspect that just counts the number of calls to methods whose name begins with "update" on objects of type `A`. The "pointcut" definition specifies where the aspect code will tie together with the main application code. The exact code ("advice") will execute after each call to an "update" method.

2.2 MAJ Basics

MAJ offers two variants of code-template operators for creating AspectJ code fragments: `‘[...]` ("quote") and `#[EXPR]` or just `#IDENTIFIER` ("unquote"). (The ellipses, `EXPR` and `IDENTIFIER` are meta-variables matching any syntax, expressions and identifiers, respectively.) The quote operator creates representations of AspectJ code fragments. Parts of these representations can be variable and are designated by the unquote operator (instances of unquote can only occur

inside a quoted code fragment). For example, the value of the MAJ expression `‘[call(* *(..))]` is a data structure that represents the abstract syntax tree for the fragment of AspectJ code `call(* *(..))`. Similarly, the MAJ expression `‘[!within(#className)]` is a quoted pattern with an unquoted part. Its value depends on the value of the variable `className`. If, for instance, `className` holds the identifier “`SomeClass`”, the value of `‘[!within(className)]` is the abstract syntax tree for the expression `!within(SomeClass)`.

MAJ also introduces a new keyword `infer` that can be used in place of a type name when a new variable is being declared and initialized to a quoted expression. For example, we can write:

```
infer pct1 = ‘[call(* *(..))];
```

This declares a variable `pct1` that can be used just like any other program variable. For instance, we can unquote it:

```
infer adv1 = ‘[Object around() : #pct1 { }];
```

This creates the abstract syntax tree for a piece of AspectJ code defining (empty) advice for a pointcut. Section 2.3 describes in more detail the type inference process.

The unquote operator can also be used with an array of expressions. We call this variant of the operator “unquote-splice”. The unquote-splice operator is used for adding arguments in a quoted context that expects a variable number of arguments (i.e., an argument list, a list of methods, or a block of statements). For an example of the unquote-splice operator, if variable `argTypes` holds an array of type names, then we can generate code for a pointcut describing all methods taking arguments of these types as:

```
infer pct2 = ‘[call(* *(#[argTypes]))];
```

That is, if `argTypes` has 3 elements and `argTypes[0]` is `int`, `argTypes[1]` is `String`, and `argTypes[2]` is `Object`, then the value of `pct2` will be the abstract syntax tree for the AspectJ code fragment `call(* *(int, String, Object))`.

We can now see a full MAJ method that generates a trivial but complete AspectJ file:

```
void generateTrivialLogging(Identifier classNm) {
  infer aspectCode =
   ‘[   package MyPackage;
        aspect #[classNm.toString() + "Aspect"] {
          before : call(* #classNm.*(..))
          { System.out.println("Method called"); }
        }
    ];
  System.out.println(aspectCode.unparse());
}
```

The generated aspect causes a message to be printed before every call of a method in a class. The name of the affected class is a parameter passed to the MAJ routine. This code also shows the `unparse` method that our abstract syntax types support for creating a text representation of their code. The abstract syntax types of the MAJ back-end[1] also support other methods for manipulating abstract syntax trees. One such method, `addMember`, is used fairly commonly: `addMember` is supported by syntactic entities that can have an arbitrary number of members (e.g., classes, interfaces, aspects, or argument lists). Although the high-level MAJ operators (quote, unquote, unquote-splice) form a complete set for generating syntax trees, it is sometimes more convenient to manipulate trees directly using the `addMember` method.

2.3 Types and Inference

We saw earlier an example of the MAJ keyword `infer`:

```
infer adv1 = ‘[Object around(): #pct1 {} ];
```

The inferred type of variable `adv1` will be `AdviceDec`. (for "advice declaration"), which is one of the types for AspectJ abstract syntax tree nodes that MAJ defines. Such types can be used explicitly both in variable definitions and in the quote/unquote operators. For instance, the fully qualified version of the `adv1` example would be:

```
AdviceDec adv1 = ‘(AdviceDec)[Object around(): #(Pcd)pct1 {}];
```

The full set of permitted type qualifiers contains the following names: `IDENT`, `Identifier`, `NamePattern`, `Modifiers`, `Import`, `Pcd`, `TypeD`, `VarDec`, `JavaExpr`, `Stmt`, `MethodDec`, `ConstructorDec`, `ClassDec`, `ClassMember`, `InterfaceDec`, `DeclareDec`, `AdviceDec`, `CompilationUnit`, `PointcutDec`, `Pcd`, `AspectDec`, `FormalDec`, and `AspectMember`. Most of the qualifiers' names are self-descriptive, but a few require explanation: `IDENT` is an unqualified name (no dots), while `Identifier` is a full name of a Java identifier, like `pack.clazz.mem`. `NamePattern` can be either an identifier, or a wildcard, or a combination of both. `Pcd` is for a pointcut body (e.g., `call(* *(..))`, as in our example) while `PointcutDec` is for full pointcut definitions (with names and the AspectJ `pointcut` keyword).

Although MAJ allows the use of type qualifiers, these are never necessary for uses of quote/unquote, as well as wherever the `infer` keyword is permitted. The correct types and flavor of the operators can be inferred from the syntax and type information. The goal is to hide the complexity of the explicit type qualifiers from the user as much as possible. Nevertheless, use of types is still necessary wherever the `infer` keyword cannot be used, notably in method signatures and in definitions of member variables that are not initialized.

[1] We currently use modified versions of the AspectJ compiler classes for the MAJ back-end, but may choose to replicate these classes in a separate MAJ package in the future.

3 Applications

There are many ways to view the value of MAJ in the application domain (and, by extension, the value of combining generative and aspect-oriented programming, in general). One can ask why a generator cannot just perform the required modifications to a program without AspectJ, using meta-programming techniques alone. Similarly, one can ask why AspectJ alone is not sufficient for the desired tasks. We address both points below.

3.1 Why Do we Need AspectJ?

Aspect-oriented programming has significant value for building generators. A vidid illustration is our previous work on the GOTECH generator [18]. GOTECH takes a Java program annotated with JavaDoc comments to describe what parts of the functionality should be remotely executable. It then transforms parts of the program so that they execute over a network instead of running on a local machine. The middleware platform used for distributed computing is J2EE (the protocol for Enterprise Java Beans—EJB). GOTECH takes care of generating code adhering to the EJB conventions and makes methods, construction calls, etc. execute on a remote machine. Internally, the modification of the application is performed by generating AspectJ code that transforms existing classes. (We give a specific example later.)

A generator or program transformer acting on Java programs could completely avoid the use of AspectJ and instead manipulate Java syntax directly. Nevertheless, AspectJ gives a very convenient vocabulary for talking about program transformations, as well as a mature implementation of such transformations. AspectJ is a very convenient, higher-level back-end for a generator. It lets its user add arbitrary code in many points of the program, like all references to a member, all calls to a set of methods, etc. If a generator was to reproduce this functionality without AspectJ, the generator would need to parse all the program files, recognize all transformation sites, and apply the rewrites to the syntax. These actions are not simple for a language with the syntactic and semantic complexity of Java. Hence, it is much better to inherit this functionality transparently from a mature tool like AspectJ. For instance, generator writers often need to use functionality similar to the AspectJ `cflow` construct. `cflow` is used for recognizing calls under the control flow of another call (i.e. while the latter is still on the execution stack). Although this functionality can be re-implemented from scratch by adding a run-time flag, this would be a tedious and ad hoc replication of the AspectJ functionality.

Note that using AspectJ as a "bag of program transformation tricks" is an unintended consequence of the power of the system. AspectJ's main intended use is for cross-cutting: the functionality additions should span several classes. The ability to have one aspect affect multiple classes at once is occasionally useful but secondary when the AspectJ code is generated instead of hand-written. On the other hand, in order to support cross-cutting, aspect-oriented tools need to have sophisticated mechanisms for specifying aspect code separately from the

main application code and prescribing precisely how the two are composed. This is the ability that is most valuable to our approach.

3.2 Why Do we Need Meta-Programming?

Despite its capabilities, there are several useful operations that AspectJ alone cannot handle. For example, AspectJ cannot be used to create an interface isomorphic to the public methods of a given class (i.e., a new interface whose methods correspond one-to-one to the public methods of a class). This is an essential action for a tool like GOTECH that needs to create new interfaces (home and remote interfaces, per the EJB conventions) for existing classes. GOTECH was used to automate activities that were previously [15] shown impossible to automate with just AspectJ. GOTECH, however, was implemented using text-based templates. With MAJ, we can do much better in terms of expressiveness and safety as the generated code is represented by a typed data structure instead of arbitrary text.

In general, using meta-programming allows us to go beyond the capabilities of AspectJ by adding arbitrary flexibility in recognizing where aspects should be applied and customizing the weaving of code. For instance, AspectJ does not allow expressing joinpoints based on properties like "all native methods", "all classes with native methods", "all methods in classes that extend a system class", etc. Such properties are, however, simple to express in a regular Java program—e.g., using reflection. Similarly, AspectJ does not allow aspects to be flexible with respect to what superclasses they add to the class they affect, whether added fields are private or public, etc. This information is instead hard-coded in the aspect definition. With a meta-program written in MAJ, the generated aspect can be adapted to the needs of the code body at hand.

3.3 Example

The above points are best illustrated with a small example that shows a task that is easier to perform with AspectJ than with ad hoc program transformation, but cannot be performed by AspectJ alone. Consider the MAJ code in Figure 1. This is a complete MAJ program that takes a class as input, traverses all its methods, and creates an aspect that makes each method argument type implement the interface `java.io.Serializable` (provided the argument type does not implement this interface already and it is not a primitive type). For example, imagine that the class passed to the code of Figure 1 is:

```
class SomeClass {
  public void meth1(Car c) { ... }
  public void meth2(int i, Tire t) { ... }
  public void meth3(float f, Seat s) { ... }
}
```

In this case, the list of all argument types is `int`, `float`, `Car`, `Tire`, and `Seat`. The first two are primitive types, thus the MAJ program will generate the following AspectJ code:

```
import java.io.*;
import java.lang.reflect.*;
import org.aspectj.compiler.base.ast.*;
import org.aspectj.compiler.crosscuts.ast.*;

public class MAJGenerate {
  public static void genSerializableAspect(Class inClass, PrintStream out)
  {
    // Create a new aspect
    infer serializedAspect = `[aspect SerializableAspect {}];

    // Add Serializable to every method argument type that needs it
    for (int meth = 0; meth < inClass.getMethods().length; meth++) {
      Class[] methSignature =
        inClass.getMethods()[meth].getParameterTypes();
      for (int parm = 0; parm < methSignature.length; parm++) {
        if (!methSignature[parm].isPrimitive() &&
            !Serializable.class.isAssignableFrom(methSignature[parm]))
          serializedAspect.addMember(`[ declare parents:
                                          #[methSignature[parm].getName()]
                                        implements java.io.Serializable;
                                      ]
                                    );
      } // for all params
    } // for all methods

    infer compU = `[ package gotech.extensions;
                       #serializedAspect
                   ];

    out.print(compU.unparse());
  }
}
```

Fig. 1. A routine that generates an aspect that makes method parameter types be serializable

```
package gotech.extensions;
aspect SerializableAspect {
  declare parents: Car implements java.io.Serializable;
  declare parents: Tire implements java.io.Serializable;
  declare parents: Seat implements java.io.Serializable;
}
```

The code of Figure 1 faithfully replicates the functionality of a template used in GOTECH: the system needs to make argument types be serializable if a method is to be called remotely. (We have similarly replicated the entire functionality of GOTECH using MAJ and have used it as a regression test during the development of MAJ.)

This example is a good representative of realistic uses of MAJ, in that the cases where AspectJ alone is not sufficient are exactly those where complex conditions determine the existence, structure, or functionality of an aspect. Observe that most of the above code concerns the application logic for finding argument types and deciding whether a certain type should be augmented to implement interface `Serializable`. MAJ makes the rest of the code be straightforward.

The above example is self-contained, but it is worth pointing out that it could be simplified by making use of reusable methods to traverse a class or a method signature. Java allows a more functional style of programming through the use of interfaces and anonymous classes. These let us write general iteration methods like `forAllMethods` or `forAllArguments` that could have been used in the above code.

4 Meta-AspectJ Design and Implementation

4.1 MAJ Design

We will next examine the MAJ design a little closer, in order to compare it to other meta-programming tools.

Structured vs. Unstructured Meta-Programming The value of a MAJ quoted code fragment is an abstract syntax tree for the code fragment. The MAJ operators ensure that all trees manipulated by a MAJ program are syntactically well-formed, although they may contain semantic errors, such as type errors or scoping errors (e.g., references to undeclared variables). That is, MAJ is based on a context-free grammar for describing AspectJ syntax. The MAJ expressions created using the quote operator correspond to words ("words" in the formal languages sense) produced by different non-terminals of this context-free grammar. Compositions of abstract syntax trees in ways that are not allowed by the grammar is prohibited. Thus, using the MAJ operators, one cannot create trees that do not correspond to fragments of AspectJ syntax. For instance, there is no way to create a tree for an "if" statement with 5 operands (instead of 3, for the condition, then-branch, and else-branch), or a class with a statement

as its member (instead of just methods and instance variable declarations), or a declaration with an operator in the type position, etc. The syntactic well-formedness of abstract syntax trees is ensured statically when the MAJ program is compiled. For example, suppose the user wrote a MAJ program containing the declarations:

```
infer pct1 = `[call(* *(..))];
infer progr = `[ package MyPackage;
                   #pct1
                 ];
```

The syntax error (pointcut in unexpected location) would be caught when this program would be compiled with MAJ.

The static enforcement of syntactic correctness for the generated program is a common and desirable property in meta-programming tools. It is often described as "the type safety of the generator implies the syntactic correctness of the generated program". The property is desirable because it increases confidence in the correctness of the generator under all inputs (and not just the inputs with which the generator writer has tested the generator). This property is the hallmark of *structured* meta-programming tools—to be contrasted with unstructured, or "text-based", tools (e.g., the C pre-processor or the XDoclet tool [16] for Java).

As a structured meta-programming tool, MAJ is superior to text-based tools in terms of safety. Additionally, however, it is also superior to text-based meta-programming in terms of convenience or expressiveness. Compared to text-based generation with a tool like XDoclet [16], MAJ programs can use any Java code to determine what should be generated, instead of being limited to a hard-coded set of attributes and annotations. Compared to text-based generation with plain Java strings, MAJ is much simpler. Instead of putting Java strings together with the "+" operator (and having to deal with low-level issues like explicitly handling quotes, new lines, etc.) MAJ lets the user use convenient code templates.

Of course, static type safety implies that some legal programs will not be expressible in MAJ. For instance, we restrict the ways in which trees can be composed (i.e., what can be unquoted in a quote expression and how). The well-formedness of an abstract syntax tree should be statically verifiable from the types of its component parts—if an unquoted expression does not have the right type, the code will not compile even if the run-time value happens to be legal. Specifically, it is not possible to have a single expression take values of two different abstract syntax tree types. For example we cannot create an abstract syntax tree that may hold either a definition or a statement and in the cases that it holds a definition use it in the body of a class (where a statement would be illegal).

Qualifier Inference MAJ is distinguished from other meta-programming tools because of its ability to infer qualifiers for the quote/unquote operators, as well

as the ability to infer types for the variables initialized by quoted fragments. Having multiple quote/unquote operators is the norm in meta-programming tools for languages with rich surface syntax (e.g., meta-programming tools for Java [3], C [20], and C++ [7]). For instance, let us examine the JTS tool for Java meta-programming—the closest comparable to MAJ. JTS introduces several different kinds of quote/unquote operators: `exp{...}exp`, `$exp(...)`, `stm{...}stm`, `$stm(...)`, `mth{...}mth`, `$mth(...)`, `cls{...}cls`, `$cls(...)`, etc. Additionally, just like in MAJ, JTS has distinct types for each abstract syntax tree form: `AST_Exp`, `AST_Stmt`, `AST_FieldDecl`, `AST_Class`, etc. Unlike MAJ, however, the JTS user needs to always specify explicitly the correct operator and tree type for all generated code fragments. For instance, consider the JTS fragment:

```
AST_Exp x = exp{ 7 + i }exp;
AST_Stm s = stm{ if (i > 0) return $exp(x); }stm;
```

This is written in MAJ as just:

```
infer x = `[7 + i];
infer s = `[if (i > 0) return #x];
```

The advantage is that the user does not need to tediously specify what flavor of the operator is used at every point and what is the type of the result. MAJ will instead infer this information. As we explain next, this requires sophistication in the parsing implementation.

4.2 MAJ Implementation

We have invested significant effort in making MAJ a mature and user-friendly tool, as opposed to a naive pre-processor. This section describes our implementation in detail.

Qualifier Inference and Type System It is important to realize that although multiple flavors of quote/unquote operators are common in syntax-rich languages, the reason for their introduction is purely technical. There is no fundamental ambiguity that would occur if only a single quote/unquote operator was employed. Nevertheless, inferring qualifiers, as in MAJ, requires the meta-programming tool to have a full-fledged compiler instead of a naive pre-processor. (An alternative would be for the meta-programming tool to severely limit the possible places where a quote or unquote can occur in order to avoid ambiguities. No tool we are aware of follows this approach.) Not only does the tool need to implement its own type system, but also parsing becomes context-sensitive—i.e., the type of a variable determines how a certain piece of syntax is parsed, which puts the parsing task beyond the capabilities of a (context-free) grammar-based parser generator.

To see the above points, consider the MAJ code fragment:

```
infer l = `[ #foo class A {} ];
```

The inferred type of `l` depends on the type of `foo`. For instance, if `foo` is of type `Modifiers` (e.g., it has the value `‘[public]`) then the above code would be equivalent to:

```
ClassDec l = ‘[ #(Modifiers)foo class A {} ];
```

If, however, `foo` is of type `Import` (e.g., it has the value `‘[import java.io.*;]`) then the above code would be equivalent to:

```
CompilationUnit l = ‘[ #(Import)foo class A {} ];
```

Thus, to be able to infer the type of the quoted expression we need to know the types of the unquoted expressions. This is possible because MAJ maintains its own type system (i.e., it maintains type contexts for variables during parsing). The type system is monomorphic and quite straightforward: when deriving the type of an expression, the types of its component subexpressions are known and there is a most specific type for each expression. No recursion is possible, since the `infer` keyword can only be used in variable declarations and the use of a variable in its own initialization expression is not allowed in Java.

The types of expressions even influence the parsing and translation of quoted code. Consider again the above example. The two possible abstract syntax trees are not even isomorphic. If the type of `foo` is `Modifiers`, this will result in an entirely different parse and translation of the quoted code than if the type of `foo` is `Import` (or `ClassDec`, or `InterfaceDec`, etc). In the former case, `foo` just describes a modifier—i.e., a branch of the abstract syntax tree for the definition of class `A`. In the latter case, the abstract syntax tree value of `foo` is at the same level as the tree for the class definition.

Parser Implementation The current implementation of the MAJ front-end consists of one common lexer and two separate parsers. The common lexer recoginizes tokens legal in both the meta-language (Java), and the object language (AspectJ). This is not a difficult task for this particular combination of meta/object languages, since Java is a subset of AspectJ. For two languages whose token sets do not match up as nicely, a more sophisticated scheme would have to be employed.

We use ANTLR [11] to generate our parser from two separate LL(k) grammars (augmented for context-sensitivity, as described below). One is the Java' grammar: Java with additional rules for handling quote and `infer`. The other is the AspectJ' grammar: AspectJ with additional rules for handling `infer`, quote and unquote. Java', upon seeing a quote operator lifts out the string between the quote delimiters (`‘[...]`) and passes it to AspectJ' for parsing. AspectJ', upon seeing an unquote, lifts out the string between the unquote delimiters and passes it to Java' for parsing. Thus, we are able to completely isolate the two grammars. This paves way for easily changing the meta or object language for future work, with the lexer caveat previously mentioned.

The heavy lifting of recognizing and type-checking quoted AspectJ is done in AspectJ'. To implement context-sensitive parsing we rely on ANTLR's facilities

for guessing as well as adding arbitrary predicates to grammar productions and backtracking if the predicates turn out to be false. Each quote entry point production is preceded by the same production wrapped in a guessing/backtracking rule. If a phrase successfully parses in the guessing mode and the predicate (which is based on the types of the parsed expressions) succeeds, then real parsing takes place and token consumption is finalized. Otherwise, the parser rewinds and attempts parsing by the next rule that applies. Thus, a phrase that begins with a series of unquoted entities might have to be guess-parsed by a number of alternate rules before it reaches a rule that actually applies to it.

The parsing time of this approach depends on how many rules are tried unsuccessfully before the matching one is found. In the worst case, our parsing time is exponential in the nesting depth of quotes and unquotes. Nevertheless, we have not found speed to be a problem in MAJ parsing. The parsing time is negligible even for code templates with a nesting depth of 5 (i.e., a quote that contains an unquote that contains a quote that contains an unquote, and so on, for 5 total quotes and 5 unquotes). Of course, more sophisticated parsing technology can be used in future versions, but arguably parsing speed is not a huge constraint in modern systems and we place a premium on using mature tools like ANTLR and expressing our parsing logic declaratively using predicates.

Translation The MAJ compiler translates its input into plain Java code. This is a standard approach in meta-programming tools [3, 4, 19]. For example, consider the trivial MAJ code fragment:

```
infer dummyAspect = ‘[ aspect myAspect { } ];
infer dummyUnit = ‘[ package myPackage;
                       #dummyAspect
                    ];
```

MAJ compilation will translate this fragment to Java code using a library for representing AspectJ abstract syntax trees. The Java compiler is then called to produce bytecode. The above MAJ code fragment will generate Java code like:

```
AspectDec dummyAspect =
  new AspectDec(
    null, "myAspect", null, null, null,
    new AspectMembers(new AspectMember[] {null}));
CompilationUnit dummyUnit =
  new MajCompilationUnit(
   new MajPackageExpr(new Identifier("myPackage")),
   null, new Decs(new Dec[] { dummyAspect }));
```

The Java type system could also catch ill-formed MAJ programs. If the unquoted expression in the above program had been illegal in the particular syntactic location, the error would have exhibit itself as a Java type error. Nevertheless, MAJ implements its own type system and performs error checking before translating its input to Java. Therefore, the produced code will never contain

MAJ type errors. (Nevertheless, there are Java static errors that MAJ does not currently catch, like access protection errors, uninitialized variable errors, and more.)

Error Handling Systems like JTS [3] operate as simple pre-processors and delegate the type checking of meta-programs to their target language (e.g., Java). The disadvantage of this approach is that error messages are reported on the generated code, which the user has never seen. Since MAJ maintains its own type system, we can emit more accurate and informative error messages than those that would be produced for MAJ errors by the Java compiler.

Recall that our parsing approach stretches the capabilities of ANTLR to perform context-sensitive parsing based on the MAJ type system. To achieve good error reporting we had to implement a mechanism that distinguishes between parsing errors due to a mistyped unquoted entity and regular syntax errors. While attempting different rule alternatives during parsing, we collect all error messages for failed rules. If parsing succeeds according to some rule, the error information from failed rules is discarded. If all rules fail, we re-parse the expression with MAJ type checking turned off. If this succeeds, then the error is a MAJ type error and we report it to the user as such. Otherwise, the error is a genuine syntax error and we report to the user the reasons that caused each alternative to fail.

Implementation Evolution and Advice It is worth briefly mentioning the evolution of the implementation of MAJ because we believe it offers lessons for other developers. The original MAJ implementation was much less sophisticated than the current one. The system was a pre-processor without its own type system and relied on the Java compiler for ensuring the well-formedness of MAJ code. Nevertheless, even at that level of sophistication, we found that it is possible to make the system user-friendlier with very primitive mechanisms.

An important observation is that type qualifier inference can be performed even without maintaining a type system, as long as ambiguous uses of the unquote operator are explicitly qualified. That is, qualifying some of the uses of unquote allows having a single unqualified quote operator and the `infer` keyword. For instance, consider the code fragment:

```
infer s = ‘[if (i > 0) return #x];
```

The syntactic types of `s` and `x` are clear from context in this case. Even the early implementation of MAJ, without its own type system, could support the above example. Nevertheless, in cases where parsing or type inference would be truly dependent on type information, earlier versions of MAJ required explicit qualification of unquotes—for instance:

```
infer l = ‘[ #(Modifiers)foo class A {} ];
```

Even with that early approach, however, parsing required unlimited lookahead, making our grammar not LL(k).

5 Related Work and Future Work Directions

In this section we connect MAJ to other work in AOP and meta-programming. The comparison helps outline promising directions for further research. We will be selective in our references and only pick representative and/or recent work instead of trying to be exhaustive.

In terms of philosophy, MAJ is a compatible approach to that of XAspects [13], which advocates the use of AspectJ as a back-end language for aspect-orientation. Aspect languages in the XAspects framework can produce AspectJ code using MAJ.

It is interesting to compare MAJ to state-of-the-art work in meta-programming. Visser [19] has made similar observations to ours with respect to concrete syntax (i.e., quote/unquote operators) and its introduction to meta-languages. His approach tries to be language-independent and relies on generalized-LR parsing (GLR). We could use GLR technology for MAJ parsing in the future. GLR is powerful for ambiguous grammars as it returns all possible parse trees. Our type system can then be used to disambiguate. Although parsing technology is an interesting topic, we do not consider it crucial for MAJ. Our current approach with ANTLR is perhaps crude but yields a clean specification and quite acceptable performance.

Multi-stage programming languages, like MetaML [17] and MetaOCaml [6] represent state-of-the-art meta-programming systems with excellent safety properties. For instance, a common guarantee of multi-stage languages is that the type safety of the generator implies the type safety of the generated program. This is a much stronger guarantee than that offered by MAJ and other abstract-syntax-tree-based tools. It means that the meta-programming infrastructure needs to keep track of the contexts (declared variables and types) of the *generated program*, even though this program has not yet been generated. Therefore the system is more restrictive in how the static structure of the generator reflects the structure of the generated program. By implication, some useful and legal generators may be harder to express in a language like MetaOCaml. In fact, although there is great value in multi-stage programming approaches, we are not yet convinced that they are appropriate for large-scale, ad hoc generator development. Current applications of multi-stage languages have been in the area of directing the specialization of existing programs for optimization purposes.

An interesting direction in meta-programming tools is that pioneered by *hygienic macro expansion* [10, 8]. Hygienic macros avoid the problem of unintended capture due to the scoping rules of a programming language. For instance, a macro can introduce code that inadvertently refers to a variable in the generation site instead of the variable the macro programmer intended. Similarly, a macro can introduce a declaration (binding instance) that accidentally binds identifier references from the user program. The same problem exists in programmatic meta-programming systems, like MAJ. In past work, we described *generation scoping* [14]: a facility for controlling scoping environments of the generated program, during the generation process. Generation scoping is the analogue of hygienic macros in the programmatic (not pattern-based, like macros) meta-

programming world. Generation scoping does not offer any guarantees about the correctness of the target program, but gives the user much better control over the lexical environments of the generated program so that inadvertent capture can be very easily avoided. Adding a generation scoping facility to MAJ is an interesting future work direction.

Other interesting recent tools for meta-programming include Template Haskell [12]—a mechanism for performing compile-time computations and syntax transformation in the Haskell language. Closer in the language domain to MAJ are tools, like JSE [1], ELIDE [5] and Maya [2] that have been proposed for manipulating Java syntax. Most of these tools aim at syntactic extensions to the language and are closely modeled after macro systems. The MAJ approach is different both in that it targets AspectJ, and in that it serves a different purpose: it is a tool for programmatic meta-programming, as opposed to pattern-based meta-programming. In MAJ, we chose to separate the problem of recognizing syntax (i.e., pattern matching and syntactic extension) from the problem of generating syntax (i.e., quoting/unquoting). MAJ only addresses the issues of generating AspectJ programs using simple mechanisms and a convenient language design. Although meta-programming is a natural way to implement language extensions, uses of MAJ do not have to be tied to syntactic extensions at all. MAJ can, for instance, be used as part of a tool that performs transformations on arbitrary programs based on GUI input or program analysis. MAJ also has many individual technical differences from tools like JSE, ELIDE, and Maya (e.g., JSE is partly an unstructured meta-programming tool, Maya does not have an explicit quote operator, etc.).

Complementing MAJ with a facility for *recognizing* syntax (i.e., performing pattern matching on abstract syntax trees or based on semantic properties) is a straightforward direction for future work. Nevertheless, note that the need for pattern matching is not as intensive for MAJ as it is for other meta-programming tools. The reason is that MAJ generates AspectJ code that is responsible for deciding what transformations need to be applied and where. The beauty of the combination of generative and aspect-oriented programming is exactly in the simplicity of the generative part afforded by the use of aspect-oriented techniques. Another promising direction for future work on MAJ is to make it an extension of AspectJ, as opposed to Java (i.e., to allow aspects to generate other aspects). We do not yet have a strong motivating example for this application, but we expect it may have value in the future.

6 Conclusions

In this paper we presented Meta-AspectJ (MAJ): a tool for generating AspectJ programs. The implementation of MAJ was largely motivated by practical concerns: although a lot of research has been performed on meta-programming tools, we found no mature tool, readily available for practical meta-programming tasks in either Java or AspectJ. MAJ strives for convenience in meta-programming but does not aspire to be a heavyweight meta-programming infrastructure that

supports syntactic extension, pattern matching, etc. Instead, MAJ is based on the philosophy that generative tools have a lot to gain by generating AspectJ code and delegating many issues of semantic matching to AspectJ. Of course, this approach limits the ability for program transformation to the manipulations that AspectJ supports. Nevertheless, this is exactly why our approach is an aspect-oriented/generative hybrid. We believe that AspectJ is expressive enough to capture many useful program manipulations at exactly the right level of abstraction. When this power needs to be combined with more configurability, generative programming can add the missing elements. We hope that MAJ will prove to be a useful tool in practice and that it will form the basis for several interesting domain-specific mechanisms.

References

1. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proceedings of the OOPSLA '01 conference on Object Oriented Programming Systems Languages and Applications*, pages 31–42. ACM Press, 2001.
2. J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 270–281. ACM Press, 2002.
3. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proceedings Fifth International Conference on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
4. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, pages 187–197. IEEE Computer Society, 2003.
5. A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the 1st international conference on Aspect-Oriented Software Development*, pages 10–18. ACM Press, 2002.
6. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2830, pages 57–76. Springer, 2003.
7. S. Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, Oct. 1995.
8. W. Clinger. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 155–162. ACM Press, 1991.
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.
10. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM Press, 1986.
11. T. J. Parr and R. W. Quong. ANTLR: A predicated LL(k) parser generator. *Software, Practice and Experience*, 25(7):789–810, July 1995.
12. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM Press, 2002.

13. M. Shonle, K. Lieberherr, and A. Shah. Xaspects: An extensible system for domain specific aspect languages. In *OOPSLA '2003, Domain-Driven Development Track*, October 2003.
14. Y. Smaragdakis and D. Batory. Scoping constructs for program generators. In *Generative and Component-Based Software Engineering Symposium (GCSE)*, 1999. Earlier version in Technical Report UTCS-TR-96-37.
15. S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM Press, 2002.
16. A. Stevens et al. *XDoclet Web site, http://xdoclet.sourceforge.net/.*
17. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 203–217. New York: ACM, 1997.
18. E. Tilevich, S. Urbanski, Y. Smaragdakis, and M. Fleury. Aspectizing server-side distribution. In *Proceedings of the Automated Software Engineering (ASE) Conference*. IEEE Press, October 2003.
19. E. Visser. Meta-programming with concrete object syntax. In *Generative Programming and Component Engineering (GPCE) Conference*, LNCS 2487, pages 299–315. Springer, 2002.
20. D. Weise and R. F. Crew. Programmable syntax macros. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 156–165, 1993.