Aspect-Oriented Programming using Reflection

Gregory T. Sullivan
Artificial Intelligence Laboratory, Massachusetts Institute of Technology
gregs@ai.mit.edu   http://www.ai.mit.edu/~gregs

August 3, 2001

[For OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Oct. 14-18, 2001, Tampa Bay, Florida, USA]

Abstract

What infrastructure is needed to support aspect-oriented software development?

To the programmer, we must supply aspect-oriented (sub)languages that are based on the constructs and basic syntax that the programmer is most familiar with, as well as facilities for reifying and manipulating the cross-cutting features of immediate concern.

At the implementation level, the aspect specifications must insinuate themselves into the application proper. Furthermore, the support for aspect specification and integration should not impose a performance penalty on the application at runtime.

We are conducting research into dynamic reflective object-oriented language design and implementation that provides a rich infrastructure for the development of aspect-oriented technology. The main concepts that we bring to bear are:

Reflection and Dynamism: A program is able to reflect on its structure and behavior, and then modify its behavior. *metaobject protocols* rely on reflection to present a programmer-modifiable dynamic semantics.

Syntactic abstraction: A procedural, hygienic macro system facilitates implementation of AOP language extensions;

Dynamic, Optimistic Optimization: Programs do not pay a performance penalty merely for the existence of a metaobject protocol. That is, if part of an application does not make use of the abstraction provided by a metaobject protocol, performance is as good as if the metaobject protocol did not exist.

1 Introduction

Some of the original inspiration for Aspect-Oriented Programming (AOP) [KLM$^+$97] draws from the research in dynamic, reflective object-oriented languages and metaobject protocols (MOPs) [KdB91][1]. Some of the features typical of a MOP include programmatic control of dynamic dispatch and subclassing of metaobjects such as classes and methods. These powerful facilities enable the sort of crosscutting metaprogramming that AOP strives to deliver. We believe that there are several reasons that research on providing aspect-oriented programming features to the programmer has strayed from its metaobject protocol roots:

Too much rope: metaobject protocols, while elegant, are complicated. Providing full control over the implementation of a language's runtime may be placing too much power and complexity in the hands of the programmer.

Too much overhead: It is generally assumed that the presence of a runtime MOP has too negative an impact on performance to be worthwhile.

Our research addresses both of these concerns. While a full runtime metaobject protocol may be available, the aspect-oriented language designer may choose to expose only some of the features. Furthermore, with sophisticated static and dynamic optimizations, we can eliminate the overhead of a MOP, except where the features of the MOP are actually used. For example, if a dynamic call takes place in a loop, the fully general dynamic dispatch protocol defined by the MOP will be optimized away, guarded by some fast checks outside of the loop. If, during the call, some assumptions are violated – for example, a new method is added to a virtual function – then the guard will fail and dispatch will revert for the time being to the general case.

2 Reflection, Dynamism, and Metaobject Protocols

Computational reflection [SMI84, Mae87] enables a program to access its internal structure and behavior and also to programmatically manipulate that structure, thereby modifying its behavior. The process of getting access to reflective data is often called "reification". Java [GJSB00] provides some reflection capability. For example, a Java program can ask for the class of a given object, find the methods on that class, and then invoke one of those methods. The `Class` and `Method` classes of Java are often referred to as "metaclasses" and their instances "metaobjects". A metaobject protocol defines execution of an application in terms of behaviors implemented by metaclasses. For example, dynamic method dispatch may be defined as first finding the methods for the virtual function, then finding the classes of receiver and other arguments, and finally choosing the most applicable method.

Java's reflection capabilities fall short of a full metaobject protocol in two important respects:

1. Java's reflection is "read-only". For example, a program can query the methods of a class, but a program cannot dynamically *change* the methods of a class. A full MOP allows modification of any meta-information that can be reified.
2. Java does not allow subclassing of metaclasses such as `Class` and `Method`. With a full MOP, subclassing metaclasses is a way to incrementally change the default behavior of a language.

[1] The astute reader will notice that many of the same personalities are involved in MOP's and AOP.

Using the terminology of [KdB91], Java provides *introspection* but not *intercession*. Java does provide some dynamism with the fairly heavyweight mechanism of dynamic class loading. Other mainstream programming languages, such as C++, provide even less in the way of computational reflection.

3 Static Specialization of Application Methods using Predicate Types

Most of our examples involve adding specialized methods to virtual functions. In order to have the method dispatch mechanism do as much of the work as possible, we have an extended type system, inspired by predicate dispatching, as introduced in [EKC98]. In addition to the class specializers allowed in typical object-oriented languages, we have an extended language of specializer types:

```
spec ::= class | eq(value-exp) | and(spec, spec) | or(spec, spec)
       | pred(x){ exp* }
```

where the base case `pred(x){ exp* }` allows for arbitrary boolean-returning functions to be used as argument specializers for the purpose of dispatch. Implication in this type system first includes subtyping between classes, and then uses logical implication such as `and(p1, p2)` implies `p1` (as well as `p2`). Implication is used to determine the most applicable method for a given call at runtime.

As a running example, suppose we have a graphics application with an abstract `Drawable` classes with concrete subclasses `Circle`, `Square`, and `Line`. There is also a `Canvas` abstract class with concrete subclasses `2DCanvas` and `3DCanvas`. The `Drawable` class has an abstract virtual function `draw(Canvas c)`.

With our extended type system, we immediately gain some of the functionality associated with aspect-oriented programming. For example, if we want to run some additional code when `draw` is invoked on a `Circle` object that satisfies some predicate `pred`, we can write:

```
void draw(and(pred, Circle) this, Canvas c) {
  ... additional code ...
  callNextMethod(); }
```

The dynamic dispatch mechanism will take care of selecting this new method when called on a `Circle` that satisfies `pred`, and also of chaining to the next-most-specific method using `callNextMethod()`.

4 Runtime Specialization of Application Functions

One of the important capabilities provided by reflection is the ability to add methods at runtime. Combined with the extended types presented above, we gain even more powerful abstraction capabilities.

Suppose there is an instance of the `Canvas` class bound to `aCanvas`, and we want to instrument the drawing of circles on `aCanvas` with some additional code. We can write:

```
addMethod(draw(Circle d, eq(aCanvas) c) {
            ... additional code ...
            callNextMethod(); });
```

The specializer `eq(aCanvas)` is a type that matches exactly one object, namely `aCanvas` – this mimics `eql` types in CLOS. Dynamic dispatch will invoke the new `draw` method when called on a circle and the particular canvas `aCanvas`.

The interesting things about this example are that (1) the addition of the new method can be done conditionally, based on runtime values, and (2) the method is specialized according to a runtime value (`aCanvas`). Both of these features are more difficult to implement with a more static approach.

5 Specializing Metaobject Protocols

For an even greater level of abstraction and crosscutting potential, we may make use of a runtime metaobject protocol.

Assume we have a metaobject protocol that implements method dispatch using the following simplified `dispatch` method:

```
Object dispatch(VirtualMethod fun, Value[] argVals) {
  Method[] funMeths = fun.methods();
  Method[] appMeths = collectApplicable(funMeths, argVals);
  Method[] appMethsSorted = methodSort(fun, appMeths, argVals);
  methodApply(appMethsSorted[0], argVals, appMethsSorted);
}
```

Consider a call `drawable.draw(canvas)` when `drawable` is an instance of `Circle` and `canvas` is an instance of `2DCanvas`. Suppose there are `Draw` methods defined for the following `(receiver, argument)` signatures: (`Object, Canvas`), (`Object, 2DCanvas`), (`Square, 2DCanvas`), and (`Circle, 2DCanvas`).

The `dispatch` function will first determine that the applicable methods are the ones with signatures (`Object, Canvas`), (`Object, 2DCanvas`),and (`Circle, 2DCanvas`). Then `methodSort` returns the list {(`Circle`, `2DCanvas`), (`Object`, `2DCanvas`), and (`Object`, `Canvas`)}. Finally, the method on (`Circle`, `2DCanvas`) is chosen and dispatched to.

So what? The end result is the same as in any object-oriented language. The key idea of metaobject protocols is that:

> *Every operation defined in the metaobject protocol can be re(de)fined by the programmer.*

Suppose we want to execute some advice before every method invoked with a `Circle` object as the receiver. We can specialize the `methodApply` method as follows:

```
Object methodApply(Method m, Value[Circle val0, ...] argVals, appmeths) {
   ... advice code ...
   callNextMethod();
```

```
}
```

The specialization of `methodApply` matches a non-empty array where the first element (the receiver) is a `Circle` instance. After executing the advice code, we use the `callNextMethod` function to chain to the next most applicable version of `methodApply` – the version (likely the default implementation) that would have been invoked before we added the specialized version of `methodApply`.

This basic mechanism of specializing predefined behaviors of the metaobject protocol enables many interesting "aspectual" definitions. For example, consider specializing `methodSort` to use profile information to choose between alternate implementations of a method; or consider specializing `collectApplicable` to search the network to find methods matching a given signature. The possibilities are endless.

5.1 Dynamic versus Static "Weaving"

Metaobject protocols enable us to expose structural and behavioral aspects of a running application to programmer control and modification; but so do more static tools such as (the current implementation of) AspectJ. What have we gained by introducing a runtime metaobject protocol? In addition to an aesthetic argument, we immediately gain the following:

1. **Robustness** in the face of dynamic class loading or other kind of dynamic modification to the program structure. Even Java, with its limited dynamism, enables dynamic class loading, which may foil any approach that relies on static weaving of aspect code with base code. For applications which support new code arriving and being incorporated into the running application, a dynamic approach is required. Furthermore, Java's dynamic class reloading does not affect the behavior of existing instances of the class, and sometimes we need that functionality.
2. **Dynamic** analysis and control of aspects. If the "parameters" of an aspect depend on runtime values, a MOP-based approach is preferable. For example, suppose we have an aspect that wraps existing classes for distribution to remote machines, but we will only know which classes we want to distribute at runtime, based on system load, number of objects created, and so forth. While such abstraction over runtime values is possible statically, it is more natural to have the actual instrumentation of the classes take place at runtime, after the classes to be distributed are identified.

Other researchers [Böl99] have explored so-called "dynamic weaving", noting that dynamic control of aspects is often very useful. The focus seems to be on mimicking the text-based weaving process at runtime, rather than the more purely reflection-based approach we have taken.

6 Syntactic Abstraction to Support Aspect-oriented Language Extension

As mentioned earlier, reflection and metaobject protocols are elegant and powerful tools, but we are not claiming that the best way to supply aspect-oriented technology to a programmer is by simply exposing metaclasses and their methods. Instead, we rely on powerful syntactic abstraction facilities that may translate into use of the underlying metaobject protocol.

Jonathan Bachrach and Keith Playford have developed the *Java Syntactic Extender*(JSE) [BP01], presented in this year's OOPSLA technical program. As an example of the use of JSE to implement aspects, suppose we want to be able to log writes to some variables. The programmer should be able to write, for example,

```
logged Square s = new Square(length, originPt);
```

prefacing the declaration of variables with the `logged` keyword.

In fact, `logged` is a macro that expands, in the above case, to

```
Square s = new Square(length, originPt);
addMethod(set-s!(eq(s), newval) {
             ... log set to s ...
             callNextMethod(); } );
```

The ability to correctly use reflective features such as `addMethod` requires an intricate knowledge of the runtime (such as that field mutators are implemented as virtual functions named `set-field!`), but the use of the `logged` keyword is much more straightforward.

7 Implementation Techniques for Metaobject Protocols

The programming languages research community is starting to embrace the idea of "dynamic optimization" – program improvement that happens at runtime, after the initial compilation phase (e.g. see the proceedings of [FDD00]). The focus on dynamic optimization has two primary motivations:

- Even for static programming languages, static analysis is reaching its limits, but further optimization can take place once the actual data against which the program runs is available.
- Dynamic features are making their way into programming languages. The ability to incorporate new code, possibly arriving over the network, into a running application, is becoming less esoteric and more of a requirement.

Supporting full reflection merely puts more of a burden on dynamic optimization. The primary insight that motivates our dynamic optimization strategy is

> *While nearly all parts of a program's structure are technically mutable, very little of it actually changes once the program gets going.*

Thus we may "optimistically optimize" the program. For example, if a virtual function consists of some three methods, and from the declared class of the argument variables we can choose the most applicable, we can remove all of the metaobject protocol general purpose dispatch overhead, and even inline the method, as long as we guard the optimization. For this example, we need to know that the only way our assumption can be violated is by a call to `add-method`, so we instrument `add-method` to undo any optimizations that depended on the set of virtual methods being constant.

We use a technique called *dynamic partial evaluation* [Sul01] to dynamically generate specialized versions of methods, with the MOP-specific calls optimized away. The technique of generating specialized versions of methods is related to the work on the Self [Cha92] and Cecil [DCG95] projects. The idea of optimizing with respect to "quasi-invariants" has been pursued by Pu et al. in the Synthesis kernel [PMI88] and then Synthetix [PAB+95] projects in the context of operating systems.

8 Summary

The net result of these initiatives is that the full power of a runtime MOP is available for AOP metaprogramming, but the front end tools enable the MOP to be integrated seamlessly and incrementally into the host programming environment.

Our research program consists of the following

- Virtual machine-level support for reflection. We are developing a Dynamic Virtual Machine that supports computational reflection and uses optimistic optimization to dynamically optimize away the overhead of reflection and a metaobject protocol.
- Syntactic abstraction for language extension. We have developed a procedural macro system for Java that allows the definition of aspect-oriented syntactic extensions to Java.

Using the Dynamic Virtual Machine and our front end language definition tools, we are exploring adding new features to Java, such as the predicate dispatching used in some of the examples in this paper as well as more powerful dispatching mechanisms such as multiple dispatch. We plan to use the approach of [CC99] to efficiently implement multiple and predicate dispatching.

References

[Böl99] Kai Böllert. On weaving aspects. In *Proceedings of Aspect-Oriented Programming Workshop at ECOOP'99, Lisbon, Portugal*, June 1999.

[BP01] Jonathan R. Bachrach and Keith Playford. The Java Syntactic Extender. In *Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, October 2001. See `http://www.ai.mit.edu/~jrb/jse`.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In Loren Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 238–255, N. Y., November 1–5 1999. ACM Press.

[Cha92] Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Computer Science Department, Stanford University, March 1992.

[DCG95] Jeffrey Dean, Craig Chambers, and David Grove. Selective specialization for object-oriented languages. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 93–102, La Jolla, California, 18–21 June 1995. *SIGPLAN Notices* 30(6), June 1995.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 186–211. Springer, 1998.

[FDD00] *Third ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3, Monterey, California, December 10, 2000*, December 2000.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[KdB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In Norman Meyrowitz, editor, *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, Orlando, FL, USA, October 1987. ACM Press.

[PAB+95] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles*, Copper Mountain CO (USA), December 1995. http://www.irisa.fr/EXTERNE/projet/lande/consel/papers/spec-sosp.ps.gz.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. In USENIX Association, editor, *Computing Systems, Winter, 1988.*, volume 1, pages 11–32, Berkeley, CA, USA, Winter 1988. USENIX.

[SMI84] B. SMITH. Reflection and semantics in lisp. In *Conference Record of POPL '84: The* 11$^{\text{th}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects 2*, volume 2053 of *LNCS*, pages 238–256. Springer-Verlag, May 2001.

A Proposed Topic for the Workshop: Crosscutting versus Encapsulation

One topic that I would appreciate some discussion on is the interaction between aspect-oriented programming / separation of concerns and the software engineering principles of encapsulation and modularity.

The basic idea is this: aspect-oriented programming is all about specifying functionality that may interact with, and have an impact on, a wide range of other application components. The impetus for aspect-oriented programming was the recurring realization that no matter how well an application was designed, the implementation of some features invariably "crosscut" the design space and involved modification to many disparate source components. Supplying tools to the programmer that allow succinct definition of such crosscutting features is a powerful idea, but there are obvious drawbacks as well.

Principle among the potential drawbacks of aspect-oriented programming tools is the fuzzy notion of "readibility" of the application. From the perspective of a programmer trying to figure out what an existing application does, it is very useful to be able to treat each module as a "black box" with clearly defined behavior and interfaces. When each aspect of an application may impact every other aspect, we may lose modularity and therefore understandability.