Type-Based Quantification of Aspect-Oriented Programs

Hridesh Rajan
Dept. of Computer Science
Iowa State University
Ames, IA, 50011
hridesh@cs.iastate.edu

Quantification is a distinguishing characteristic of AspectJ-like aspect-oriented languages. Such languages use advice constructs to modify the behavior of execution points. In this work, we contribute an approach and a language design for quantification based on type hierarchies that we call *type-based quantification*. The key idea is to superimpose a crosscutting type hierarchy over the object-oriented inheritance hierarchy. This crosscutting type hierarchy can then be utilized for quantification, instead of or in addition to current syntactic quantification mechanisms based on regular expressions. A subsequent evaluation reveals that type-based quantification improves the robustness of the advising code against base code changes, and makes it easier for the advice constructs to uniformly access contextual information about the join point without breaking the encapsulation of the advised code.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features—Data Types and Structures, Classes and Objects, Modules and Packages

General Terms
Design, Languages

Keywords
aspect-oriented, quantification, types

1. INTRODUCTION

Aspect-oriented languages [26, 15] have shown the potential to improve the separation of traditionally non-modular concerns. Aspect-oriented languages in the style of AspectJ use predicates, called *pointcuts*, to select points in the execution of the object-oriented program (base code [29]), called *join points*, for behavioral modifications by by advice constructs. Advice is patterned on ideas in Common Lisp Object System (CLOS) [40, ch 28]. Using predicates to select join points is also referred to as *quantification* in the aspect-oriented terminology [16, 18]. Except for a few approaches such as SetPoint [3], Functional Queries [14], etc, prominent means of quantification are lexical. Lexical pointcuts are fragile [41, 46], exhibit quantification failures [43], and make it unnecessarily hard to uniformly access relevant contextual information at the join point [43, pp. 170].

The contribution of this work is an alternative approach for join point selection. The key idea is to superimpose a crosscutting type hierarchy over the object-oriented type hierarchy. This superimposed type hierarchy explicitly creates another view of the program that is of interest from the perspective of another concern [22]. The advantages of explicitly imposing a type hierarchy are observed in a more robust quantification approach with respect to the base code changes, precise interfaces between the advised code and the code being advised that preserves encapsulation, and in the improved abilities to provide uniform contextual information to the advice construct.

The rest of this paper is organized as follows. Section 2 briefly describes aspect-oriented programming. Section 3 and 4 motivate and present our approach. Section 3 describes the problems in more detail. In Section 4, we first present our ideas at an abstract level introducing the notion of the types of a join point and *type-based quantification* of join points. We then present a language design that adopts our ideas in current aspect language design. Section 7 presents a discussion of related issues. Section 8 compares and contrasts our approach with related work and Section 9 concludes.

2. ASPECT-ORIENTED PROGRAMMING

Aspect-oriented software development (AOSD) techniques [15, 26] aim to improve the software engineers’ ability to separate conceptual concerns by providing new design and implementation mechanisms. The key argument for AOSD is that all *dimensions of design decisions, or concerns*, are not amenable to modularization by a single dimension of decomposition [45]. Instead, some concerns cut across the dominant dimension of decomposition. An aspect-oriented approach typically extends an object-oriented language to include concepts such as *join point*, which refers to a point in the execution of the program, and constructs to add additional behavior to be executed at these join points. These constructs improve the separation of traditionally
1 aspect Tracing {
2      pointcut tracedCall():
3          execution(* *(..));
4      before(): tracedExecution() {
5          /* Trace the Execution */
6      }
7  }

Figure 1: A Simple Example Aspect

non-modular concerns thereby enhancing modularity. Typically, the part of the system that can be adequately modularized using object-oriented techniques is referred to as the base [29]. Following the tradition of a popular language AspectJ [25], the module that encapsulates the (traditionally non-modular) new behavior added to the base is called an aspect. Not all techniques make such distinction, however [36, 39]. These languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. Inter-type declarations are beyond the scope of this work, so we will not discuss them here. A simple example is shown in Figure 1 to make the points concrete. An aspect (lines 1-7), modifies the behavior of a program at certain selected execution events exposed to such modification by the semantics of the programming language. These events are called join points. The execution of a method in the program in which the Tracing aspect appears is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for such modification declaratively – here, execution of any method. This selection process is often referred to as quantification [16, 17]. In AspectJ [4], for example, the expression call(public Point.SetX(..)) would mean selecting the join point call to the method SetX of the class Point. Selecting join points using these syntactic regular expressions is convenient, allowing join points that span over a large section of a program to be selected using simple expressions. For example, a simple expression calls(* *.*(..)) selects all method calls in the program. An advice (see lines 4-6) is a special method-like constructs that effect such a modification at each join point selected by a pointcut. For example, statements to output the trace at all method calls could be added. An advice would often access the context at the join points, such as to find the name of the method that is being called for tracing output. An aspect is a class-like module that uses these constructs to modify behaviors defined by the classes of a software system.

3. MOTIVATION

3.1 Untyped View of Join Points

The notion of join points is central to the notion of aspect-orientation, however, it has not received the attention that it deserves. Most attention is directed towards formalizing and validating the behavior modifications that happen at these join points [9, 48, 12, 28, 47]. The common knowledge is largely informal. The AspectJ programming guide [4], for example, informally defines a join point as a new concept and explains that it is a well-defined point in the execution of the program. Informally, we know that a certain point in the execution of a program is a kind of method-execution join point or a kind of field execution join point, etc. Beyond this macroscopic classification technique, current literature does not provide any other mean to classify or define these concepts in an aspect language design. The central research question of this work is what defines a join point? We contrast the argument that being a point in the execution of a program fully defines a join point. Instead, we argue that a join point is defined by its type.

At this juncture, we would like to direct the reader’s attention to Cardelli and Wegner’s argument [7] twenty-one year ago.

As soon as we start working in an untyped universe, we begin to organize it in different ways for different purposes. Types arise informally in any domain to categorize objects according to their usage and behavior. The classification of objects in terms of the purposes for which they are used eventually results in a more or less well-defined type system. Types arise naturally, even starting from untyped universes.

Untyped universes of computational objects decompose naturally into subsets with uniform behavior. Sets of objects with uniform behavior may be named and are referred to as types. For example, all integers exhibit uniform behavior by having the same set of applicable operations. Functions from integers to integers behave uniformly in that they apply to objects of a given type and produce values of a given type.

After a valiant organization effort, then, we may start thinking of untyped universes as if they were typed. But this is just an illusion, because it is very easy to violate the type distinctions we have just created. [7, pp. 471]

Join points are also traveling on the exact same road. From the completely untyped universe, where a point in the program is a join point, a “seemingly typed” world has emerged where an organization is imposed upon these computational objects. Completely untyped points in the programs are now organized into these kinds or types of join point based on their behavior. Embarrassing questions, similar to those that Cardelli and Wegner [7] point out, are asked about these computational entities. For example, can we view these entities uniformly from a behavioral modification point of view?

In the rest of this section, we discuss four problems that arise partially due to the untyped view of join points.

3.2 Fragile Pointcuts

The first problem is that due to the lack of an alternative, principled way, to select a subset of these join points for behavioral modification, current language designs employ mostly syntactic predicates as quantification mechanism. These syntactic predicates are likely to change in the face of base code modifications. Some have called this problem the fragile pointcut problem [41], others AOSD evolution paradox [46].

To illustrate let us consider the source code in Figure 2. The Figure shows two implementations. A simple List implementation that uses an inner collection, provides a method to add an element, and a method to add an array of
class List {
    collection innerList;
    public bool add(Element e){ return innerList.add(e); }
    public bool add(Element[] elist){
        foreach(Element e in elist)
            if(innerList.add(e)) return false;
        return true;
    }
看着(): call(public bool List.add(…)){ counter++; }
}

int counter=0;

aspect Counter {
    return true;
    if(!add(e)) return false;
    foreach(Element e in elist)
        public bool add(Element[] elist){
            collection innerList;
        
        }
    // Simple Element Counter Aspect
    aspect Counter {
        list_counter();
        after(): call(public bool List.add(…)){ counter++; }
    }
}

Figure 2: A Simple List Implementation

elements. A simple aspect Counter that counts the number of elements in the list using an after advice. An alternative implementation of the List class is shown in Figure 3 in which the method to add multiple elements is modified to use the method to add a single element multiple times. The listing shows that a seemingly innocuous change that should have been encapsulated in the class List is propagating to the aspects of the system triggering changes that may not be obvious without a through analysis of the encapsulated implementation.

3.3 Quantification Failure

The second problem is what Sullivan et al [43] have called quantification failure. In the context of the AO design of the Hypercursor system, they observed that “many join points that have to be advised in the same way cannot be captured by a quantified PCD, e.g., using wild-card notations. A separate PCD is required for each join point. There were about 180 places in the base code where logging was required. Most of the join points do not follow a common pattern. Not only is there a lack of meaningful naming conventions across the set of join points, but also variation in syntax: method calls, field setting, etc.” [43, pp. 170] In addition to that, they observe that many join points of interest are not available as interface elements but deeply embedded into the methods such as in iteration and conditional statements. Exposing such join points as additional language constructs [21, 38] seems to be a solution to the quantification failure, however, these constructs further couple the aspects with the base code and expose the implementation details of the base code violating encapsulation.

The root of quantification failure lies in existing techniques for join point classification and quantification. These techniques work by determining, for a given point in the program, whether it is a kind of execution, call, field access, etc. We can understand these techniques better by drawing an analogy to the untyped set theory. Let $J$ be the set of all potential join points in a program. The join point classification can be thought of as partitioning $J$ into disjoint subsets $\bigcup_{k \in KIND} J_k = J$, where $k \in KIND$ the set of different kinds of join points such as method-execution, field-access, etc. Some of these subsets may not be available for behavioral modification in a given language semantics. For example, iteration, conditional, and most expressions are not available in AspectJ.

The limitation of this view of join point classification, where it is fixed by the language semantics, partially leads to the quantification failure. The quantification failure arises mainly because in the existing language models one may not specify a user-defined decomposition of the base program. As long as the developer utilizes the dominant decomposition based on classes and methods, current quantification mechanisms work remarkably well and a large set of join points can be selected using succinct pointcut expressions. However, as soon as a different decomposition is needed to modularize a concern, language models need explicit enumerations, pointcut expressions become verbose and more fragile. Here by different decomposition, we mean a decomposition of the base concerns that is not the same as the dominant decomposition. Tarr and Ooshier have called it the tyranny of the dominant decomposition [45]. The irony is that modularization of precisely these type of concerns is driving the invention and the refinement of aspect-oriented techniques.

Existing techniques for quantification first determine the kind of join point selected and then further filters the results based on other constraints such as matching on names. We may think of evaluation of a pointcut expression $P$ as a function $\text{matchKind}: P \rightarrow KIND$ composed with the function $\text{matchJP}: J_{\text{kind}} \rightarrow \{\text{true, false}\}$, where this function evaluates to true for all filtered join points. This is similar to the function $\text{matched}$ defined by Wand et al [48, pp. 896]. The second part of this quantification technique is largely syntactic. As discussed previously, the problem with syntactic techniques is that they are likely to change in the face of base code modifications.

3.4 Context Exposure Issues

The third problem is with being able to retrieve the right context information from a join point and the fourth problem is with being able to retrieve a different set of contextual information from different join points selected by the same pointcut.

Current aspect languages provide an interface for accessing contextual (or reflective) information about a join point. A fundamental problem is that this interface between the join point and the aspects is fixed in current AspectJ-like languages. An aspect can access the contextual information at the join point using pointcuts such as this to access the executing object (this), target to access the target object (such as the target of a call), args to access the arguments at a join point, etc. Alternatively, one can explicitly marshal this information from an implicit argument, often called thisJoinPoint, available to the advice, where other miscellaneous information such as source code location, name, etc, is also available. This rather limited interface does not satisfy all usage scenarios.

Even the canonical concerns such as logging exhibit these problems. For modularizing the logging functionality in a program, the aspect developers need access to the context.
of the join points that are to be logged. This information is often stored in local variables at the location of the join point. However, local variables are not available to the advice as contextual information.

There are rational reasons for limiting the interface between the code being advised and the advising code. The use of this interface introduces coupling between the design of the advised and advising code. The thinner this interface is the lower the coupling will be, resulting in perhaps easier and independent evolution of these two designs. Extending the set of language constructs to include access to more primitives also takes away regularity from the language design [32]. Not all such primitives will be valid for all kinds of join points. As it is, current language constructs for retrieving contextual information are not completely regular, e.g. this, target, and arguments are not available at all join points [4]. However, in this work we show that without introducing irregularity and additional arbitrary coupling between the join points and the aspects, it is possible to access contextual information at the join point in a more flexible way.

4. TYPE-BASED QUANTIFICATION

We argue that while talking about a join point, one should not be concerned about its kind. Instead, one should ask about its type, which leads to the question. What is the type of a join point? Types have traditionally been used in programming languages to constrain the interaction of the rest of the world with an entity so that illegal operations on the entity are eliminated through static or dynamic check. Cardelli and Wegner aptly view it as a suit of armor [7]. We observe that in the case of a join point, the rest of the world (of aspect-like constructs and such) interacts with it through the reflective information that is exposed by the join point. The special aspect methods, advice, depend on this information at the join point to perform additional behavioral modifications.

Based on this observation, we define the type of a join point as an explicitly defined record of the types of reflective information exposed at the join point. A record is defined as finite association of values to labels [6]. The view is similar to that taken by Ligatti et al [31] and Clifton and Leavens [9] in their semantics but has not appeared in aspect language designs. The main argument is that advice and join point exchange data through the reflective information. Therefore, they mush agree upon the cardinality and the type of data that is to be exchanged. This view of join points hides the underlying representation of the join points from its client, limiting the interface to the explicitly exposed type.

To make these points concrete consider a classic example [25], where the aim is to build a simple tool for editing drawings comprising points, lines and other such figure elements (See Figure 4). The display always reflects the current state of a figure element. In a typical implementation of this simple tool, the concrete classes Point and Line implement the interface FigureElement. The class Display manages the display and provides a method update() for keeping the state of figure elements consistent. The aim now is to modularize the policy that states that display must be updated when the abstract state of a FigureElement changes.

In an aspect-oriented implementation of this example, an aspect will select all points that change the abstract state of all figure elements by writing pointcut expressions such as execution (FigureElement.set*(..)) || execution (FigureElement.moveBy(.,..)), where the intention is to select the execution of mutator methods that start with set and another mutator moveBy. This pointcut expressions will select appropriate join points, if and only if all such points in the program are systematically exposed, possibly by enforcing a design rule to do so [20]. This implementation is prone to fragility, quantification failure, and context exposure issues. Even when a design rule is enforced, the developer of a module has no local textual hint so she should expose the join points by following the naming convention.

Consider an alternative, based on our ideas of typed join points. This implementation contains a new existential type [34] or type abstraction called FigureElementChange. The declaration of the FigureElementChange type exposes a join point of type Change. The join point type Change in turn is defined as a record type (jpThis: FigureElement). The record type defines only one label jpThis that can be associated to values of type FigureElement. All conforming implementations of the FigureElement type such as the implementation of the Point and the Line class, are also evolved to become the conforming implementations of the FigureElementChange type. These implementations provide a concrete implementation of the join point Change. In Java this would be equivalent to implementing the FigureElementChange interface as well.

4.1 Selecting Join Points

Given the alternative described above, one would be able to select all join points that contribute to a change in a FigureElement by selecting all the classes that have the type FigureElementChange. An expression such as FigureElementChange+ can be used. This quantification strategy based strictly on types would be far more robust to base code changes, thus solving the fragile pointcut problem.

A module developers will have a principled way to provide a concrete implementation of the join point, similar to open modules [2]. As a result, the developer can now explicitly expose even those program points that were not amenable to syntactic quantification. This solves the problem of quantification failure. The implementations of these join points may point to different kinds of program points, eliminating
interface Members
| interface Methods
| ... |
|jpDec;

jpDec : joinpoint type identifier([formal_parameters]);

jpImpl : [attributes] joinpoint type
| identifier([argument_list]) block
| ; // For abstract join point implementations

Figure 5: Syntax of Join Point Declaration and Implementation

the need for explicit enumeration.

Finally, letting the developer provide the implementation of the join point, gives them the flexibility to expose the right context information at these join points. In the drawing editor example, the developer of Point, and Line class is perhaps the right person to identify what constitutes a FigureElement change, not the aspect developer. The same is true for the example presented in Figure 2 and 3. The responsibility to expose the desired join points and corresponding context should rest with the developer of the class List. If we are trying to modularize a concern such as logging, a module perhaps encapsulates the knowledge about the kind of events in that should be logged, and the kind of messages that should be logged about these events. Providing a flexible typed means to expose join points solves these problems.

Our proposal thus appears to solve the four problems with aspect-oriented language design and usage that we documented in Section 3. In the next two sections, we will confirm these initial observations using the Eos-I language design and some representative examples.

5. LANGUAGE DESIGN
Eos-I is a version of Eos [39, 37], an aspect-oriented extension of C# [13], a .NET [33] language. Eos was the first AspectJ-like language with first-class aspect instances and instance-level advising. Later versions of Eos also unified classes and aspects as classpects. Eos-I extends Eos with constructs for type-based quantification. The rest of this section presents the Eos-I language design model in detail.

5.1 Join Point Declaration
Eos-I adds a new construct join point declaration to Eos. The grammar production, jpDec, in Figure 5 presents our join point declaration construct. A jpDec has four parts. The first, joinpoint is a new keyword added to the language to disambiguate join point declarations from method and event declarations. The second, type specifies the return type at the join point. The third, identifier, specifies the name of the join point declaration. The fourth optional part, formalParameters, specifies the set and types of reflective information exposed by the join point. The second and the fourth part together define the type of the join point.

A type member declaration such as a class declaration, an interface declaration, etc. may contain one or more join point declaration. If a join point declaration is contained in an interface declaration, it may not provide a corresponding join point implementation. If a join point declaration is contained in an abstract class, it may optionally provide a corresponding join point implementation.

Figure 6 shows an example join point declaration Changed (line 4) inside an interface declaration FigureElementChange (lines 5). In principle, this join point declaration can also be included in the interface FigureElement, but here we choose to use a separate interface for clarity of presentation. The intention of this join point declaration is to provide an abstraction for all join points in the program that contribute to an abstract state change in a figure element, such as a moving point, line, etc. The type of this join point declaration is a record {void, jpThis : FigureElement}. A join point is of this type iff the return type at this join point is void and it exposes a contextual element of type FigureElementChange. Please note that at this time the semantics of the language does not support subtyping. We will explore these directions in future.

5.2 Join Point Implementation
A join point implementation serves to label contiguous region in a single lexical scope of the program as a join point. It does not expand the interface of a module. Rather, it only provides a concrete implementation for the join point declarations that are explicitly exposed at the modules’ interface. A join point implementation can label a list of statements, or an expression. As we will discuss later, the capability to address statements and expressions solves the quantification failure problem.

Our approach has two benefits compared to earlier proposals on providing statement and expression-level join points [21, 38] that allow pointcut expressions in external modules to select statement and expression level join points for behavioral modification by advice. First, the implementation of these join points is hidden from the design of the advising code by the typed interface. The advising code is never coupled with the encapsulated details of the base code, only with its interface. Second, a join point implementation provides explicit textual hint to the module developer, in the module code itself that may reduce unintentional impact of the base code changes on the aspect code.

The grammar production, jpImpl, in Figure 5 presents our join point implementation construct. A jpImpl has five parts. The first optional part, attributes, specifies attributes or annotations for the join point implementations. These annotations can also serve to quantify join points similar to annotation-based pointcuts in AspectJ. Similar to join point declaration the second, joinpoint, is a new keyword added to the language to disambiguate join point implementations from method and event declarations. The third, type specifies the return type at the join point. The fourth, IDENTIFIER, specifies the name of the join point dec-
laration. The fifth, \texttt{opt_argument_list}, specifies the context information that will be exposed by that join point. Finally, the sixth part, \texttt{joinpoint\_implementation} is either a semicolon or an expression or a list of statements in the code that constitute the join point shadow [23].

A type declaration explicitly specifies that it is part of a crosscutting type hierarchy. For example, in Figure 7 the class \texttt{Point} and \texttt{Line} declare to be part of the crosscutting type hierarchy \texttt{FigureElementChange} by specifying that they implement this interface (lines 1-2 and 18-19). When a type declaration implements an interface \texttt{I}, in other words, it claims to be of type \texttt{I}, it must provide implementations for interface member declarations such as methods, events, etc. If an interface declaration contains a join point declaration, corresponding join point implementation must also be provided. A key difference in semantics is that while a type declaration may provide exactly one member implementation corresponding to each interface member declaration for methods, events, etc. it may provide one or more join point implementations for an interface join point declaration. It must provide at least one, and may provide several implementations.

To make the ideas concrete, let us consider the class \texttt{Point} and the class \texttt{Line} in Figure 7. These classes implement the interface \texttt{FigureElementChange} and provides more then one join point implementations for the interface join point declaration \texttt{FigureElementChange.Changed}. Two join point implementations for the class \texttt{Point} (lines 6-8 and 12-15) and one join point implementation for the class \texttt{Line} (lines 22-25) are presented here. The rest are elided for presentation purposes. The first join point implementation encloses the body of the method \texttt{Point.SetX}, declaring this region in the program to be the join point shadow. The join point implementation also specifies that the current object will be exposed as the join point context \texttt{jp.This}. Note that the result of a more complex expression can also be exposed as a context. All sub-expressions of this complex expression must also be defined within the lexical scope of the join point implementation.

6. ANALYSIS

In this section, we analyze our approach with respect to two criteria: robustness against base code changes and the ability to provide uniform access to reflective information about the advised code to the advising code.

6.1 Robustness

For analyzing robustness against base code changes, let us consider two simple pointcuts in our drawing application in Figure 9. The purpose of these pointcuts is to expose the abstract state transitions in the \texttt{FigureElement} so that aspects can add behaviors at these state transitions [7]. The first pointcut, taken from [20, pp. 56], is a syntactic pointcut that uses regular expression such as \texttt{set\_\{..\}} to select all join points, whereas the second pointcut uses the type-hierarchy \texttt{FigureElementChange} to aggregate all join points implementations by the modules that are crosscut by this type-hierarchy.

The syntactic approach wins hands down with respect to the ease of the first time implementation. It is definitely much easier for the programmer. By just writing a simple crosscutting type hierarchy, they can select join points throughout the code base. On the other hand, using our approach a
programmer will have to systematically modify modules to implement the `FigureElementChange` interface, as described in the previous section.

However, the ease of selecting join points provided by syntactic approaches may turn out to be a double-edge sword. For example, consider the following evolutionary scenario. Each composite `FigureElement` has to be extended to include a reference to the parent `FigureElement` for ease of traversing the composite structure, e.g., `Point` is to be extended to include a reference to `Line`. A mutator `setParent` and an accessor `getParent` for this reference are also added. The syntactic pointcut in Figure 9 will also select the join points `call to mutator setParent` for advising, which is incorrect. Setting the reference to the parent, just for ease of implementation, is not an abstract state transition for a `FigureElement`. An aspect-oriented tool such as AJDT [1] may warn the developer against such inadvertent selection of join point by showing visual cues at the shadow of the join point.

A solution is to explicitly exclude the calls to `setParent` by adding a simple expression `&& !call<void FigureElement+.setParent(..)`; however, this solution is not desirable due to two reasons. First, this enumerated list of exceptions can get large in realistic systems. Second, each item in this list of exception introduces a dependency between the base code and the aspect code, thereby increasing the coupling between the two.

In our approach, this change will not affect the selected join points. The calls to method `setParent` are not automatically selected by the pointcut. However, in cases where the join points exposed by a module are affected by a change, the developer may choose to restrict or extend the join point implementations in the module. For example, while changing a `FigureElement` subclass to include the methods `setParent` and `getParent`, the developer may choose to extend the implementation of the join point declaration `FigureElementChange.Changed` for that subclass to include the calls to `setParent`.

In summary, it is easier to separate a crosscutting concern using syntactic quantification; however, changes that affect the advised code have a direct impact on the advising code implementation. Some of these impacts may potentially break the advising code. On the other hand, type-based quantification requires preparation of the code to be advised to systematically superimpose a crosscutting type-hierarchy. However, advising code is shielded from the changes in advised code by the type-hierarchy. Our approach is thus more robust compared to syntactic quantification against base code changes.

### 6.2 Uniform Reflective Access

For the purpose of this analysis, let us consider a canonical concern `logging`. Method call tracing is easily implemented using a combination of quantification expression such as `call(* , . {...})`, which selects all desired join points, and the standard reflective interface `this.joinpoint` that is available to the advising code in AspectJ-like languages. The implementation of the logging concern is, however, significantly difficult using syntactic quantification because a correct logging implementation requires access to the join point specific messages. The join point specific messages are often constructed from the local information available in the lexical scope of the join point. This information is not available to the advice. Please see Sections 3.3 and 3.4 for more discussion.

Figure 10 and 11 show an implementation of the logging concern using type-based quantification. To enable logging in the drawing application, a new type-hierarchy `IRecordable` is defined. This type-hierarchy provides a join point declaration `Log` of type `void, string : message`. The join point declaration means that the conforming join point implementations will expose one reflective variable of type `string`, which will contain the message to be logged. The class `Point` and `Line` also declare to be of type `IRecordable` by implementing this interface (lines 2 and 14). These classes may provide several implementations of the join point declaration `Log`.

Two such join point implementations are shown in the figure. Both these join point implementations are contained in the class constructors for the `Point` and the `Line` class. In each case, a class specific message is created using the variables available in the lexical scope of the join point implementation. Note that both messages are unique to the advised code; however, the advising code uses the public reflective variable `message` made available by the crosscutting type hierarchy to uniformly access these messages.

![Figure 10: The IRecordable interface](image)

### 7. DISCUSSION

Our proposal would not be complete without the discussion of obliviousness [16, 17]. Obliviousness is a widely
accepted tenet for aspect-oriented software development. In
an oblivious AOSD process, the designers and developers of
base need not be aware of, anticipate or design code to be
advised by aspects. This criterion, although attractive, has
been questioned by others for various reasons. Clifton and
Leavens [10] were the first to point out that a category of
aspects that they call assistants should not be used obvi-
uously. There is at least some consensus among researchers
that complete obliviousness between base and aspect de-
signers and developers may not be possible [2, 8, 11, 12,
20, 27, 43]. To understand the behavior of a module in the
presence of aspects and for independent evolution of base
and aspect code, one need to first find and understand all
aspects that apply to that module. Tools such as AspectJ De-
velopment Tools (AJDT) alleviate the problem [1] but do
not completely solve it.

According to Sullivan et al [43], there are many variants
of the notion of obliviousness, language-level obliviousness,
feature obliviousness, designer obliviousness, and pure obli-
vousness. Language-level obliviousness comes from intro-
ducing quantification mechanisms in the language. Feature
obliviousness is when designer of the base code is aware of
the presence of aspects but unaware of the features that the
aspect implements. Designer obliviousness comes when the
base code designer can be unaware of the presence of an as-
pect. Pure obliviousness is when both base and aspect code
designers are symmetrically unaware of each other. Our pro-
posal on type-based quantification discards designer obliv-
iousness. The base code designers have to prepare their code
for advising by aspects. However, similar to XPI’s [43, 20] it
preserves feature obliviousness. The base code designers can
be completely unaware of spectators [8] or harmless aspects
[12] that quantify on the interfaces that they implement.

In our drawing example, the FigureElement expose the
abstract event “A change in the FigureElement” without
being aware of the type of aspects that may be interested in
advising such abstract events. The example that we dis-
1cussed was the modularization of the display update policy,
but the base code designers need not make separate prepa-
ration for a persistence policy that updates the persistent
representation of the FigureElement, whenever there is a
change. Neither does she need to be aware of an integration
relationship [44] between a visual and a textual relationship
of the FigureElement, similar to that between Word and
Visio in the fault-tree analysis tool Galileo [42], where the
representations are to be consistent with each other. All
these policies may be implemented simultaneously as differ-
ent aspects without the base code designer being aware of
any of them and without these aspects being dependent on
the details of the advised base code.

8. RELATED WORK

Aldrich’s proposal on Open Modules[2] is closely related
to this work [2]. Both approaches have two similar advan-
tages. First, like our work, open modules also allows a class
developer to explicitly expose pointcuts for behavioral mod-
ifications by aspects. The implementations of these point-
cuts remain hidden from the aspects. As a result, the im-
 pact of base code changes on the aspect is reduced. Second,
with appropriate language extensions, an explicitly exposed
pointcut may also expose the right contextual information
uniformly across the join points selected by the pointcut.
However, open modules exacerbates the problem of quan-
tification failure. Each explicitly declared pointcuts has to
be enumerated by the aspect for advising. On the other
hand, our approach significantly simplifies quantification.
Instead of manually enumerating the join points of inter-
est, one can use the crosscutting type-hierarchy for implicit
non-syntactic selection of join points.

Similar to Open Modules, a programmer using type-based
quantification need to systematically modify modules in a
system that a given concern crosscuts to expose join points
that are to be advised. These modules will be modified to
conform to the crosscutting type hierarchy. For exam-
ple, the modules Line, Point etc. will be modified to con-
form to the FigureElementChange type hierarchy. To con-
form to the type-hierarchy the modules Line and Point will
implement the interface FigureElementChange. They will
each provide an implementation of the exposed join points
Change. However, unlike Open Modules once these modules
have declared to be part of the FigureElementChange hier-
archy, no awkward enumeration of explicitly exposed join
points is necessary for quantification. An expression such
as FigureElementChange+ Change aggregates these exposed
join points.

Ongkingco et al’s [35] work on adding Open Modules to
AspectJ [25] is similarly related to our work. Ongkingco et al
’s [35] propose language constructs such as friend, advertise,
and expose to allow unrestricted access to join points inside
or external to a module with varying degree of freedom.
Sullivan et al. [43] recently proposed a methodology for
aspect-oriented design based on design rules. The key idea
is to establish a design rule interface that serves to decouple
the base design and the aspect design. These design rules [5]
govern exposure of execution phenomena as join points, how
they are exposed through the join point model of the given
language, and constraints on behavior across join points (e.g.
provides and requires conditions [20]). These design rule
interfaces were later called crosscut programming interface
(XPI) by Griswold et al. [20].

XPIs prescribe rules for join point exposure, but do not
provide a compliance mechanism. Griswold et al. have
shown that at least some design rules can be enforced au-
tomatically. In this work, we present a principled quanti-
fication technique that might help to enforce XPI design
rules. The key idea is to superimpose a crosscutting type
hierarchy over the OO type hierarchy of the base program.
The quantification then becomes equivalent to using these
crosscutting types. Enforcing design rules become equiva-
lent to type checking of programs. One can then use this
crosscutting type hierarchy for quantification instead of or
in addition to syntax-based quantification.

Another related area is implicit invocation [19] and
mediator-based design styles [44]. In this design style, in
addition to providing methods that can be called, modules
declare and announce events. Other modules can register
operations to be invoked by events. An invocation rela-
tion is thus created without introducing names dependences.

Our approach for type-based quantification (as well as Open
Modules [2] has the similar rationale that visible actions of a
modules should be part of its interface, and interfaces should
be explicit. The notion of superimposing a crosscutting
type-hierarchy that our work introduces is, however, novel.
This type hierarchy provides a method for easy quantifica-
tion for behavioral modifications. Similar to Open Modules,
in implicit invocation systems, a developer has to resort to
explicit and possibly error-prone enumerations to achieve the same results.

9. CONCLUSION AND FUTURE WORK

The main contribution of this work is a mechanism for type-based quantification in aspect-oriented programs, including the Eos-I language, a compiler able to handle production code, and evidence that suggests that this synthesis has potentially significant benefits in aspect-oriented program design. In particular, we showed that type-based quantification improves the robustness of the advising code against base code changes, and makes it easier for the advice constructs to uniformly access reflective information about the join point without breaking the encapsulation of the advised code. Our current proposal offers new directions for the join point without breaking the encapsulation of the constructs to uniformly access reflective information about the join point declaration serves as the post-condition to advice invocation, similar to XPI's. The post-condition of the join point declaration serves as the pre-condition to advice invocation, similar to the post-condition of the join point declaration serves as the post-condition to advice invocation.

10. ACKNOWLEDGMENTS

The author is supported in part by the National Science Foundation under grant CNS-0627354 and the Iowa State University’s startup grant. The discussions with Gary Leavens were very helpful.

11. REFERENCES


