Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

ABSTRACT

This paper makes two contributions to a formal understanding of aspect-oriented programming. First, we define `TinyAspect`, a formal model capturing core AOP concepts. Compared to previous formalizations of AOP constructs, `TinyAspect`is extremely small, models aspects at the source level, and is defined using structured operational semantics and syntax-directed typing rules. In combination, these properties make it easy to investigate aspect-oriented language extensions and prove theorems about them.

Second, we propose Open Modules, a module system for `TinyAspect`that guarantees modular reasoning in the presence of aspects. Modular reasoning can be challenging in AOP systems because advice can change the semantics of a module from the outside. Open Modules are "open" in that external aspects can advise functions and pointcuts in their interface, providing significant aspect-oriented expressiveness that is missing in non-AOP systems. In order to guarantee modular reasoning, however, our system places limits on advice: external aspects may not advise function calls internal to a module, except for calls explicitly exposed through pointcuts in the module's interface.

We use a notion of bisimulation to show that Open Modules enforce Reynolds' abstraction theorem, a strong encapsulation property. This theorem guarantees that clients are unaffected by changes to a module, as long as those changes preserve the semantics of the functions and pointcuts in the module's interface.

1. Outline

This paper makes two contributions: the definition of a new formal model for aspect-oriented programming, and a proposed module system for aspects. In order to cleanly separate these contributions, we motivate each in its own section. We begin in Section 2 with the presentation of `TinyAspect`, a minimal core language for aspect-oriented programming.

Section 3 motivates the need for better module systems in aspect-oriented programming, and provides an overview of Open Modules, our proposed design. Section 4 formalizes the Open Modules proposal as an extension to `TinyAspect`. In Section 5 we use a notion of bisimulation to show that Open Modules enforce Reynolds' abstraction theorem, a strong encapsulation property. Section 6 discusses related work, Section 7 outlines future work, and Section 8 concludes.

Names	n	::=	x
Expressions	e	::=	$n \mid \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e \mid e_1\ e_2 \mid \texttt{()}$
Declarations	d	::=	$\bullet$
		|	$\texttt{val}\ x = e\ \ d$
		|	$\texttt{pointcut}\ x = p\ \ d$
		|	$\texttt{around}\ p(x{:}\tau) = e\ \ d$
Pointcuts	p	::=	$n \mid \texttt{call}(n)$
Types	τ, σ	::=	$\texttt{unit} \mid \tau_1 \rightarrow \tau_2 \mid \texttt{pc}(\tau_1 \rightarrow \tau_2)$

Figure 1: `TinyAspect` Source Syntax

2. The TinyAspect Language

We would like to use a formal model of aspect-oriented programming in order to study language extensions like the module system discussed in the next section. While other researchers have used denotational semantics [22], big-step operational semantics [12], and translation systems [14, 20] to study the semantics of aspect-oriented programming, small-step operational semantics have the advantage of providing a simple and direct semantics that is amenable to syntactic proof techniques.

Jagadeesan et al. have proposed an operational semantics for the core of AspectJ, incorporating several different kinds of pointcuts and advice in an object-oriented setting [10]. These features are ideal for modeling AspectJ, but the complexity of the model makes it tedious to prove properties about the system.

Walker et al. propose a much simpler formal model incorporating just the lambda calculus, labeled join points, and advice [21]. However, their system is not intended to model the source-level constructs of langauges like AspectJ directly; instead, it is a foundational calculus into which source-level AOP constructs can be translated. It is considerably more general than existing languages like AspectJ, and so properties that might be true at the source level of a language may not hold in the foundational calculus. Thus, a source-level formal model would be a more effective way to investigate many source-level properties.

We have developed a new functional core language for aspect-oriented programming called `TinyAspect` that is intended to make proofs of source-level properties as straightforward as possible. As the name suggests, `TinyAspect` is tiny, containing only the lambda calculus with units, declara-

tions, pointcuts, and `around` advice. `TinyAspect` directly models AOP constructs similar to those found in AspectJ, making source-level properties easy to specify and prove using small-step operational semantics and standard syntactic techniques. Although we are working in an aspect-oriented, functional setting, our system's design is inspired by that of Featherweight Java [9], which has been successfully used to study a number of object-oriented language features.

Figure 1 shows the syntax of `TinyAspect`. Our syntax is modeled after ML [15], so that `TinyAspect` programs are easy to read and understand. Names in `TinyAspect` are simple identifiers; we will extend this to paths when we add module constructs to the language. Expressions include the monomorphic lambda calculus – names, functions, and function application. To this core, we add a primitive unit expression, so that we have a base case for types. We could add primitive booleans and integers in a completely standard way. Since these constructs are orthogonal to aspects, we omit them.

In most aspect-oriented programming languages, including AspectJ, the pointcut and advice constructs are second-class and declarative. So as to be an accurate source-level model, a `TinyAspect` program is made up of a sequence of declarations. Each declaration defines a scope that includes the following declarations. A declaration is either the empty declaration, or a value binding, a pointcut binding, or advice. The `val` declaration gives a static name to a value so that it may be used or advised in other declarations.

The `pointcut` declaration names a pointcut in the program text. A pointcut of the form `call`(n) refers to any call to the function value defined at declaration n, while a pointcut of the form n is just an alias for a previous pointcut declaration n. A real language would have more pointcut forms; we include only the most basic possible form in order to keep the language minimal.

The `around` declaration names some pointcut p describing calls to some function, binds the variable x to the argument of the function, and specifies that the advice e should be run in place of the original function. Inside the body of the advice e, the special variable `proceed` is bound to the original value of the function, so that e can choose to invoke the original function if desired.

`TinyAspect` types τ include the `unit` type, function types of the form $\tau_1 \rightarrow \tau_2$, and pointcut types representing calls to a function of type $\tau_1 \rightarrow \tau_2$.

2.1 Fibonacci Caching Example

We illustrate the language by writing the Fibonacci function in it, and writing a simple aspect that caches calls to the function to increase performance. While this is not a compelling example of aspects, it is standard in the literature and simple enough for an introduction to the language.

Figure 2 shows the `TinyAspect` code for the Fibonacci function. We assume integers and booleans have been added to illustrate the example.

`TinyAspect` does not have recursion as a primitive in the language, so the `fib` function includes just the base case of the Fibonacci function definition, returning 1.

We use `around` advice on calls to `fib` to handle the recursive cases. The advice is invoked first whenever a client calls `fib`. The advice is invoked first whenever a client calls `fib`. The body of the advice checks to see if the argument is greater than 2; if so, it returns the sum of `fib(x-1)` and `fib(x-2)`. These recursive calls are intercepted by the advice, rather than the original function, allowing recursion to work properly. In the case when the argument is less than 3, the advice invokes `proceed` with the original number `x`. Within the scope of an advice declaration, the special variable `proceed` refers to the advised definition of the function. Thus, the call to `proceed` is forwarded to the original definition of `fib`, which returns 1.

```
val fib = fn x:int => 1
around call(fib) (x:int) =
    if (x > 2)
        then fib(x-1) + fib(x-2)
        else proceed x

(* advice to cache calls to fib *)
val inCache = fn ...
val lookupCache = fn ...
val updateCache = fn ...

pointcut cacheFunction = call(fib)
around cacheFunction(x:int) =
    if (inCache x)
        then lookupCache x
        else let v = proceed x
            in updateCache x v; v
```

Figure 2: The Fibonacci function written in `TinyAspect`, along with an aspect that caches calls to `fib`.

In the lower half of the figure is an aspect that caches calls to `fib`, thereby allowing the normally exponential function to run in linear time. We assume there is a cache data structure and three functions for checking if a result is in the cache for a given value, looking up an argument in the cache, and storing a new argument-result pair in the cache.

So that we can make the caching code more reusable, we declare a `cacheFunction` pointcut that names the function calls to be cached–in this case, all calls to `fib`. Then we declare `around` advice on the `cacheFunction` pointcut which checks to see if the argument `x` is in the cache. If it is, the advice gets the result from the cache and returns it. If the value is not in the cache, the advice calls `proceed` to calculate the result of the call to `fib`, stores the result in the cache, and then returns the result.

In the semantics of `TinyAspect`, the last advice to be declared on a declaration is invoked first. Thus, if a client calls `fib`, the caching advice will be invoked first. If the caching advice calls `proceed`, then the first advice (which recursively defines `fib`) will be invoked. If that advice in turn calls `proceed`, the original function definition will be invoked. However, if the advice makes a recursive call to `fib`, the call will be intercepted by the caching advice. Thus, the cache works exactly as we would expect–it is invoked on all recursive calls to `fib`, and thus it is able to effectively avoid the exponential cost of executing `fib` in the naïve way.

2.2 Operational Semantics

We define the semantics of `TinyAspect` more precisely as a set of small-step reduction rules. These rules translate a series of source-level declarations into the values shown in Figure 3.

Expression-level values include the unit value and func-

Expression values	v	::=	`()` | `fn` $x{:}\tau$ `=>` e | ℓ
Pointcut values	p_v	::=	`call`(ℓ)
Declaration values	d_v	::=	$\bullet$
		|	`val` $x \equiv v$ d_v
		|	`pointcut` $x \equiv p_v$ d_v
Evaluation contexts	C	::=	$\Box\ e_2$ | $v_1\ \Box$ | `val` $x = \Box$ d
		|	`val` $x \equiv v$ $\Box$
		|	`pointcut` $x \equiv p_v$ $\Box$

Figure 3: `TinyAspect` Values and Contexts

tions. In `TinyAspect`, advice applies to declarations, not to functions. We therefore need to keep track of declaration usage in the program text, and so a reference to a declaration is represented by a label ℓ. In the operational semantics, below, an auxiliary environment keeps track of the advice that has been applied to each declaration.

A pointcut value can only take one form: calls to a particular declaration ℓ. In our formal system we model execution of declarations by replacing source-level declarations with "declaration values," which we distinguish by using the $\equiv$ symbol for binding.

Figure 3 also shows the contexts in which reduction may occur. Reduction proceeds first on the left-hand side of an application, then on the right-hand side. Reduction occurs within a value declaration before proceeding to the following declarations. Pointcut declarations are atomic, and so they only define an evaluation context for the declarations that follow.

Figure 4 describes the operational semantics of `TinyAspect`. A machine state is a pair (η, e) of an advice environment η (mapping labels to values) and an expression e. Advice environments are similar to stores, but are used to keep track of a mapping from declaration labels to declaration values, and are modified by advice declarations. We use the $\eta[\ell]$ notation in order to look up the value of a label in η, and we denote the functional update of an environment as $\eta' = [\ell{\mapsto}v]\ \eta$. The reduction judgment is of the form $(\eta, e) \mapsto (\eta', e')$, read, "In advice environment η, expression e reduces to expression e' with a new advice environment η'."

The rule for function application is standard, replacing the application with the body of the function and substituting the argument value v for the formal x. We normally treat labels ℓ as values, because we want to avoid "looking them up" before they are advised. However, when we are in a position to invoke the function represented by a label, we use the rule *r-lookup* to look up the label's value in the current environment.

The next three rules reduce declarations to "declaration values." The `val` declaration binds the value to a fresh label and adds the binding to the current environment. It also substitutes the label for the variable x in the subsequent declaration(s) d. We leave the binding in the reduced expression both to make type preservation easier to prove, and also to make it easy to extend `TinyAspect` with a module system which will need to retain the bindings. The `pointcut` declaration simply substitutes the pointcut value for the variable x in subsequent declaration(s).

The `around` declaration looks up the advised declaration

$$\frac{}{(\eta,\ (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ e)\ v) \mapsto (\eta,\ \{v/x\}e)}\ \textit{r-app}$$

$$\frac{\eta[\ell] = v_1}{(\eta,\ \ell\ v_2) \mapsto (\eta,\ v_1\ v_2)}\ \textit{r-lookup}$$

$$\frac{\ell \notin \mathit{domain}(\eta) \quad \eta' = [\ell{\mapsto}v]\ \eta}{(\eta,\ \texttt{val}\ x = v\ \ d) \mapsto (\eta',\ \texttt{val}\ x \equiv \ell\ \ \{\ell/x\}d)}\ \textit{r-val}$$

$$\frac{}{\begin{array}{c}(\eta,\ \texttt{pointcut}\ x = \texttt{call}(\ell)\ \ d) \mapsto \\ (\eta,\ \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ \{\texttt{call}(\ell)/x\}d)\end{array}}\ \textit{r-pointcut}$$

$$\frac{\begin{array}{c}v' = (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ \{\ell'/\texttt{proceed}\}e) \\ \ell' \notin \mathit{domain}(\eta) \quad \eta' = [\ell{\mapsto}v', \ell'{\mapsto}\eta[\ell]]\ \eta\end{array}}{(\eta,\ \texttt{around}\ \texttt{call}(\ell)(x{:}\tau) = e\ \ d) \mapsto (\eta',\ d)}\ \textit{r-around}$$

$$\frac{(\eta,\ e) \mapsto (\eta',\ e')}{(\eta,\ C[e]) \mapsto \eta',\ C[e'])}\ \textit{r-context}$$

Figure 4: `TinyAspect` Operational Semantics

ℓ in the current environment. It places the old value for the binding in a fresh label ℓ', and then re-binds the original ℓ to the body of the advice. Inside the advice body, any references to the special variable `proceed` are replaced with ℓ', which refers to the original value of the advised declaration. Thus, all references to the original declaration will now be redirected to the advice, while the advice can still invoke the original function by calling `proceed`.

The last rule shows that reduction can proceed under any context as defined in Figure 3.

2.3 Typechecking

Figure 5 describes the typechecking rules for `TinyAspect`. Our typing judgment for expressions is of the form $\Gamma; \Sigma \vdash e : \tau$, read, "In variable context Γ and declaration context Σ expression e has type τ." Here Γ maps variable names to types, while Σ maps labels to types (similar to a store type).

The rules for expressions are standard. We look up the types for variables and labels in Γ and Σ, respectively. Other standard rules give types to the `()` expression, as well as to functions and applications.

The interesting rules are those for declarations. We give declaration signatures β to declarations, where β is a sequence of variable to type bindings. The base case of an empty declaration has an empty signature. For `val` bindings, we ensure that the expression is well-typed at some type τ, and then typecheck subsequent declarations assuming that the bound variable has that type. Pointcuts are similar, but the rule ensures that the expression p is well-typed as a pointcut denoting calls to a function of type $\tau_1 \to \tau_2$. The around advice rule checks that the declared type of x matches the argument type in the pointcut, and checks that the body is well-typed assuming proper types for the variables x and `proceed`.

Finally, the judgment $\Sigma \vdash \eta$ states that η is a well-formed environment with typing Σ whenever all the values in η have the types given in Σ. This judgment is analogous to store

$$\frac{x{:}\tau \in \Gamma}{\Gamma;\Sigma \vdash x : \tau}\ \textit{t-var}$$

$$\frac{\Gamma;\Sigma \vdash n : \tau_1 \to \tau_2}{\Gamma;\Sigma \vdash \texttt{call}(n) : \texttt{pc}(\tau_1 \to \tau_2)}\ \textit{t-pctype}$$

$$\frac{\ell{:}\tau \in \Sigma}{\Gamma;\Sigma \vdash \ell : \tau}\ \textit{t-label}$$

$$\frac{}{\Gamma;\Sigma \vdash \texttt{()} : \texttt{unit}}\ \textit{t-unit}$$

$$\frac{\Gamma, x{:}\tau_1;\Sigma \vdash e : \tau_2}{\Gamma;\Sigma \vdash \texttt{fn}\ x{:}\tau_1\ \texttt{=>}\ e : \tau_1 \to \tau_2}\ \textit{t-fn}$$

$$\frac{\Gamma;\Sigma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma;\Sigma \vdash e_2 : \tau_2}{\Gamma;\Sigma \vdash e_1\ e_2 : \tau_1}\ \textit{t-app}$$

$$\frac{}{\Gamma;\Sigma \vdash \bullet : \bullet}\ \textit{t-empty}$$

$$\frac{\Gamma;\Sigma \vdash e : \tau \quad \Gamma, x{:}\tau;\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \texttt{val}\ x = e\ \ d : (x{:}\tau, \beta)}\ \textit{t-val}$$

$$\frac{\Gamma;\Sigma \vdash p : \texttt{pc}(\tau_1 \to \tau_2) \quad \Gamma, x{:}\texttt{pc}(\tau_1 \to \tau_2);\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \texttt{pointcut}\ x = p\ \ d : (x{:}\texttt{pc}(\tau_1 \to \tau_2), \beta)}\ \textit{t-pc}$$

$$\frac{\begin{array}{c}\Gamma;\Sigma \vdash p : \texttt{pc}(\tau_1 \to \tau_2) \quad \Gamma;\Sigma \vdash d : \beta \\ \Gamma, x{:}\tau_1, \texttt{proceed}{:}\tau_1 \to \tau_2;\Sigma \vdash e : \tau_2\end{array}}{\Gamma;\Sigma \vdash \texttt{around}\ p(x{:}\tau_1) = e\ \ d : \beta}\ \textit{t-around}$$

$$\frac{\forall \ell.(\Sigma[\ell] = \tau \wedge \eta[\ell] = v \implies \bullet;\Sigma \vdash v : \tau)}{\Sigma \vdash \eta}\ \textit{t-env}$$

Figure 5: `TinyAspect` Typechecking

typings in languages with references.

2.4 Type Soundness

We now state progress and preservation theorems for `TinyAspect`. The theorems quantify over both expressions and declarations using the metavariable E, and quantify over types and declaration signatures using the metavariable T. The progress property states that if an expression is well-typed, then either it is already a value or it will take a step to some new expression.

Theorem 1 (Progress)
If $\bullet;\Sigma \vdash E : T$ and $\Sigma \vdash \eta$, then either E is a value or there exists η' such that $(\eta, E) \mapsto (\eta', E')$.

Proof: By induction on the derivation of $\bullet;\Sigma \vdash E : T$. ■

The type preservation property states that if an expression is well-typed and it reduces to another expression in a new environment, then the new expression and environment are also well-typed.

Theorem 2 (Type Preservation)
If $\bullet;\Sigma \vdash E : T$, $\Sigma \vdash \eta$, and $(\eta, E) \mapsto (\eta', E')$, then there exists some $\Sigma' \supseteq \Sigma$ such that $\bullet;\Sigma' \vdash E' : T$ and $\Sigma' \vdash \eta'$.

Proof: By induction on the derivation of $(\eta, E) \mapsto (\eta', E')$. The proof relies on a standard substitution and weakening lemmas. ■

Together, progress and type preservation imply type soundness. Soundness means that there is no way that a well-typed `TinyAspect` program can get stuck or "go wrong" because it gets into some bad state.

3. Open Modules

In this section, we explore one possible way to define a module system for aspects. The following section models our proposed module system in `TinyAspect`, providing an initial evaluation of the core language design, and gaining insight into the potential benefits of our module system.

3.1 Motivation

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules in order to hide information that is likely to change [17]. Thus if change is anticipated with reasonable accuracy, the system can be evolved with local rather than global system modifications, easing many software maintenance tasks. Furthermore, the correctness of each module can be verified in isolation from other modules, allowing developers to work independently on different sub-problems.

Unfortunately, developers do not always respect the information hiding boundaries of modules–it is often tempting to reach across the boundary for temporary convenience, while causing more serious long-term evolution problems. Thus, encapsulation mechanisms such as Java's packages and public/private data members were developed to give programmers compiler support for enforcing information hiding boundaries.

The central insight behind aspect-oriented programming is that conventional modularity and encapsulation mechanisms are not flexible enough to capture many concerns that are likely to change. These concerns cannot be effectively hidden behind information-hiding boundaries, because they are scattered in many placed throughout the system and tangled together with unrelated code. Aspect-oriented programming systems provide mechanisms for modularizing a more diverse set of concerns. However, few aspect-oriented programming projects have addressed the problem of providing an encapsulation facility for aspect-oriented programming.

3.2 Existing Encapsulation Approaches

The most widely-used AOP system, AspectJ, leaves Java's existing encapsulation mechanisms largely unchanged [11]. AspectJ provides new programming mechanisms that capture concerns which crosscut Java's class and package structure. Because these mechanisms can reach across encapsulation boundaries, AspectJ does not enforce information hiding between aspects and other code.

For example, Figure 6 shows how an aspect can depend on the implementation details of another module. The figure shows two different `Shape` subclasses, one representing points and another representing rectangles. Both classes have a method `moveBy`, which moves the rectangles on the screen. An assurance aspect checks certain invariants of

```
package shape;

public class Point extends Shape {
  public void moveBy(int dx, int dy) {
    x += dx; y += dy;
  ...
}

public class Rectangle extends Shape {
  public void moveBy(int dx, int dy) {
    p1x += dx; p1y += dy;
    p2x += dx; p2y += dy;
  ...
}


package assure;

aspect AssureShapeInvariants {
  pointcut moves():
      call(void shape.Shape+.moveBy(..));

  after(): moves() {
    scene.checkInvariants();
  }
}
```

Figure 6: In this AspectJ code, the correctness of the shape invariants aspect depends on the implementation of the shapes. If the implementation is changed so that `Rectangle` uses `Point` to hold its coordinates, then the invariants will be checked in the middle of a `moveBy` operation, possibly leading to a spurious invariant failure.

the scene every time a shape moves. The aspect is triggered by a pointcut made up of all calls to the `moveBy` function in shapes. We assume the assurance aspect is checking application-level invariants, rather than invariants specific to the `shape` package, and therefore it is defined in a package of its own.

Unfortunately, this aspect is tightly coupled to the implementation details of the `shape` package, and will break if these implementation details are changed. For example, consider what happens if the rectangle is modified to store its coordinates as a pair of points, rather than two pairs of integer values. The body of `Rectangle.moveBy` would be changed to read:

```
p1.moveBy(dx, dy);
p2.moveBy(dx, dy);
```

Now the `moves` pointcut will be invoked not only when the `Rectangle` moves, but also when its constituent points move. Thus, the scene invariants will be checked in the middle of the rectangle's `moveBy` operation. Since the scene invariants need not be true in the intermediate state of motion, this additional checking could lead to spurious invariant failures.

The aspect in Figure 6 violates the information hiding boundary of the `shape` package by placing advice on method calls within the package. This means that the implementor of `shape` cannot freely switch between semantically equivalent implementations of `Rectangle`, because the external aspect may break if the implementation is changed. Because the aspect violates information hiding, evolving the `shape` package becomes more difficult and error prone.

AspectJ is not the only system in which aspects can violate information hiding boundaries. Other aspect-oriented programming systems that support method interception, such as Hyper/J [19] and ComposeJ [23], share the issue. Even recent proposals describing module systems for AOP allow these kinds of violations [13, 6].

Clearly the programmer of the assurance aspect could have written the aspect to be more robust to this kind of change. However, the whole point of an encapsulation system is to protect the programmer from violating information hiding boundaries. In the rest of this paper, we explore a proposed module system that is able to enforce information hiding, while preserving much of the expressiveness of existing aspect-oriented programming systems.

3.3 Overview

We propose Open Modules, a new module system for aspect-oriented programs that is intended to be *open* to aspect-oriented extension but *modular* in that the implementation details of a module are hidden. The goals of openness and modularity are in tension, and so we try to achieve a compromise between them.

The principle behind the design of Open Modules is that interfaces should mediate the interaction between the implementation of a module and its clients, even in the presence of aspects. Our system can capture crosscutting concerns in much the same way as previous aspect-oriented programming systems; the only difference is that some pointcuts may have to be moved from the aspect code to the interface of the module being advised.

For example, the assurance aspect in Figure 6 would be prohibited by our system as written, because the aspect's pointcut potentially includes calls that are within the private implementation of the `shape` package. However, the aspect could be re-written in one of two ways to be compatible with Open Modules.

In the first solution, the pointcut in the aspect would additionally specify that it captures only calls from *outside* the `shape` package:

```
pointcut moves():
    call(void shape.Shape+.moveBy(..))
    && !within(shape.*);
```

This solution fits with Open Modules because it advises only incoming calls to the interface of the package; the aspect is decoupled from the implementation, permitting implementation changes like the one discussed in the previous subsection.

In the second solution, the pointcut in the aspect would be moved to the `shape` module, and then referenced by the aspect:

```
after(): shape.Shape.moves() { ... }
```

In this case, the pointcut becomes part of the interface of the `shape` package, again decoupling the aspect from the package's implementation. If that implementation changes, the maintainer of the module has the responsibility to maintain the semantics of the pointcut so that external aspects are unaffected by the change.

This second solution, called *pointcut interfaces*, was originally proposed by Gudmundson and Kiczales as an engineering technique that can ease software evolution by decoupling an aspect from the code that it advises [8]. It is also related to the Demeter project's use of *traversal strategies* to isolate an aspect from the code that it advises [16].

We now provide a more technical definition for Open Modules, which can be used to distinguish our contribution from previous work:

Definition [Open Modules]: *A module system that:*

- *allows external aspects to advise external calls to functions in the interface of a module*
- *allows external aspects to advise pointcuts in the interface of a module*
- *does not allow external aspects to advise calls from within a module to other functions within the module (including exported functions).*

3.4 Expressiveness

Like the Gudmundson and Kiczales proposal on which they are based [8], Open Modules sacrifice some amount of *obliviousness* [7] in order to support better information hiding. Base code is not completely oblivious to aspects, because the author of a module must expose relevant internal events in pointcuts so that aspects can advise them[1]. However, our design still preserves important cases of obliviousness:

- While a module can expose interesting implementation events in pointcuts, it is oblivious to which aspects might be interested in those events.
- Pointcuts in the interface of a module can be defined *non-invasively* with respect to the rest of the module's implementation, using the same pointcut operations available in other AOP languages.
- A module is completely oblivious to aspects that only advise external calls to its interface.

A possible concern is that the strategy of adding a pointcut to the interface of a base module may be impossible if the source code for that module cannot be changed. In this case, the modularity benefits of Open Modules can be achieved with environmental support for associating an external pointcut with the base module. If the base module is updated, the maintainer of the pointcut is responsible for rechecking the pointcut to ensure that its semantics have not been invalidated by the changes to the base module.

Experiment. In a companion paper, we performed a micro-experiment applying the ideas of Open Modules to SpaceWar, a small demonstration application distributed with AspectJ. The experiment was far too small to provide definitive results. However, we found that Open Modules support nearly all of the aspects in this program with no changes or only minor changes to the code [2].

The only concern our system could not handle was an extremely invasive debugging aspect. Debugging is an inherently non-modular activity, so we view it as a positive sign that our module system does not support it. In a practical system, debugging can be supported either through external tools, or through a compiler flag that makes an exception to the encapsulation rules during debugging activity.

[1]We note that many in the AOP community feel "obliviousness" is too strong a term, preferring a notion of "non-invasiveness" that is compatible with our proposal. See for example posts to the aosd-discuss mailing list by Dean Wempler and Gregor Kiczales in August 2003, available at aosd.net.

Names	n	::=	$\ldots \mid m.x$
Declarations	d	::=	$\ldots \mid \texttt{structure}\ x = m\ d$
Modules	m	::=	n
		$\mid$	$\texttt{struct}\ d\ \texttt{end}$
		$\mid$	$m\ \texttt{:>}\ \sigma$
		$\mid$	$\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m$
		$\mid$	$m_1\ m_2$
Types	τ, σ	::=	$\ldots \mid \texttt{sig}\ \beta\ \texttt{end}$
Decl. values	d_v	::=	$\ldots \mid \texttt{structure}\ x = \square\ d$
Module values	m_v	::=	$\texttt{struct}\ d_v\ \texttt{end}$
		$\mid$	$\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m$
Contexts	C	::=	$\ldots \mid \texttt{structure}\ x = \square\ d$
		$\mid$	$\texttt{structure}\ x \equiv m_v\ \square$
		$\mid$	$\texttt{struct}\ \square\ \texttt{end} \mid \square\ \texttt{:>}\ \sigma$
		$\mid$	$\square\ m_2 \mid m_v\ \square$

Figure 7: Module System Syntax, Values, and Contexts

Comparison to non-AOP techniques. One way to evaluate the expressiveness of Open Modules is to compare them to non-AOP alternatives. One alternative is using wrappers instead of aspects to intercept the incoming calls to a module, and using callbacks instead of pointcuts in the module's interface. The aspect-oriented nature of Open Modules provides several advantages over the wrapper and callback solution:

- Open Modules are compatible with the *quantification* [7] constructs of languages like AspectJ, so that many functions can be advised with a single declaration. Implementing similar functionality with conventional wrappers–which do not support quantification–is far more tedious because a wrapper must be explicitly applied to each function.
- In Open Modules, a single, locally-defined aspect can implement a crosscutting concern by non-locally extending the interface of a number of modules. Wrappers cannot capture these concerns in a modular way, because each target module must be individually wrapped.
- Callbacks are invasive with respect to the implementation of a module because the implementation must explicitly invoke the callback at the appropriate points. In contrast, pointcut interfaces are non-invasive in that the pointcut is defined orthogonally to the rest of the module's implementation, thus providing better support for separation of concerns.

```
structure Cache =
  functor(X : sig f : pc(int->int) end) =>
    struct
        around X.f(x:int) = ...
            (* same definition as before *)
    end

structure Math = struct
    val fib = fn x:int => 1
    around call(fib) (x:int) =
        if (x > 2)
            then fib(x-1) + fib(x-2)
            else proceed x

    structure cacheFib =
        Cache (struct
                   pointcut f = call(fib)
               end)
end :> sig
  fib : int->int
end
```

Figure 8: Fibonacci with Open Modules

These advantages illustrate how the quantification and non-invasive extension provided by Open Modules distinguish our proposal from solutions that do not use aspects [7].

4. Formalization of Open Modules

We now extend `TinyAspect` to model Open Modules. Our module system is modeled closely after that of ML, providing a familiar concrete syntax and benefiting from the design of an already advanced module system.

Figure 7 shows the new syntax for modules. Names include both simple variables x and qualified names $m.x$, where m is a module expression. Declarations can include structure bindings, and types are extended with module signatures of the form `sig` β `end`, where β is the list of variable to type bindings in the module signature.

First-order module expressions include a name, a `struct` with a list of declarations, and an expression m `:>` σ that seals a module with a signature, hiding elements not listed in the signature. The expression `functor`$(x{:}\sigma)$ `=>` m describes a functor that takes a module x with signature σ as an argument, and returns the module m which may depend on x. Functor application is written like function application, using the form $m_1\ m_2$.

Our module system does not include abstract types, and so the abstraction property we enforce is one of implementation independence, not representation independence. The underlying problem is the same in both cases: external aspects should not be able to observe the internal behavior of module functions. Thus, we conjecture that our solution to the implementation independence problem will also enforce representation independence once abstract types are added in standard ways [15].

4.1 Fibonacci Revisited

Figure 8 shows how a more reusable caching aspect could be defined using functors. The `Cache` functor accepts a module that has a single element `f` that is a pointcut of calls to some function with signature `int->int`. The `around` advice then advises the pointcut from the argument module `X`.

The `fib` function is now encapsulated inside the `Math` module. The module implements caching by instantiating the `Cache` module with a structure that binds the pointcut `f` to calls to `fib`. Finally, the `Math` module is sealed with a signature that exposes only the `fib` function to clients.

```
structure shape = struct
    val createShape = fn ...
    val moveBy = fn ...
    val animate = fn ...
    ...
    pointcut moves = call(moveBy)
end :> sig
    createShape : Description -> Shape
    moveBy      : (Shape,Location) -> unit
    animate     : (Shape,Path) -> unit
    ...
    moves       : pc((Shape,Location)->unit)
end
```

Figure 9: A shape library that exposes a position change pointcut

4.2 Sealing

Our module sealing operation has an effect both at the type system level and at the operational level. At the type level, it hides all members of a module that are not in the signature σ–in this respect, it is similar to sealing in ML's module system. However, sealing also has an operational effect, hiding internal calls within the module so that clients cannot advise them unless the module explicitly exports the corresponding pointcut.

For example, in Figure 8, clients of the `Math` module would not be able to tell whether or not caching had been applied, even if they placed advice on `Math.fib`. Because `Math` has been sealed, external advice to `Math.fib` would only be invoked on external calls to the function, not on internal, recursive calls. This ensures that clients cannot be affected if the implementation of the module is changed, for example, by adding or removing caching.

The strategy used to protect information hiding in our formal system is slightly different from the informal strategy presented in Section 3. There we were assuming the semantics of AspectJ, and so in order to avoid capturing internal calls we had to explicitly say `!within(shape.*)` in the pointcut. In the formal system, we provide a cleaner solution, where once a module is sealed, externally defined pointcuts *automatically* include the limitation to external calls.

4.3 Exposing Semantic Events with Pointcuts

Figure 9 shows how the shape example described above could be modeled in `TinyAspect`. Clients of the shape library cannot advise internal functions, because the module is sealed. To allow clients to observe internal but semantically important events like the motion of animated shapes, the module exposes these events in its signature as the `moves` pointcut. Clients can advise this pointcut without depending on the internals of the shape module. If the module's implementation is later changed, the `moves` pointcut must

$$\frac{bind\ x \equiv v \in d_v}{(\eta,\ \texttt{struct}\ d_v\ \texttt{end}.x) \mapsto (\eta,\ v)}\ \textit{r-path}$$

$$\frac{}{\begin{array}{c}(\eta,\ \texttt{structure}\ x = m_v\ \ d) \mapsto \\ (\eta,\ \texttt{structure}\ x \equiv m_v\ \ \{m_v/x\}d)\end{array}}\ \textit{r-structure}$$

$$\frac{}{(\eta,\ (\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m_1)\ m_2) \mapsto (\eta,\ \{m_2/x\}m_1)}\ \textit{r-fapp}$$

$$\frac{seal(\eta, d_v, \beta) = (\eta', d_{seal})}{\begin{array}{c}(\eta,\ \texttt{struct}\ d_v\ \texttt{end}\ \texttt{:>}\ \texttt{sig}\ \beta\ \texttt{end}) \\ \mapsto (\eta',\ \texttt{struct}\ d_{seal}\ \texttt{end})\end{array}}\ \textit{r-seal}$$

$$\frac{}{seal(\eta, \bullet, \bullet) = (\eta, \bullet)}\ \textit{s-empty}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{seal(\eta, bind\ x \equiv v\ \ d, \beta) = (\eta', d')}\ \textit{s-omit}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d') \quad \eta'' = [\ell \mapsto v]\ \eta' \quad \ell \notin domain(\eta')}{seal(\eta, \texttt{val}\ x \equiv v\ \ d, (x{:}\tau, \beta)) = (\eta'', \texttt{val}\ x \equiv \ell\ \ d')}\ \textit{s-v}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{\begin{array}{c}seal(\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d, (x{:}\texttt{pc}(\tau), \beta)) \\ = (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d')\end{array}}\ \textit{s-p}$$

$$\frac{seal(\eta, d_s, \beta_s) = (\eta', d_s') \quad seal(\eta', d, \beta) = (\eta'', d')}{\begin{array}{c}seal(\eta, \texttt{structure}\ x \equiv \texttt{struct}\ d_s\ \texttt{end}\ \ d, \\ (x{:}\texttt{sig}\ \beta_s\ \texttt{end}, \beta)) \\ = (\eta'', \texttt{structure}\ x \equiv \texttt{struct}\ d_s'\ \texttt{end}\ \ d')\end{array}}\ \textit{s-s}$$

$$\frac{seal(\eta, d, \beta) = (\eta', d')}{\begin{array}{c}seal(\eta, \texttt{structure}\ x \equiv \texttt{functor}(y{:}\sigma_y)\ \texttt{=>}\ m\ \ d, (x{:}\sigma, \beta)) \\ = (\eta', \texttt{structure}\ x \equiv \texttt{functor}(y{:}\sigma_y)\ \texttt{=>}\ m\ \ d')\end{array}}\ \textit{s-f}$$

Figure 10: Module System Operational Semantics

also be changed to ensure that client aspects are not affected.

Thus, sealing enforces the abstraction boundary between a module and its clients, allowing programmers to reason about and change them independently. However, our system still allows a module to export semantically important internal events, allowing clients to extend or observe the module's behavior in a principled way.

4.4 Open Modules Operational Semantics

Figure 10 shows the operational semantics for Open Modules. In the rules, module values m_v mean either a struct with declaration values d_v or a functor. The path lookup rule finds the selected binding within the declarations of the module. We assume that bound names are distinct in this rule; it is easy to ensure this by renaming variables appropriately. Because modules cannot be advised, there is no need to create labels for structure declarations; we can just substitute the structure value for the variable in subsequent declarations. The rule for functor application also uses substitution.

The rule for sealing uses an auxiliary judgment, *seal*, to generate a fresh set of labels for the bindings exposed in the signature. This fresh set of labels insures that clients can affect external calls to module functions by advising the new labels, but cannot advise calls that are internal to the sealed module.

At the bottom of the diagram are the rules defining the sealing operation. The operation accepts an old environment η, a list of declarations d, and the sealing declaration signature β. The operation computes a new environment η' and new list of declarations d'. The rules are structured according to the first declaration in the list; each rule handles the first declaration and appeals recursively to the definition of sealing to handle the remaining declarations.

An empty list of declarations can be sealed with the empty signature, resulting in another empty list of declarations and an unchanged environment η. The second rule allows a declaration $bind\ x \equiv v$ (where *bind* represents one of `val`, `pointcut`, or `struct`) to be omitted from the signature, so that clients cannot see it at all. The rule for sealing a value declaration generates a fresh label ℓ, maps that to the old value of the variable binding in η, and returns a declaration mapping the variable to ℓ. Client advice to the new label ℓ will affect only external calls, since internal references still refer to the old label which clients cannot change. The rule for pointcuts passes the pointcut value through to clients unchanged, allowing clients to advise the label referred to in the pointcut. Finally, the rules for structure declarations recursively seal any internal struct declarations, but leave functors unchanged.

$$\frac{\Gamma; \Sigma \vdash m : \texttt{sig}\ \beta\ \texttt{end} \quad x{:}\tau \in \beta}{\Gamma; \Sigma \vdash m.x : \tau}\ \textit{t-name}$$

$$\frac{\Gamma; \Sigma \vdash m : \sigma \quad \Gamma, x{:}\sigma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{structure}\ x = m\ \ d : (x{:}\sigma, \beta)}\ \textit{t-structure}$$

$$\frac{\Gamma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{struct}\ d\ \texttt{end} : \texttt{sig}\ \beta\ \texttt{end}}\ \textit{t-struct}$$

$$\frac{\Gamma; \Sigma \vdash m : \sigma_m \quad \sigma_m <: \sigma}{\Gamma; \Sigma \vdash m\ \texttt{:>}\ \sigma : \sigma}\ \textit{t-seal}$$

$$\frac{\Gamma, x{:}\sigma_1; \Sigma \vdash m : \sigma_2}{\Gamma; \Sigma \vdash \texttt{functor}(x{:}\sigma_1)\ \texttt{=>}\ m : \sigma_1 \to \sigma_2}\ \textit{t-functor}$$

$$\frac{\Gamma; \Sigma \vdash m_1 : \sigma_1 \to \sigma \quad \Gamma; \Sigma \vdash m_2 : \sigma_2 \quad \sigma_2 <: \sigma_1}{\Gamma; \Sigma \vdash m_1\ m_2 : \sigma}\ \textit{t-fapp}$$

Figure 11: Open Modules Typechecking

4.5 Typechecking

The typechecking rules, shown in Figure 11, are largely standard. Qualified names are typed based on the binding in the signature of the module m. Structure bindings are given a declaration signature based on the signature σ of the bound module. The rule for `struct` simply puts a `sig` wrapper around the declaration signature. The rules for sealing and functor application allow a module to be passed into a context where a supertype of its signature is expected.

Figure 12 shows the definition of signature subtyping. Subtyping is reflexive and transitive. Subtype signatures may have additional bindings, and the signatures of constituent bindings are covariant. Finally, the subtyping rule

$$\frac{}{\sigma <: \sigma} \textit{ sub-reflex}$$

$$\frac{\sigma <: \sigma' \quad \sigma' <: \sigma''}{\sigma <: \sigma''} \textit{ sub-trans}$$

$$\frac{\beta <: \beta'}{\texttt{sig}\ \beta\ \texttt{end} <: \texttt{sig}\ \beta'\ \texttt{end}} \textit{ sub-sig}$$

$$\frac{\beta <: \beta'}{x : \tau, \beta\ <:\ \beta'} \textit{ sub-omit}$$

$$\frac{\beta <: \beta' \quad \tau <: \tau'}{x : \tau, \beta\ <:\ x : \tau', \beta'} \textit{ sub-decl}$$

$$\frac{\sigma_1' <: \sigma_1 \quad \sigma_2 <: \sigma_2'}{\sigma_1 \to \sigma_2\ <:\ \sigma_1' \to \sigma_2'} \textit{ sub-contra}$$

Figure 12: Signature Subtyping

for functor types is contravariant.

4.6 Type Soundness

When extended with Open Modules, `TinyAspect` enjoys the same type soundness property that the base system has. The theorems and proofs are similar, and so we omit them.

5. Abstraction

The example programs in Section 3 are helpful for understanding the benefits of `TinyAspect`'s module system at an intuitive level. However, we would like to be able to point to a concrete property that enables separate reasoning about the clients and implementation of a module.

Reynolds' *abstraction* property [18] fits these requirements in a natural way. Intuitively, the abstraction property states that if two module implementations are semantically equivalent, no client can tell the difference between the two. This property has two important benefits for software engineering. First of all, it enables reasoning about the properties of a module in isolation. For example, if one implementation of a module is known to be correct, we can prove that a second implementation is correct by showing that it is semantically equivalent to the first implementation. Second, the abstraction property ensures that the implementation of a module can be changed to a semantically equivalent one without affecting clients. Thus, the abstraction property helps programmers to more effectively hide information that is likely to change, as suggested in Parnas' classic paper [17].

In `TinyAspect`, we can state the abstraction property as follows. If two modules m and m' are observationally equivalent and have module signature σ, then for all client declarations d that are well-typed assuming that some variable x has type σ, the client behaves identically when executed with either module.

Intuitively, two modules are observationally equivalent if all of the bound functions and values in the module are equivalent. Two functions are equivalent if they always produce equivalent results given equivalent arguments, *even if a client advises other functions exported by the module*. This illustrates the importance of using sealing to limit the scope of client advice. If two modules are sealed, then they can be proved equivalent assuming that clients can only advise the exported pointcuts. In this sense, module sealing enables separate reasoning that would be impossible otherwise.

$$\frac{\Lambda \vdash (\eta, V) \simeq (\eta', V') : T}{\Lambda \vdash (\eta, V) \cong (\eta', V') : T}$$

$$\frac{\begin{array}{c}(\eta_1, E_1) \overset{\Lambda}{\mapsto}{}^{*} (\eta_1', E_1') \qquad (\eta_2, E_2) \overset{\Lambda}{\mapsto}{}^{*} (\eta_2', E_2') \\ \Lambda' \vdash (\eta_1', E_1') \cong (\eta_2', E_2') : T \\ (\Lambda' - \Lambda) \cap domain(\eta_1 \cup \eta_2) = \emptyset\end{array}}{\Lambda \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{\Lambda \vdash (\eta_1, C_1[\eta_1[\ell]\ v_1]) \cong (\eta_2, C_2[\eta_2[\ell]\ v_2]) : T}{\Lambda \vdash (\eta_1, C_1[\ell\ v_1]) \cong (\eta_2, C_2[\ell\ v_2]) : T}$$

$$\frac{(\eta, E)\ and\ (\eta', E')\ diverge\ following\ the\ rules\ above}{\Lambda \vdash (\eta, E) \cong (\eta', E') : T}$$

Figure 14: `TinyAspect` Observational Equivalence for Expressions

5.1 Formalizing Abstraction

We can define abstraction formally using judgments for observational equivalence of values, written $\Lambda \vdash (\eta, V) \simeq (\eta', V') : T$ and read, "In the context of a set of visible labels Λ, value V in environment η is observationally equivalent to value V' in environment η' at type T. A similar judgment of the form $\Lambda \vdash (\eta, E) \cong (\eta', E') : T$ is used for observationally equivalent expressions. The judgments depend on the set of labels Λ that are visible and thus capable of being advised; in order for two values to be observationally equivalent, they must use these labels in the same way.

The main rules for observational equivalence of values are defined in Figure 13. Most of the rules are straightforward–for example, there is only one unit value, so all values of type unit are equivalent.

The most interesting rule is the one for function values. Two function values are equivalent if for any observationally equivalent argument values v_1 and v_2, they produce equivalent results. Note that the rule for observational equivalence for function values includes both syntactic functions and labels that denote functions. A similar rule is used for observational equivalence of functors.

Two `val` declarations are equivalent if they bind the same variable to the same label (since labels are generated fresh for each declaration we can always choose them to be equal when we are proving equivalence), and the label is equivalent in the two environments η and η'. Since the label exposed by the `val` declaration is visible, it must be in Λ. Pointcut and structure declarations just check the equality of their components. All three declaration forms ensure that subsequent declarations are also equivalent; we assume that the empty declaration $\bullet$ is equivalent to itself. Finally, two first-order modules are equivalent if the declarations inside them are also equivalent.

Figure 14 shows the rules for observational equivalence of expressions. Two expressions are equivalent if they are

$$\Lambda \vdash (\eta_1, v) \simeq (\eta_2, v) : \texttt{unit}$$

$$\Lambda \vdash (\eta_1, v_1) \simeq (\eta_2, v_2) : \tau' \to \tau \quad \text{iff} \quad \begin{array}{l}\text{for all } v_1', v_2' \\ \text{such that } \Lambda \vdash (\eta_1, v_1') \simeq (\eta_2, v_2') : \tau' \\ \text{we have } \Lambda \vdash (\eta_1, v_1\ v_1') \cong (\eta_2, v_2\ v_2') : \tau\end{array}$$

$$\Lambda \vdash (\eta, \texttt{val}\ x \equiv \ell\ \ d_v) \simeq (\eta', \texttt{val}\ x \equiv \ell\ \ d_v') : (x{:}\tau, \beta) \quad \text{iff} \quad \begin{array}{l}\ell \in \Lambda, \Lambda \vdash (\eta, \ell) \simeq (\eta', \ell) : \tau, \\ \text{and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta\end{array}$$

$$\begin{array}{c}\Lambda \vdash (\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v) \simeq \\ (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v') : (x{:}\texttt{pc}(\tau), \beta)\end{array} \quad \text{iff} \quad \ell \in \Lambda \text{ and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta$$

$$\begin{array}{c}\Lambda \vdash (\eta, \texttt{structure}\ x \equiv m_v\ \ d_v) \simeq \\ (\eta', \texttt{structure}\ x \equiv m_v'\ \ d_v') : (x{:}\sigma, \beta)\end{array} \quad \text{iff} \quad \begin{array}{l}\Lambda \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma, \\ \text{and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta\end{array}$$

$$\Lambda \vdash (\eta, \texttt{struct}\ d_v\ \texttt{end}) \simeq (\eta', \texttt{struct}\ d_v'\ \texttt{end}) : \texttt{sig}\ \beta\ \texttt{end} \quad \text{iff} \quad \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta$$

$$\Lambda \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma_1 \to \sigma_2 \quad \text{iff} \quad \begin{array}{l}\text{for all } m_v^1, m_v^2 \text{ such that } \Lambda \vdash (\eta, m_v^1) \simeq (\eta', m_v^2) : \sigma_1 \\ \text{we have } \Lambda \vdash (\eta_1, m_v\ m_v^1) \cong (\eta_2, m_v'\ m_v^2) : \sigma_2\end{array}$$

Figure 13: `TinyAspect` Observational Equivalence for Values

equivalent values. Otherwise, the expressions must be *bisimilar* with respect to the set of labels in Λ. That is, they must look up the same sequence of labels in Λ while either diverging or reducing to observationally equivalent values (since client aspects can use advice to observe lookups to labels in Λ).

We formalize this with three rules. The first allows two expressions to take any number of steps that does not include looking up a label in Λ (using the evaluation relation $\overset{\Lambda}{\mapsto}{}^*$ which is identical to $\mapsto^*$ except that the rule *r-lookup* may not be applied to any label in Λ). The second allows two expressions to lookup the same label in Λ. The third allows computation to diverge according to the first two rules, rather than terminating with a value.

Now that we have defined observational equivalence, we can state the abstraction theorem:

Theorem 3 (Abstraction)
If $\Lambda \vdash (\bullet, m_v) \cong (\bullet, m_v') : \sigma$*, then for all* d *such that* $x{:}\sigma; \bullet \vdash d : \beta$ *we have* $\Lambda \vdash (\bullet, \texttt{structure}\ x = m_v\ \ d) \cong (\bullet, \texttt{structure}\ x = m_v'\ \ d) : (x{:}\sigma, \beta)$

For space reasons, we give only a brief sketch of the proof of abstraction. More details are available in a companion technical report [1]. The proof uses a structural congruence property: the expressions are structurally equal except for closed values, which are observationally equivalent. A key lemma states that structural congruence is preserved by reduction.

We then observe that the two programs being compared are initially structurally congruent. By the lemma, they either remain structurally congruent indefinitely, corresponding to the divergence case of observational equivalence, or else they eventually reduce to values which are observationally equivalent.

5.2 Applying Abstraction

The abstraction theorem can be used to show that two different implementations of a module are equivalent and thus interchangeable. For example, Figure 15 shows two definitions of the Fibonacci function. The first one uses recursion

```
structure Fib1 = struct
    val fib = fn x:int => 1
    around call(fib) (x:int) =
        if (x > 2)
            then fib(x-1) + fib(x-2)
            else proceed x
end :> sig
    fib : int->int
end

structure Fib2 = struct
    val helper = fn x:int => 1
    around call(helper) (x:int) =
        if (x > 2)
            then helper(x-1) + helper(x-2)
            else proceed x
    val fib = fn x:int => helper x
end :> sig
    fib : int->int
end
```

Figure 15: Two equivalent modules that define the Fibonacci function

directly to compute the result, while the second one invokes a helper function. Since we have sealed both modules, it is easy to prove that they are equivalent. Clients can only advise the fresh label exported by the sealed modules, which doesn't affect the internal semantics of the module at all. Therefore, we can prove that the modules are equivalent by showing that the `fib` functions always return the same value when passed the same argument. A simple proof by induction on the argument value will suffice.

However, if we did not use `TinyAspect`'s sealing operation on these modules but instead used a more conventional module system to hide the `helper` function in `Fib2`, we would be unable to prove the modules equivalent. In this case, a client could advice `fib`, which would capture the recursive calls in module `Fib1` but not in module `Fib2`. Thus, the client's behavior would depend on the module's imple-

mentation.

This example shows that the properties of the module sealing operation are crucial for formal reasoning about aspect-oriented systems. Sealing is also important for more informal kinds of reasoning, for example allowing engineers to change the internals of a module with some assurance that clients will not be affected.

The Fibonacci example is simplistic in that it does not export any pointcuts to clients. However, similar equivalence properties can be proven in the presence of pointcuts, if it can be shown that two modules always treat their exported pointcut labels in an identical way, as defined by the observational equivalence relation.

6. Related Work

Formal Models. Walker et al. model aspects using an expression-oriented functional language that includes the lambda calculus, labeled join points, and advice [21]. They show that their model is type-safe, but they model `around` advice using a low-level exception construct and so their soundness theorem includes the possibility that the program could terminate with an uncaught exception error. `TinyAspect` guarantees both type safety and a lack of runtime errors because it models advice with high-level constructs similar to those in existing aspect-oriented programming languages. In addition, the declarative, source-level nature of `TinyAspect` allows us to easily explore modularity and prove an abstraction result.

Jagadeesan et al. describe an object-oriented aspect calculus modeling many of the features of AspectJ [10]. Their formal model is much richer than ours, capturing complex pointcuts and different forms of advice in a rich subset of Java. `TinyAspect` is intentionally much more minimal than their aspect calculus, so that it is easy to investigate language extensions such as a module system and prove properties such as abstraction.

In other work on formal systems for aspect-oriented programming, Lämmel provides a big-step semantics for a method-call interception extension to object-oriented languages [12]. Wand et al. give an untyped, denotational semantics for advice advice and dynamic join points [22]. Masuhara and Kiczales describe a general model of crosscutting structure, using implementations in Scheme to give semantics to the model [14]. Tucker and Krishnamurthi show how scoped continuation marks can be used in untyped higher-order functional languages to provide static and dynamic aspects [20].

Aspects and Modules. Dantas and Walker are currently extending the calculus of Walker et al. to support a module system [6]. Their type system includes a novel feature for controlling whether advice can read or change the arguments and results of advised functions. In their design, pointcuts are first-class, providing more flexibility compared to the second-class pointcuts in `TinyAspect`. This design choice breaks abstraction and thus separate reasoning, however, because it means that a pointcut can escape from a module even if it is not explicitly exported in the module's interface. In their system, functions can only be advised if this is planned in advance; in contrast, `TinyAspect` allows advice on all function declarations, providing unplanned extensibility without compromising abstraction.

Lieberherr et al. describe Aspectual Collaborations, a construct that allows programmers to write aspects and code in separate modules and then compose them together into a third module [13]. Since they propose a full aspect-oriented language, their system is much richer and more flexible than ours, but its semantics are not formally defined. Their module system does not encapsulate internal calls to exported functions, and thus does not enforce the abstraction property.

Other researchers have studies ways of achieving modular reasoning without the use of explicit module systems. For example, the Eclipse plugin for AspectJ includes a view showing which aspects affect each line of source code. Clifton and Leavens propose engineering techniques that reduce dependencies between concerns in aspect-oriented code [4].

Our module system is based on that of standard ML [15]. `TinyAspect`'s sealing construct is similar to the freeze operator in the module calculus of Ancona and Zucca, which closes a module to extension [3].

The name Open Modules indicates that modules are open to advice on functions and pointcuts exposed in their interface. Open Classes is a related term indicating that classes are open to the addition of new methods [5].

7. Future Work

In future work, we plan to extend the module system presented here to support recursive modules and abstract data types, as well as supporting modules that can be loaded and instantiated at run time. We would like to extend the base language with polymorphism, references, and objects; enforcing abstraction in the context of these features is an open problem. Based on this foundation, we intend to design and implement a user-level language with aspect-oriented features, including richer mechanisms for pointcuts and advice.

8. Conclusion

This paper described `TinyAspect`, a minimal core language for reasoning about aspect-oriented programming systems. `TinyAspect` is a source-level language that supports declarative aspects. We have given a small-step operational semantics to the language and proven that its type system is sound. We have described a proposed module system for aspects, formalized the module system as an extension to `TinyAspect`, and proved that the module system enforces abstraction. Abstraction ensures that clients cannot affect or depend on the internal implementation details of a module. As a result, programmers can both separate concerns in their code and reason about those concerns separately.

9. Acknowledgments

I thank Ralf Lämmel, Karl Lieberherr, David Walker, Curtis Clifton, Derek Dreyer, Todd Millstein, Robert Harper, and the anonymous reviewers for comments and conversations on an earlier draft of this paper.

10. REFERENCES

[1] J. Aldrich. Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming. Carnegie Mellon Technical Report CMU-ISRI-04-108, available at http://www.cs.cmu.edu/~aldrich/aosd/, March 2004.

[2] J. Aldrich. Open Modules: Reconciling Extensibility and Information Hiding. In *AOSD workshop on Software*

Engineering Properties of Languages for Aspect Technologies (SPLAT '04), March 2004.

[3] D. Ancona and E. Zucca. A Calculus of Module Systems. *Journal of Functional Programming*, 12(2):91–132, March 2002.

[4] C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages*, April 2002.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Object-Oriented Programming Systems, Languages, and Applications*, October 2000.

[6] D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Unpublished manuscript, 2003.

[7] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Advanced Separation of Concerns*, October 2000.

[8] S. Gudmundson and G. Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Advanced Separation of Concerns*, July 2001.

[9] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[10] R. Jagadeesan, A. Jeffrey, and J. Riely. An Untyped Calculus of Aspect-Oriented Programs. In *European Conference on Object-Oriented Programming*, July 2003.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *European Conference on Object-Oriented Programming*, June 2001.

[12] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Aspect-Oriented Software Development*, Apr. 2002.

[13] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[14] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *European Conference on Object-Oriented Programming*, July 2003.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, Massachusetts, 1997.

[16] D. Orleans and K. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, September 2001.

[17] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[18] J. C. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Information Processing*, 1983.

[19] P. Tarr, H. Ossher, W. Herrison, and S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Interntional Conference on Software Engineering*, May 1999.

[20] D. B. Tucker and S. Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Aspect-Oriented Software Development*, March 2003.

[21] D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *International Conference on Functional Programming*, 2003.

[22] M. Wand, G. Kiczales, and C. Dutchyn. A Semantics for Advice and Dynamic Join Points in Aspect-Oriented Programming. *Transactions on Programming Languages and Systems*, To Appear 2003.

[23] J. C. Wichman. ComposeJ - The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Masters Thesis, University of Twente, 1999.