Open Modules: Modular Reasoning about Advice

Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15213, USA,
jonathan.aldrich@cs.cmu.edu

Abstract. Advice is a mechanism used by advanced object-oriented and aspect-oriented programming languages to augment the behavior of methods in a program. Advice can help to make programs more modular by separating crosscutting concerns more effectively, but it also challenges existing ideas about modularity and separate development.

We study this challenge using a new, simple formal model for advice as it appears in languages like AspectJ. We then add a module system designed to leave program functionality as open to extension through advice as possible, while still enabling separate reasoning about the code within a module. Our system, Open Modules, can either be used directly to facilitate separate, component-based development, or can be viewed as a model of the features that certain AOP IDEs provide. We define a formal system for reasoning about the observational equivalence of programs under advice, which can be used to show that clients are unaffected by semantics-preserving changes to a module's implementation. Our model yields insights into the nature of modularity in the presence of advice, provides a mechanism for enforceable contracts between component providers and clients in this setting, and suggests improvements to current AOP IDEs.

1 Modularity and Advice

The Common Lisp Object System introduced a construct called *advice*, which allows a developer to externally augment the behavior of a method [2]. Advice comes in at least three flavors: *before* advice is run before the execution of a method body, *around* advice wraps a method body, and *after* advice runs after the method body. In general, advice can view or change the parameters or result of a method, or even control whether the method body is executed, allowing a rich set of adaptations to be implemented through this mechanism.

This paper examines advice in the context of Aspect-Oriented Programming (AOP), the most widely-used application of advice today [12]. The goal of AOP is to modularize concerns that crosscut the primary decomposition of a software system. AOP systems allow developers to modularize these *crosscutting concerns* within a single, locally-defined module, using an advice mechanism to allow definitions in that module to affect methods defined elsewhere in the system.

Although AOP in general and advice in particular provide many benefits to reasoning about concerns that are scattered and tangled throughout the code in conventional systems, questions have been raised about the ability to reason about and evolve code that is subject to advice. Advice appears to make reasoning about the effect of calling a method more challenging, for example, because it can intercept that call and change its semantics. In turn, this makes evolving AOP systems without tool support error-prone, because seemingly innocuous changes to base code could break the functionality of an aspect that advises that code.

For example, an important issue in separate development is ensuring that improvements and bug-fixes to a third-party component can be integrated into an application without breaking the application. Unfortunately, however, because the developer of a component does not know how is deployed, any change she makes could potentially break fragile pointcuts in a client [13] Although we could solve the problem by prohibiting all advice to third-party components, we would prefer a compromise that answers the research question:

1. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still allowing the component to be changed in meaningful ways without affecting clients?*

Another important issue is protecting the internal invariants of component implementations in the presence of advice. For example, consider the Java standard library, which is carefully designed to provide a "sandbox" security model for running untrusted code. In general, it is unsafe for a user to load any code that advises the standard library, because the advice could be used by an attacker to bypass the sandbox.

One possible solution to ensuring the security of the Java library is to prohibit all advice to the standard library. Unfortunately, this rule would prohibit many useful applications of advice. A better solution to the problem would allow as much advice as possible, while still preserving the internal invariants of the Java standard library. We thus use our formal model of modularity to address a second research question:

2. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still ensuring correctness properties of the component implementation?*

The research questions above imply a solution that prohibits certain uses of advice in the case of separate development. However, in practice, some applications may only be able to reuse a library or component if they can get around these prohibitions. We think this should be an option for developers, but it should be a conscious choice: a developer should know when she is writing an aspect that may break when a new version of a component is released, and when she is writing an aspect that is resilient to new releases. Similarly, a user uploading code should know whether that code only includes advice that is guaranteed not to violate the Java sandbox, or whether the code contains advice that might violate the sandbox, and therefore ought to be signed by a trusted principal. Our research is aimed at providing developers and users with these informed choices.

1.1 Outline and Contributions

The next section of the paper describes *Open Modules*, a novel module system for advice that provides informal answers to the research questions. Open Modules is the first module system that supports many beneficial uses of advice, while still ensuring that security and other properties of a component implementation are maintained, and verifying that advice to a component will not be affected by behavior-preserving changes to the component's implementation.

We would like to make these answers precise, and to do so, Section 3 proposes `TinyAspect`, a novel formal model of AOP languages that captures a few core advice constructs while omitting complicating details. Section 4 extends `TinyAspect` with Open Modules, precisely defining the semantics and typing rules of the system.

Section 5 describes a formal system for reasoning about the observational equivalence of modules in the presence of advice. This is the first result to show that complete behavioral equivalence reasoning can be done on a module-by-module basis in a system with advice, providing a precise answer to research question 2. It also precisely defines what changes can be made to a component without affecting clients, answering research questions 1.

Section 6 discusses lessons learned from our formal model. Section 7 describes related work, and Section 8 concludes.

2 Open Modules

We propose Open Modules, a new module system for languages with advice that is intended to be *open* to extension with advice but *modular* in that the implementation details of a module are hidden. The goals of openness and modularity are in tension (at least in the case of separate development), and so we try to achieve a compromise between them.

In AOP systems, advice is used as a way to reach across module boundaries to capture crosscutting concerns. We propose to adopt the same advice constructs, but limit them so that they respect module boundaries. In order to capture concerns that crosscut the boundary of a module, we use AOP's *pointcut* abstraction to represent abstract sets of events that external modules may be interested in advising. As suggested by Gudmundson and Kiczales [9], exported pointcuts form a contract between a module and its client aspects, allowing the module to be evolved independently of its clients so long as the contract is preserved.

Figure 1 shows a conceptual view of Open Modules. Like ordinary module systems, open modules export a list of data structures and functions such as `moveBy` and `animate`. In addition, however, open modules can export pointcuts denoting internal semantic events. For example, the `moves` pointcut in Figure 1

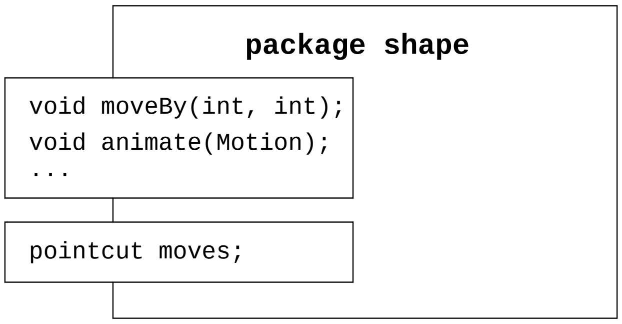

Fig. 1. A conceptual view of Open Modules. The `shape` module exports two functions and a pointcut. Clients can place advice on external calls to the exported functions, or on the exported pointcut, but not on calls that are internal to the module.

is triggered whenever a shape moves. Since a shape could move multiple times during execution of the `animate` function, clients interested in fine-grained motion information would want to use this pointcut rather than just placing advice on calls to `animate`.

By exporting a pointcut, the module's maintainer is making a promise to maintain the semantics of that pointcut as the module's implementation evolves, just as the maintainer must maintain the semantics of the module's exported functions.

Open Modules are "open" in two respects. First, their interfaces are open to advice; all calls to interface functions from outside the module can be advised by clients. Second, clients can advise exported pointcuts.

On the other hand, open modules encapsulate the internal implementation details of a module. As usual with module systems, functions that are not exported in the module's public interface cannot be called from outside the module. In addition, in the case of separate development, calls between functions within the module cannot be advised from the outside—even if the called function is in the public interface of the module. For example, a client could place advice on external calls to `moveBy`, but not calls to `moveBy` from another function within the `shape` module.

In concurrent work, Kiczales and Mezini propose the notion of Aspect-Aware Interfaces, which are ordinary functional interfaces augmented with information about the advice that applies to a module. They point out that in a local development setting, analysis tools such as the AspectJ plugin for Eclipse (AJDT) [4] can compute aspect-aware interfaces automatically given whole-program information. Their work shows that in the case of local development and tool support, the benefits of Open Modules can be attained with no restrictions on the use of aspects. Instead, whenever an aspect that depends on internal calls is defined, the tools simply add a new pointcut to the module's aspect-aware interface, so that the new aspect conforms to the rules of Open Modules.

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

Definition [Open Modules]: *Open Modules describes a module system that:*

- *allows external advice to interactions between a module and the outside world (including external calls to functions in the interface of a module)*
- *allows external advice to pointcuts in the interface of a module*
- *does not allow external modules to directly advise internal events within the module, such as calls from within a module to other functions within the module (including calls to exported functions).*

3 Formally Modeling Advice

In order to reason formally and precisely about modularity, we need a formal model of advice. The most attractive models for this purpose are based on small-step operational semantics, which provide a very simple and direct formalization and are amenable to standard 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]. Their model is very rich and is thus ideal for specifying the semantics of a full language like AspectJ [11]. However, we would like to define and prove the soundness of a strong equivalence reasoning framework for the language, and doing so would be prohibitively difficult in such a complex model.

Walker et al. propose a much simpler formal model incorporating just the lambda calculus, advice, and labeled hooks that describe where advice may apply [19]. As a foundational calculus, their model is ideal for studying compilation strategies for AOP languages. However, because their model is low-level, it lacks some essential characteristics of advice in AOP, including the obliviousness property since advice applies to explicit labels [8]. The low-level nature of their language also means that certain properties of source-level languages like AspectJ—including the modularity properties we study—do not hold in their calculus. Thus, previous small-step operational models of aspects are inappropriate for our purposes.

3.1 TinyAspect

We have developed a new functional core language for aspect-oriented programming called `TinyAspect`. The `TinyAspect` language is intentionally small, making it feasible to rigorously prove strong properties such as the soundness of logical equivalence in Section 5. Although `TinyAspect` leaves out many features of full languages, we directly model advice constructs similar to those in AspectJ. Thus our model retains the declarative nature and oblivious properties of advice in existing AOP languages, helping to ensure that techniques developed in our model can be extended to full languages.

Names	$n ::= x$
Expressions	$e ::= n \mid \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e \mid e_1\ e_2 \mid \texttt{()}$
Declarations	$d ::= \bullet \mid \texttt{val}\ x = e\ \ d \mid \texttt{pointcut}\ x = p\ \ d \mid \texttt{around}\ p(x{:}\tau) = e\ \ d$
Pointcuts	$p ::= n \mid \texttt{call}(n)$
General exp.	$E ::= e \mid d \mid p$
Types	$\tau ::= \texttt{unit} \mid \tau_1 \rightarrow \tau_2$
Decl. Types	$\beta ::= \bullet \mid x{:}\tau, \beta \mid x{:}\pi, \beta$
Pcut. types	$\pi ::= \texttt{pc}(\tau_1 \rightarrow \tau_2)$
General types	$T ::= \tau \mid \beta \mid \pi$

Fig. 2. `TinyAspect` Source Syntax

Because our paper is focused on studying modular reasoning for advice, we omit many of the powerful pointcut constructs of AOP languages like AspectJ. We do include simple pointcuts representing calls to a particular function in order to show how pointcuts in the interface of a module can contribute to separate reasoning in the presence of advice. Our system can easily be extended to other forms of static pointcuts, but an extension to a dynamic pointcut language with constructs like `cflow` [11] is beyond the scope of this work.

Figure 2 shows the syntax of `TinyAspect`. Our syntax is modeled after ML [18]. Names in `TinyAspect` are simple identifiers. 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, and constructs like let can be encoded using lambdas. Since these constructs are orthogonal to aspects, we omit them for simplicity's sake.

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 declaration n, while a pointcut of the form n is just an alias for a previous pointcut declaration n. 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.

```
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
```

Fig. 3. The Fibonacci function written in `TinyAspect`, along with an aspect that caches calls to `fib`.

`TinyAspect` types τ include the `unit` type and function types of the form $\tau_1 \rightarrow \tau_2$. We syntactically distinguish pointcut types π and declaration types β in order to enforce the second-class nature of these constructs (e.g., they cannot be computed by functions, nor can they be used to simulate fully general references).

3.2 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 3 shows the `TinyAspect` code for the Fibonacci function. Integers, booleans, and if statements have been added to illustrate the example.

`TinyAspect` does not include a fixpoint operator for defining recursion, but advice can express the same thing. In the `fib` function above, we define the base case as an ordinary function definition, returning 1. We then place `around` advice that intercepts calls to `fib` and handles the recursive cases. 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.

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

$$\begin{array}{lrcl} \text{Expression values} & v & ::= & \texttt{()} \mid \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e \mid \ell \\ \text{Declaration values} & d_v & ::= & \bullet \mid \texttt{val}\ x \equiv \ell\ \ d_v \mid \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v \\ \text{Evaluation contexts} & C & ::= & \Box\ e_2 \mid v_1\ \Box \mid \texttt{val}\ x = \Box\ \ d \\ & & \mid & \mathit{bind}\ x \equiv V\ \ \Box \mid \texttt{pointcut}\ x = \Box\ \ d \\ \text{General values} & V & ::= & v \mid d_v \mid \texttt{call}(\ell) \end{array}$$

Fig. 4. `TinyAspect` Values and Contexts

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.

3.3 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 4.

Expression-level values include the unit value and functions. In `TinyAspect`, advice applies to declarations, *not* to functions. This is crucial for the modular reasoning result described later, as declarations can be hidden behind a module interface but first-class functions cannot. 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 4 also shows the contexts in which reduction may occur. Call-by-value 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.

$$\frac{}{(\eta,\ (\texttt{fn}\ x{:}\tau\ \texttt{=>}\ e)\ v) \mapsto (\eta,\ \{v/x\}e)}\ \textit{r-app} \qquad \frac{\eta[\ell] = v_1}{(\eta,\ \ell\ v_2) \mapsto (\eta,\ v_1\ v_2)}\ \textit{r-lookup}$$

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

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

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

Fig. 5. `TinyAspect` Operational Semantics

Figure 5 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, and there is no rule $\ell \mapsto \eta[\ell]$ 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 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

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

$$\frac{\ell{:}\tau \in \Sigma}{\Gamma;\Sigma \vdash \ell : \tau}\ \textit{t-label} \qquad \frac{}{\Gamma;\Sigma \vdash \texttt{()} : \texttt{unit}}\ \textit{t-unit}$$

$$\frac{\Gamma, x{:}\tau_1;\Sigma \vdash e : \tau_2}{\Gamma;\Sigma \vdash \texttt{fn}\ x{:}\tau_1\ \texttt{=>}\ e : \tau_1 \to \tau_2}\ \textit{t-fn} \qquad \frac{\Gamma;\Sigma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma;\Sigma \vdash e_2 : \tau_2}{\Gamma;\Sigma \vdash e_1\ e_2 : \tau_1}\ \textit{t-app}$$

$$\frac{}{\Gamma;\Sigma \vdash \bullet : \bullet}\ \textit{t-empty} \qquad \frac{\Gamma;\Sigma \vdash v : T \quad \Gamma;\Sigma \vdash d : \beta}{\Gamma;\Sigma \vdash \textit{bind}\ x \equiv v\ \ d : (x{:}T, \beta)}\ \textit{t-vdecl}$$

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

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

$$\frac{\forall \ell \in \textit{domain}(\Sigma)\ .\ \ \bullet;\Sigma \vdash \eta[\ell] : \Sigma[\ell])}{\Sigma \vdash \eta}\ \textit{t-env}$$

Fig. 6. `TinyAspect` Typechecking

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 4.

3.4 Typechecking

Figure 6 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 $\tau_1 \to \tau_2$. When a val or pointcut binding becomes a value, the typing rule is the same except that subsequent declarations cannot see the bound variable (as it has already been substituted in). 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, used in the soundness theorem, is analogous to store typings in languages with references.

3.5 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 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.

Our type soundness theorem is slightly stronger than the previous result of Walker et al., in that we guarantee both type safety and a lack of run time errors. Walker et al. model `around` advice using a lower-level exception construct, and so their soundness theorem includes the possibility that the program will terminate with an uncaught exception [19].

4 Formalizing Modules

We now extend `TinyAspect` with Open Modules, a module system that allows programmers to enforce an abstraction boundary between clients and the implementation of a module. Our module system is modeled closely after that

Names	$n ::= \ldots \mid m.x$
Declarations	$d ::= \ldots \mid \texttt{structure}\ x = m\ \ d$
Modules	$m ::= n \mid \texttt{struct}\ d\ \texttt{end} \mid m\ \texttt{:>}\ \sigma \mid \texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m \mid m_1\ m_2$
Decl. types	$\beta ::= \ldots \mid x{:}\sigma, \beta$
Module types	$\sigma ::= \texttt{sig}\ \beta\ \texttt{end} \mid \sigma_1 \to \sigma_2$
Module values	$m_v ::= \texttt{struct}\ d_v\ \texttt{end} \mid \texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m$
Contexts	$C ::= \ldots \mid \texttt{structure}\ x = \square\ \ d \mid \texttt{struct}\ \square\ \texttt{end}$
	$\mid\ \ \square\ \texttt{:>}\ \sigma \mid \square\ m_2 \mid m_v\ \square$

Fig. 7. Module System Syntax, Values, and Contexts

```
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
```

Fig. 8. Fibonacci with Open Modules

of ML, providing a familiar concrete syntax and benefiting from the design of an already advanced module system. In a distributed development setting, our module system places restrictions on aspects in order to provide the strong reasoning guarantee in Section 5. In a local development setting, however, our "module interfaces" could be computed by tools, and place no true restrictions on developers.

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

```
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
```

Fig. 9. A shape library that exposes a position change pointcut

module m which may depend on x. Functor application is written like function application, using the form $m_1\ m_2$.

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.

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 in a distributed development setting, 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.

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 changed, the `moves` pointcut must also be updated so that client aspects are triggered in the same way.

Explicitly exposing internal events in an interface pointcut means a loss of some obliviousness in the distributed development case, since the author of the module must anticipate that clients might be interested in the event. On the other hand, we are still better off than in a non-AOP language, because the interface pointcut is defined in a way that does not affect the actual implementation of the module, as opposed to an invasive explicit callback, and because external calls to interface functions can still be obliviously advised.

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 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

$$\frac{\mathit{bind}\ x \equiv v \in d_v}{(\eta,\ \texttt{struct}\ d_v\ \texttt{end}.x) \mapsto (\eta,\ v)}\ \textit{r-path} \qquad \frac{}{\begin{array}{c}(\eta,\ \texttt{structure}\ x = m_v\ \ d) \mapsto \\ (\eta,\ \texttt{structure}\ x \equiv m_v\ \ \{m_v/x\}d)\end{array}}\ \textit{r-struct}$$

$$\frac{}{\begin{array}{c}(\eta,\ (\texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m_1)\ m_v) \\ \mapsto (\eta,\ \{m_v/x\}m_1)\end{array}}\ \textit{r-fapp} \qquad \frac{\mathit{seal}(\eta, d_v, \beta) = (\eta', d_{seal})}{\begin{array}{c}(\eta,\ \texttt{struct}\ d_v\ \texttt{end :> sig}\ \beta\ \texttt{end}) \\ \mapsto (\eta',\ \texttt{struct}\ d_{seal}\ \texttt{end})\end{array}}\ \textit{r-seal}$$

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

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

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

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

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

Fig. 10. Module System Operational Semantics

$$\frac{\Gamma; \Sigma \vdash m : \texttt{sig}\ \beta\ \texttt{end} \quad x{:}\tau \in \beta}{\Gamma; \Sigma \vdash m.x : \tau}\ \textit{t-name} \qquad \frac{\Gamma; \Sigma \vdash m : \sigma \quad \Gamma, x{:}\sigma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{structure}\ x = m\ \ d : (x{:}\sigma, \beta)}\ \textit{t-str}$$

$$\frac{\Gamma; \Sigma \vdash d : \beta}{\Gamma; \Sigma \vdash \texttt{struct}\ d\ \texttt{end} : \texttt{sig}\ \beta\ \texttt{end}}\ \textit{t-struct} \qquad \frac{\Gamma; \Sigma \vdash m : \sigma_m \quad \sigma_m\ \texttt{<:}\ \sigma}{\Gamma; \Sigma \vdash m\ \texttt{:>}\ \sigma : \sigma}\ \textit{t-seal}$$

$$\frac{\Gamma, x{:}\sigma_1; \Sigma \vdash m : \sigma_2}{\Gamma; \Sigma \vdash \texttt{functor}(x{:}\sigma_1)\ \texttt{=>}\ m : \sigma_1 \rightarrow \sigma_2}\ \textit{t-ftor} \qquad \frac{\begin{array}{c}\Gamma; \Sigma \vdash m_1 : \sigma_1 \rightarrow \sigma \\ \Gamma; \Sigma \vdash m_2 : \sigma_2 \qquad \sigma_2\ \texttt{<:}\ \sigma_1\end{array}}{\Gamma; \Sigma \vdash m_1\ m_2 : \sigma}\ \textit{t-fapp}$$

Fig. 11. Open Modules Typechecking

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.

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

$$\frac{\beta \texttt{ <: } \beta'}{\texttt{sig } \beta \texttt{ end } \texttt{ <: } \texttt{ sig } \beta' \texttt{ end}}\ \textit{sub-sig} \qquad \frac{\beta \texttt{ <: } \beta'}{x : T, \beta \texttt{ <: } \beta'}\ \textit{sub-omit}$$

$$\frac{\beta \texttt{ <: } \beta' \quad \sigma \texttt{ <: } \sigma'}{x : \sigma, \beta \texttt{ <: } x : \sigma', \beta'}\ \textit{sub-decl} \qquad \frac{\sigma_1' \texttt{ <: } \sigma_1 \quad \sigma_2 \texttt{ <: } \sigma_2'}{\sigma_1 \rightarrow \sigma_2 \texttt{ <: } \sigma_1' \rightarrow \sigma_2'}\ \textit{sub-contra}$$

Fig. 12. Signature Subtyping

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 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 Reasoning about Equivalence

The example programs in Section 4 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.

Asking whether the implementation of a module is correct, or whether changes can be made to the module without affecting clients, is asking about the equivalence between a module implementation and a specification or between two module implementations. For the purposes of this paper, we assume that a specification is given as a reference implementation, reducing both questions to comparing two implementations. This definition is limited, since many specifications are intended to leave some behavior up to the implementor, but we leave a more flexible definition to future work.

A natural definition of equivalence is called *observational equivalence* [17] or *contextual equivalence*, meaning that no client context can distinguish two different implementations of a component. A simple way to define contextual equivalence is to use program termination as the observable variable: two expressions in a program are contextually equivalent if, for all client contexts, the client will either terminate when linked to both implementations of a component, or will run forever when linked to both implementations. We formalize this as follows:

Definition [Contextual Equivalence]: *Two expressions E_1 and E_2 are contextually equivalent, written $E_1 \equiv E_2$, if and only if for all contexts C such that $\bullet; \bullet \vdash C[E_1] : \tau$ and $\bullet; \bullet \vdash C[E_2] : \tau$ we have $(\bullet, E_1) \mapsto^* (\eta_1, V_1) \iff (\bullet, E_2) \mapsto^* (\eta_2, V_2)$.*

By definition, two contextually equivalent modules cannot be distinguished by any client.[1] Thus, contextual equivalence is adequate to answer whether a change to a module might affect clients, or whether an optimized implementation of a module is semantically equivalent to a reference implementation.

$$\frac{(\bullet, E_1)\ \mathit{diverges} \qquad (\bullet, E_1)\ \mathit{diverges}}{E_1 \cong E_2}$$

$$\frac{\begin{array}{c}(\bullet, E_1) \mapsto^* (\eta_1, V_1) \qquad (\bullet, E_2) \mapsto^* (\eta_2, V_2) \qquad \Lambda, \Sigma \vdash (\eta_1, V_1) \simeq (\eta_2, V_2) : T \\ \Lambda = (\mathit{domain}(\eta_1) \cup \mathit{domain}(\eta_2)) - (\mathit{fl}(V_1) \cup \mathit{fl}(V_2)) \qquad \Lambda, \Sigma \vdash \eta_1 \simeq \eta_2\end{array}}{E_1 \cong E_2}$$

Fig. 13. Logical Equivalence for Program Text

5.1 Logical Equivalence

Although contextual equivalence intuitively captures the semantics of equivalence, it is not very useful for actually proving that two modules are equivalent, because it requires quantifying over all possible clients. Instead, we give a more useful set of *logical equivalence* rules that can be used to reason about a module in isolation from possible clients. We then prove that these rules are sound with respect to the more natural, but less useful, contextual equivalence semantics. Finally, we briefly outline how the logical equivalence rules can be used to prove that two different implementations of a module are observationally equivalent.

Figure 13 defines logical equivalence for `TinyAspect` expressions. If two expressions diverge, they are logically equivalent. Otherwise, two expressions are equivalent if, in the empty context, they both reduce to environment-value pairs

[1] Note that since our formal system only models functional behavior, it cannot distinguish implementations with different performance characteristics.

that obey a value equivalence relation, defined below. Note that these equivalence rules apply only to closed expressions; this is not a significant limitation, as expressions with free variables can easily be re-written as functions or functors.

The equivalence relation for values is somewhat more complex, because whether two values are equivalent depends both on the environment bindings for free labels in the value, and on which labels clients can advise. For example, the `Math` module in Figure 8 is equivalent to the same module without caching if clients cannot advise the internal recursive calls to `fib`, but would not be equivalent if clients can advise these calls.

We define an value equivalence judgment of the form $\Lambda, \Sigma \vdash (\eta, V) \simeq (\eta', V') : T$. Here, Λ represents a set of hidden labels that a client cannot advise. Σ is the type of labels that the client can advise (i.e., those not in Λ). The equivalence judgment includes both values and their corresponding environments, because whether two values are equivalent may depend on how they each use their private labels in Λ. A similar judgment, $\Lambda, \Sigma \vdash \eta_1 \simeq \eta_2$, used in the second logical equivalence rule and defined in Figure 14, verifies that two environments map all labels not in Λ to logically equivalent values.

The second rule in Figure 13 sets Λ to be all of the labels in the two environments that are not free in the values being compared (the set of free labels in V is written $fl(V)$). For the `Math` module in Figure 8, only the label generated for the `fib` function as part of the module sealing operation is free in the module value; all other labels, including the one that captures advice on recursive calls to `fib`, are hidden in Λ. This is the technical explanation for why the special semantics of `TinyAspect`'s sealing operation are important; without it, all internal calls to the public functions of a module would be available to advice.

This rule also shows the critical importance of keeping advice second-class in `TinyAspect`. In a system with second-class advice, clients can only advise the free labels in a module value, as shown in the rule. If pointcuts and advice were first-class, a function could compute a pointcut dynamically, return it to clients, which could then advise the pointcut. Keeping track of which functions clients could advise would be extremely difficult in this setting.

The rules for logical equivalence of value/environment pairs are defined in Figure 14. In these rules, we consider machine configurations to be equivalent up to alpha-conversion of label names in the environment.

Our rules are similar to typical logical relations rules, but have one important difference. Because `TinyAspect` supports a limited notion of state through the advice mechanism, logical equivalence is defined as a bisimulation [17]. That is, equivalent functions must not only produce equivalent results given equivalent arguments, they must *also* trigger advice on client-accessible labels in the same sequence and with the same arguments. Another way of saying this is that all possibly-infinite traces of pairs of (client-accessible label, argument value) triggered by logically equivalent functions must be themselves equivalent.

This bisimulation cannot be defined inductively on types as is usual for logical relations, because a function of type $\tau \rightarrow \tau'$ may trigger advice on labels whose types are themselves bigger than τ or τ'. Instead, the rules in Figure 14 should be

$$\frac{}{\Lambda, \Sigma \vdash (\eta_1, \texttt{()}) \simeq (\eta_2, \texttt{()}) : \texttt{unit}} \qquad \frac{\ell \notin \Lambda}{\Lambda, \Sigma \vdash (\eta_1, \ell) \simeq (\eta_2, \ell) : \Sigma[\ell]}$$

$$\frac{\ell \in \Lambda \qquad \Lambda, \Sigma \vdash (\eta_1, \eta_1[\ell]) \simeq (\eta_2, v) : \tau}{\Lambda, \Sigma \vdash (\eta_1, \ell) \simeq (\eta_2, v) : \tau} \qquad \frac{\ell \in \Lambda \qquad \Lambda, \Sigma \vdash (\eta_1, v) \simeq (\eta_2, \eta_2[\ell]) : \tau}{\Lambda, \Sigma \vdash (\eta_1, v) \simeq (\eta_2, \ell) : \tau}$$

$$\frac{\ell \notin \Lambda \qquad \Lambda, (\Sigma, \ell{:}\tau') \vdash (\eta_1, (\texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_1)\ \ell) \cong (\eta_2, (\texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_2)\ \ell) : \tau}{\Lambda, \Sigma \vdash (\eta_1, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_1) \simeq (\eta_2, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_2) : \tau' \to \tau}$$

$$\frac{}{\Lambda,\Sigma \vdash (\eta, \bullet) {\simeq} (\eta', \bullet) {:} (\bullet)} \qquad \frac{\ell \notin \Lambda \qquad \Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{val}\ x \equiv \ell\ \ d_v) \simeq (\eta', \texttt{val}\ x \equiv \ell\ \ d_v') : (x{:}\Sigma[\ell], \beta)}$$

$$\frac{\ell \notin \Lambda \qquad \Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda,\Sigma \vdash (\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v) \simeq (\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v') : (x{:}\texttt{pc}(\Sigma[\ell]), \beta)}$$

$$\frac{\Lambda, \Sigma \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma \qquad \Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{structure}\ x \equiv m_v\ \ d_v) \simeq (\eta', \texttt{structure}\ x \equiv m_v'\ \ d_v') : (x{:}\sigma, \beta)}$$

$$\frac{\Lambda, \Sigma \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta}{\Lambda, \Sigma \vdash (\eta, \texttt{struct}\ d_v\ \texttt{end}) \simeq (\eta', \texttt{struct}\ d_v'\ \texttt{end}) : \texttt{sig}\ \beta\ \texttt{end}}$$

$$\frac{\begin{array}{c}\forall m^3, m^4 \quad \emptyset \vdash (\bullet, m^3) \cong (\bullet, m^4) : \sigma' \implies \\ \Lambda, \Sigma \vdash (\eta_1, m_v^1\ m^3) \cong (\eta_2, m_v^2\ m^4) : \sigma\end{array}}{\Lambda, \Sigma \vdash (\eta_1, m_v^1) \simeq (\eta_2, m_v^2) : \sigma' \to \sigma}$$

$$\frac{\forall \ell\ .\ \ell \in (\mathit{domain}(\eta_1) \cup \mathit{domain}(\eta_2)) \wedge \ell \notin \Lambda \implies \Lambda, \Sigma \vdash (\eta_1, \eta_1[\ell]) \simeq (\eta_2, \eta_2[\ell]) : \Sigma[\ell]}{\Lambda, \Sigma \vdash \eta_1 \simeq \eta_2}$$

Fig. 14. Logical Equivalence for Machine Values

interpreted coinductively for ordinary types–and thus all the rules for ordinary types τ are designed to be monotonic to ensure that the greatest fixpoint of the equivalence relation exists. We can still use an inductive definition of equivalence for module types σ, since module definitions cannot be advised To the best of our knowledge, this coinductive interpretation of logical equivalence rules is novel.

The first rule states that all unit values are equivalent. The second states that we can assume that any non-private label (i.e., one not in Λ) is equivalent to itself. Other labels can be judged equivalent to another value by looking up the label in the environment.

Two functions are equivalent if, when invoked with a fresh label, they execute with that label in a bisimilar way (using the machine expression equivalence judgment from Figure 15). We cannot use the usual logical relations rule for function equivalence, because this rule quantifies over logically equivalent pairs of arguments and is thus non-monotonic and incompatible with our co-inductive definition of equivalence.

$$\frac{(\eta_1, E_1) \mapsto^*_\Lambda (\eta'_1, V_1) \qquad (\eta_2, E_2) \mapsto^*_\Lambda (\eta'_2, V_2) \qquad \Lambda, \Sigma \vdash (\eta'_1, V_1) \simeq (\eta'_2, V_2) : T}{\Lambda, \Sigma \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{(\eta_1, E_1) \mapsto^+_\Lambda (\eta'_1, E'_1) \qquad (\eta_2, E_2) \mapsto^+_\Lambda (\eta'_2, E'_2) \qquad \Lambda, \Sigma \vdash (\eta'_1, E'_1) \cong (\eta'_2, E'_2) : T}{\Lambda, \Sigma \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{\begin{array}{cc} \ell, \ell' \notin \Lambda & \Lambda, \Sigma \vdash (\eta_1, v_1) \simeq (\eta_2, v_2) : \tau \\ \Sigma[\ell] = \tau \to \tau' & \Lambda, (\Sigma, \ell' {:} \tau') \vdash (\eta_1, C_1[\ell']) \cong (\eta_2, C_2[\ell']) : T \end{array}}{\Lambda, \Sigma \vdash (\eta_1, C_1[\ell\ v_1]) \cong (\eta_2, C_2[\ell\ v_2]) : T}$$

Fig. 15. Logical Equivalence for Machine Expressions

Two empty declarations are equivalent to each other. 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). Since the label exposed by the `val` declaration is visible, it must not be in the private set of labels Λ. Pointcut and structure declarations just check the equality of their components. All three declaration forms ensure that subsequent declarations are also equivalent. Two first-order modules are equivalent if the declarations inside them are also equivalent.

For functors, we use the usual logical relations rule: two functors are equivalent if they produce logically-equivalent module results for any logically-equivalent, closed module arguments. This definition is well-formed because equivalence for module types σ is defined inductively rather than coinductively, using the coinduction only for the base case of functions within modules.

Figure 15 shows the rules for logical equivalence of expression/environment pairs. These rules enforce bisimilarity with respect to values returned by a function or functor, or values passed to a non-private label. Our definitions are similar to weak bisimilarity in the π-calculus, with ordinary reductions or lookups of private labels corresponding to τ-transitions in the standard notion of weak bisimilarity.

The first rule states that two expressions are equivalent if they take any number of non-observable steps (written $\mapsto^*_\Lambda$) to reduce to values that are also equivalent. The non-observable step relation $\to_\Lambda$ is equivalent to ordinary reduction $\to$, except that the *r-lookup* rule may only be applied to labels in Λ (i.e., those that cannot be advised by clients).

The second rule allows two expressions to each take one or more non-observable steps (indicated by the $+$ superscript instead of the $*$ superscript for zero or more steps), resulting in observationally-equivalent expressions. Finally, the last rule states that if two equivalent expressions are both at the point where they need to look up a label that is *not* in Λ in order to continue, we must verify that the values to which the labels are applied are also equivalent, and that the contexts are equivalent once a fresh label (representing the result of the application, which is unknowable due to client advice) is substituted into the contexts. Since our definition of equivalence is coinductive, we can use an infinite sequence of the second and third rules to conclude that two non-terminating expressions are logically equivalent.

Now that we have defined logical equivalence, we can state a soundness theorem relating logical and contextual equivalence:

Theorem 3 (Soundness of Logical Equivalence). *If* $E_1 \cong E_2$ *then* $E_1 \equiv E_2$*.*

For space reasons, we give only a brief sketch of the proof of soundness. More details are available in a companion technical report [1]. The proof proceeds by establishing a bisimulation between two programs that consist of the same context with logically equivalent embedded values. The bisimulation invariant states that the two programs are structurally equivalent except for embedded closed values, which are themselves logically equivalent. The key lemma in the theorem states that the bisimulation is sound; that is, the bisimulation invariant is preserved by reduction.

We then observe that the logically equivalent expressions E_1 and E_2 either both diverge, or both reduce to logically equivalent values. In the former case, any context surrounding the expressions will also diverge, so the expressions are contextually equivalent in this case. In the latter case, the expressions will both reduce to values in the same context, which will then obey the bisimulation invariant described above. The soundness of bisimulation implies that these expressions will either both diverge or will reduce to logically equivalent values. Thus, the expressions are contextually equivalent in this case as well.

5.2 Applying Logical Equivalence

The definition of logical equivalence can be used to ensure that changes to the implementation of one module within an application preserve the application's semantics. For example, consider replacing the recursive implementation of the Fibonacci function in Figure 8 with an implementation based on a loop. In AspectJ, or any module system that does not include the dynamic semantics of our sealing operation, this seemingly innocuous change does not preserve the semantics of the application, because some aspect could be broken by the fact that `fib` no longer calls itself recursively.

Open Modules ensure that this change does not affect the enclosing application, and the logical equivalence rules can be used to prove this. When the module in Figure 8 is sealed, `fib` is bound to a fresh label that forwards external calls to the internal implementation of `fib`. We can show that the two implementations of the module are logically equivalent by showing that no matter what argument value the `fib` function is called with, the function returns the same results and invokes the *external* label in the same way. But the external label is fresh and is unused by either `fib` function, so this reduces to proving ordinary function equivalence, which can easily be done by induction on the argument value. We can then apply the abstraction theorem to show that clients are unaffected by the change.

6 Discussion

In this section, we reconsider the research questions from the introduction in light of the formal model of Open Modules and the logical equivalence definition. Since the formal model can be used to represent the tool support provided by IDEs like the AJDT, we consider how these tools might be enhanced in light of the model.

1. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still ensuring critical properties of the component implementation?*

They can do so by declaring explicit pointcuts in the interface of the component that describe "supported" internal events. These pointcuts form a contract between a component provider and a client: The provider promises that if the client obeys the rules of the Open Module system, then the system will have the desired properties, and upgrades to the component will preserve client functionality. The client's side of the contract can be enforced by a compiler, while the component provider's side can be verified using the logical equivalence rules (and their principled extension to a full AOP language)

Once the proper interface has been declared, our logical equivalence rules show how to prove the full functional correctness of a module in a completely modular way, given that an aspect-aware interface has been computed and that an appropriate specification (e.g., in the form of a reference implementation) is available.

Of course, this answer is also limited: we give formal rules for a core language, and full languages are far more complex. However, our system could be used to prove the correctness of compiler optimizations for the constructs that are expressed in the core language. Our system may also be the foundation for a richer set of equivalence rules that can be applied to a full system. Finally, the formal rules in our system may be most helpful by showing engineers how to reason *informally* about the correctness of changes to an aspect-oriented program. Our system yields strong theoretical support to the intuitive notion that changes to a module will not affect clients as long as functions compute the same result and trigger pointcuts in the same way as the original module does.

2. *How can developers specify an interface for a library or component that permits as many uses of advice as possible, while still allowing the component to be changed in meaningful ways without affecting clients?*

The same kind of interface can be defined as in the question above. In the context of software evolution, however, the logical equivalence rules can be used to ensure that a proposed change to a module cannot affect that module's clients, even if the clients themselves are unknown. As long as the client code obeys the Open Module interface, it cannot be broken by an upgraded version of a third-party component.

Note that the client is free to choose to bypass the Open Module rules; a future compiler for Open Modules might issue a warning but compile the code anyway. In this case the client would lose the guarantee that upgrades would preserve the correctness of client code, but gain the ability to reuse or adapt the component in more flexible ways than are permitted by the Open Module system. Depending on the precise circumstances, the reuse benefits might outweigh the potential costs of fixing code that breaks after an upgrade.

Tool Support. As Parnas argued, the primary goal of modularity is to ease software evolution. The model we have developed shows that evolving a module's implementation might also involve changing the pointcuts that act on the module, so that they capture events in the new module's implementation that correspond to the events captured by the original pointcut in the original module specification. Currently the AJDT IDE aids this process by identifying what these pointcuts are, but the pointcuts must still be changed at their definition points, making the implementation task non-local. Our model suggests an improvement to IDEs: providing an editable view of the portion of each pointcut that intersects with a module's source code would allow many changes to be made in a more local way.

Language Design. Our modularity result suggests a number of guidelines for AOP language designers who want to preserve modular reasoning. First, declarative, second-class advice (as in `TinyAspect`) is easier to reason about than first-class advice. Second, in a language with first-class functions, advice should affect function declarations, not the functions themselves, again because this allows a module system to scope the effect of advice. Finally, languages should distinguish advice that affects only the interface of a module from more invasive forms of advice that affect a module's implementation.

7 Related Work

Formal Models. The most closely related formal models are the foundational calculus of Walker et al. [19], and the model of AspectJ by Jagadeesan et al. [10], both of which were discussed in the beginning of Section 3. In other work on formal models of AOP, Lämmel provides a big-step semantics for method-call interception in object-oriented languages [15]. Wand et al. give an untyped, denotational semantics for advice and dynamic join points [20].

Aspects and Modules. Dantas and Walker have extended the calculus of Walker et al. to support a module system [7]. 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 and advice applies to functions, providing more flexibility compared to `TinyAspect`, where pointcuts are second-class and advice applies to declarations. This design choice makes it much more difficult to prove logical equivalence properties, however, because either making pointcuts first-class or tying advice to functions allows join points to escape from a module even if they are not explicitly exported in the module's interface. In their system, functions can only be advised if the function declaration explicitly permits this, and so their system is not oblivious in this respect [8]. In contrast, `TinyAspect` allows advice on all function declarations, and on all functions exported by a module, providing significant “oblivious” extensibility without compromising modular reasoning.

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 [16]. 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 provide as strong an abstraction boundary as Open Modules does.

Other researchers have studied modular reasoning without the use of explicit module systems. For example, Clifton and Leavens propose engineering techniques that reduce dependencies between concerns in aspect-oriented code [5]. Other work has studied analyzing base code and advice separately using interfaces similar to those of Open Modules [14], and analyzing what advice might be affected by a change to code [13].

Our module system is based on that of standard ML [18]. `TinyAspect`’s sealing construct is similar to the freeze operator that is used to close a module to future extensions in module calculi such as Jigsaw [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 [6].

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 Gregor Kiczales, Mira Mezini, Shriram Krishnamurthi, Ralf Lämmel, Mitch Wand, Karl Lieberherr, David Walker, Tim Halloran, Curtis Clifton, Derek Dreyer, Todd Millstein, Robert Harper, and anonymous reviewers for comments on this material. This work was supported by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled ”Perpetually Available and Secure Information Systems.”

References

1. J. Aldrich. Open Modules: Modular Reasoning about Advice. Carnegie Mellon Technical Report CMU-ISRI-04-141, available at http://www.cs.cmu.edu/~aldrich/aosd/, Dec. 2004.
2. D. G. Bobrow, L. G. DiMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. In *SIGPLAN Notices 23*, September 1988.
3. G. Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. Ph.D. Thesis, Dept. of Computer Science, University of Utah, 1992.
4. A. Clement, A. Colyer, and M. Kersten. Aspect-Oriented Programming with AJDT. In *ECOOP Workshop on Analysis of Aspect-Oriented Software*, July 2003.
5. C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages*, April 2002.
6. 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.
7. D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Princeton University Technical Report TR-696-04, 2004.
8. R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Advanced Separation of Concerns*, October 2000.
9. S. Gudmundson and G. Kiczales. Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface. In *Advanced Separation of Concerns*, July 2001.
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. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, June 1997.
13. C. Koppen and M. Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *European Interactive Workshop on Aspects in Software*, September 2004.
14. S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying Aspect Advice Modularly. In *Foundations of Software Engineering*, Nov. 2004.
15. R. Lämmel. A Semantical Approach to Method-Call Interception. In *Aspect-Oriented Software Development*, Apr. 2002.
16. K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, September 2003.
17. R. Milner. *Communicating and Mobile Systems: The π-Calculus.* Cambridge University Press, 1999.
18. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.
19. D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. In *International Conference on Functional Programming*, 2003.
20. 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*, 26(5):890–910, September 2004.