A Formal Model of Modularity 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

Aspect-Oriented Programming promises to make programs more modular by separating crosscutting concerns more effectively, but also challenges existing ideas about modularity and separate development. Although models of modularity have been proposed for aspects, these models lack a precise, formal definition and a guarantee of properties such as separate reasoning or ease of change.

This paper develops a formal theory of modularity for aspect-oriented programming, making earlier informal theories more precise and describing specific conditions under which separate reasoning is possible. We begin with a new, simple formal model for aspect-oriented programming languages. We then add a module system that 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. Using this module system, we give the first proof of a representation independence property for any aspect-oriented programming language, showing that clients are unaffected by semantics-preserving changes to a module's implementation. Our model yields insights into the nature of AOP modularity, provides a mechanism for enforceable contracts between an AOP component provider and clients, and suggests improvements to current AOP IDEs.

1. Modularity and Aspects

Although AOP provides 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 advised by aspects. Oblivious extension mechanisms such as advice in AspectJ appear to make reasoning about the effect of calling a method more challenging, for example, because they 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.

In practice, tool support such as the AspectJ plugin for Eclipse (AJDT) can help to address this problem [4]. The AJDT shows a programmer where aspects apply to her code, so that she can examine the relevant aspects to get a full picture of what happens when a method call is made. This information is essential for correctly evolving code as well, because developers can easily see which aspects depend on a segment of base code and ensure that their changes preserve the aspects' functionality.

Recently, Kiczales and Mezini proposed a model of modularity in aspect-oriented programming that explains why tool support is helpful for modular reasoning [11]. Their model introduces the concept of *aspect-aware interfaces*, which are conventional interfaces augmented with information about the pointcuts and advice that apply to a module. For example, if a `DisplayUpdate` aspect places `after` advice on the `move` method of the `Shape` class, then the interface of `Shape` would include this information.

In the Aspect-Aware Interface model, the interface of a module depends on the system in which it is deployed. Different systems may include different aspects that advise the same module in different ways, and so the same module's interface will be different in each system. Thus, the interface cannot be specified with the module itself—in fact, if it were specified with the module, it would compromise the obliviousness criterion of AOP. Instead, the aspect-aware interface must be computed by tools with knowledge of the deployment configuration. The AJDT environment mentioned above is an example of such a tool.

In this paper, we develop a formal model that describes aspect-aware interfaces *after* they are computed by tools. The goal of our model is to more precisely answer the following research questions:

1. *What information must be included in an aspect-aware interface?*
2. *What kind of modular reasoning is possible once an aspect-aware interface has been computed?*
3. *Given an aspect-aware interface, what kinds of changes can be made to a module without affecting clients?*

1.1 AOP and Separate Development

While the aspect-aware interface model is a valuable aid to separate reasoning when the configuration of a system is known, it does not provide guidance for reasoning in the case where the system configuration is unknowable. For example, when a third-pary component or library is deployed by a client, the original developers of the component typically have no knowledge of the deployment information, and thus cannot compute an aspect-aware interface for the module.

Consider the Java standard library, which is carefully designed to provide a "sandbox" security model for running untrusted code. If the developers of the Java standard library knew the aspect-aware interface of the library for a particular

configuration, they could carefully check the interface to ensure that aspects do not violate the security constraints of the library. In practice, however, this is impossible because the library designers cannot foresee all possible deployments of the library. Thus, it is unsafe in general for a user to load any aspect that advises the standard library, because the aspect 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 aspects. 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 the following additional research question:

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

Another 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 might make could potentially break some client's aspects. Although we could solve the problem by prohibiting all advice to third-party components, we would prefer a compromise that answers the research question:

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

The research questions above imply a solution that prohibits certain uses of aspects 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 he is writing an aspect that may break when a new version of a component is released, and when he is writing an aspect that is resilient to new releases. Similarly, a user uploading code should know whether that code only includes aspects that are guaranteed not to violate the Java sandbox, or whether the code contains aspects 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.

1.2 Outline and Contributions

The next section of the paper describes *Open Modules*, a novel module system for aspects that provides informal answers to research questions 4 and 5 above. Open Modules is the first module system that supports many beneficial uses of aspects, while still ensuring that security and other properties of a component implementation are maintained, and verifying that the aspect-oriented extensions to a component will not be affected by behavior-preserving changes to the component's implementation. We can consider AJDT's IDE support for aspects as a tool that computes an Aspect-Aware/Open Module Interface from a given deployment configuration, and thereby obtain informal answers to research questions 1-3 as well.

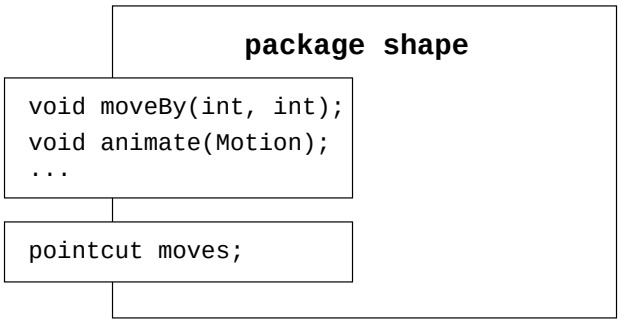

Figure 1: A conceptual view of Open Modules. The `shape` module exports two functions and a pointcut. Clients can place advice on external calls to the exported functions, or on the exported pointcut, but not on calls that are internal to the module.

We would like to make these answers precise, and to do so, Section 3 proposes `TinyAspect`, a novel formal model of aspect-oriented programming languages that captures a few core constructs of AOP while omitting complicating details.

Section 4 extends `TinyAspect`with Open Modules, precisely defining the semantics and typing rules of the system. By describing exactly what must be stated in an aspect-aware/open module interface, our system gives precise answers to research question 1 above.

Section 5 proves a representation independence or abstraction theorem for the module system. This is the first result to show that certain kinds of reasoning can be done on a module-by-module basis in an AOP system, providing a precise answer to research questions 2 and 4. It also precisely defines what changes can be made to a component without affecting clients, answering research questions 3 and 5.

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 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 (at least in the case of separate development), and so we try to achieve a compromise between them.

Previous systems support AOP using constructs like advice that can reach across module boundaries to capture crosscutting concerns. We propose to adopt these same constructs, but limit them so that they respect module boundaries. In order to capture concerns that crosscut the boundary of a module, the module can export pointcuts that represent *abstractions* of events that might be relevant to external aspects. 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 de-

noting internal semantic events. For example, the `moves` pointcut in Figure 1 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 the case of local development and tool support, there is ***no*** restriction on the use of aspects—when 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 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 (including pointcuts automatically added to the interface by a tool)*
- *does not allow external aspects to directly advise calls from within a module to other functions within the module (including calls to exported functions).*

For internal calls within a module, the open module system, originally presented at the FOAL 2004 workshop, corresponds closely to the computation of aspect-aware interfaces that has since been proposed. Our proposal is more flexible, however, in the case of advice on external calls to a module, which can be implemented without adding an explicit pointcut to the module's interface. This benefit is important when programmers write these interfaces (as in the case of separate development but not in the case of local development with the AJDT), because many aspects rely only on interface calls, and writing explicit pointcuts for all of these would be cumbersome.

3. Formally Modeling Aspects

In order to reason formally and precisely about modularity, we need a formal model of aspect-oriented programming. The most attractive models for this purpose are based on small-step operation semantics, which provide a very simple and direct formalization and are amenable to standard syntactic proof techniques.

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

Figure 2: `TinyAspect` Source Syntax

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. However, we would like to prove representation independence, a strong modularity property, 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 AOP, including the obliviousness property [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 abstraction theorem in Section 5. Nevertheless, it includes the typed lambda calculus with recursion and is thus Turing-complete. Although `TinyAspect` leaves out many features of full languages, we directly model pointcut and 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.

Figure 2 shows the syntax of `TinyAspect`. Our syntax is modeled after ML [15]. 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-

```
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 end
```

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

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. 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 $\tau_1 \rightarrow \tau_2$.

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. We assume integers, booleans, and if statements 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 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.

Expression values	v	::=	`()` | **`fn`** x:τ `=>` 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 4: `TinyAspect` Values and Contexts

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.

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. 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

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

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

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

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

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

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

Figure 5: `TinyAspect` Operational Semantics

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. 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 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, 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 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: `TinyAspect` Typechecking

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 $\tau_1 \rightarrow \tau_2$. 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 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].

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

Figure 7: Module System Syntax, Values, and Contexts

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 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" are really 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 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

```
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

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.

```
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

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

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

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

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

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

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

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

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

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

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

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

Figure 10: Module System Operational Semantics

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

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

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

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

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

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

Figure 11: Open Modules Typechecking

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

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

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

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

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

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

Figure 12: Signature Subtyping

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.

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. Abstraction

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.

Reynolds' *abstraction* or *representation independence* property [17] 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 [16].

In `TinyAspect`, we can state the abstraction property as follows. If two modules m and m' are logically 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 logically equivalent if all of the bound functions 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 and pointcuts 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.

5.1 Formalizing Abstraction

We can define abstraction formally using judgments for logical equivalence of values, written
$\Lambda \vdash (\eta, V) \simeq (\eta', V') : T$ and read, "In the context of a set of private labels Λ, value V in environment η is logically 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 logically equivalent expressions. The judgments depend on the set of labels Λ that are private to the two abstractions and are thus protected from advice; since all other labels may be advised by a client, in order for two expressions to be logically equivalent, they must use these labels in the same way.

The rules for logical 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. Logical equivalence is defined coinductively as the greatest fixed point of the value rules in Figure 13 and the expression rules in Figure 14.

The most interesting rule is the one for function values. Two function values are equivalent if for any logically equivalent argument values v_1 and v_2 that do not refer to any private labels in Λ, they produce equivalent results. A similar rule is used for logical equivalence of functors.

Two empty declarations are equivalent to each other, and a label is equivalent to itself as long as it's not in the set of private labels Λ. 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.

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

$$\frac{(\eta_1, E_1) \overset{\Lambda}{\mapsto}{}^* (\eta_1', E_1') \qquad (\eta_2, E_2) \overset{\Lambda}{\mapsto}{}^* (\eta_2', E_2') \qquad \Lambda \vdash (\eta_1', E_1') \cong (\eta_2', E_2') : T}{\Lambda \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T}$$

$$\frac{\ell \notin \Lambda \qquad \Lambda \vdash (\eta_1, v_1) \simeq (\eta_2, v_2) : \tau \qquad \Lambda \vdash (\eta_1, C_1) \cong (\eta_2, C_2) : \tau' \to T}{\Lambda \vdash (\eta_1, C_1[\ell\ v_1]) \cong (\eta_2, C_2[\ell\ v_2]) : T}$$

$$\frac{\forall v_1, v_2 \text{ such that } \Lambda \vdash (\eta_1, v_1) \simeq (\eta_2, v_2) : \tau \text{ and } fl(v_1) \cup fl(v_2) \cap \Lambda = \emptyset \qquad \Lambda \vdash (\eta_1, C_1[v_1]) \cong (\eta_2, C_2[v_2]) : T}{\Lambda \vdash (\eta_1, C_1) \cong (\eta_2, C_2) : \tau \to T}$$

Figure 14: Coinductive Definition of Logical Equivalence for Expressions

We also define equivalence for two environments; the rule is that they must be equivalent at all labels not in Λ.

Figure 14 shows the rules for logical equivalence of expressions. Two expressions are equivalent if they are 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 (ignoring the hidden set of labels Λ) while either diverging or reducing to logically equivalent values (since clients can use advice to observe lookups to any label not in Λ).

We formalize this with two rules. The first allows two expressions to take any number of steps that include lookup of labels in Λ, but not other labels. We represent this with the evaluation relation $\overset{\Lambda}{\mapsto}{}^*$, which is identical to $\mapsto^*$ except that the rule *r-lookup* may only be applied to labels in Λ. The resulting machine configurations must be logically equivalent with respect to Λ. The second rule states that two expressions can look up the same label ℓ not in Λ, as long as the argument values (v_1, v_2) are equivalent, and as long as the surrounding contexts C_1 and C_2 treat the returned values in equivalent ways. This last property is defined in the final rule, stating that two contexts are equivalent if whenever they are given equivalent argument values, they execute in a logically equivalent way.

Note that since logical equivalence is coinductively defined, expressions that diverge according to the logical equivalence rules are logically equivalent. This would *not* be true in an inductive definition, because being in the least fixed point of the rules requires the value base case, but the greatest fixed point includes infinite sequences of logically equivalent expressions. Using coinduction is also essential to making the definition meaningful, since the definition of logically equivalent values and expressions are mutually dependent.

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

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

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

$$\Lambda \vdash (\eta_1, \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e_1) \simeq (\eta_2, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_2) : \tau' \to \tau \quad \text{iff} \quad \begin{array}{l}\text{for all } v_1', v_2' \text{ such that } (fl(v_1) \cup fl(v_2)) \cap \Lambda = \emptyset \\ \text{and } \Lambda \vdash (\eta_1, v_1') \simeq (\eta_2, v_2') : \tau' \text{ we have} \\ \Lambda \vdash (\eta_1, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_1\ v_1') \cong (\eta_2, \texttt{fn}\ x{:}\tau'\ \texttt{=>}\ e_2\ v_2') : \tau\end{array}$$

$$\Lambda \vdash (\eta_1, \ell) \simeq (\eta_2, \ell) : \tau \quad \text{iff} \quad \ell \notin \Lambda$$

$$\Lambda \vdash (\eta, \bullet) \simeq (\eta', \bullet) : (\bullet)$$

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

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

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

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

$$\Lambda \vdash (\eta_1, m_v^1) \simeq (\eta_2, m_v^2) : \sigma' \to \sigma \quad \text{iff} \quad \begin{array}{l}\text{for all } m_v^3, m_v^4 \text{ such that } \Lambda \vdash (\eta_1, m_v^3) \simeq (\eta_2, m_v^4) : \sigma' \\ \text{and } (fl(m_v^3) \cup fl(m_v^4)) \cap \Lambda = \emptyset \\ \text{we have } \Lambda \vdash (\eta_1, m_v^1\ m_v^3) \cong (\eta_2, m_v^2\ m_v^4) : \sigma\end{array}$$

$$\Lambda \vdash \eta_1 \simeq \eta_2 : \Sigma \quad \text{iff} \quad \begin{array}{l}\Lambda \subseteq (domain(\eta_1) \cup domain(\eta_1)), \\ domain(\Sigma) = (domain(\eta_1) \cup domain(\eta_2)) - \Lambda \\ \forall (\ell{:}T) \in \Sigma\ .\ \Lambda \vdash (\eta_1, \eta_1[\ell]) \simeq (\eta_2, \eta_2[\ell]) : T\end{array}$$

Figure 13: Coinductive Definition of Logical Equivalence for Values

$(\eta', \textbf{\textit{structure}}\ x = m_v'\ \ d) : (x{:}\sigma, \beta)$

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 proceeds by establishing a bisimulation between two programs that consist of the same client running against logically equivalent modules. 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 two programs being compared are initially bisimilar. By the soundness lemma, they either remain bisimilar indefinitely, corresponding to the divergence case of logical equivalence, or else they eventually reduce to values which are logically equivalent.

5.2 Applying Abstraction

The abstraction theorem can be used together with the definition of logical equivalence 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 abstraction theorem 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 abstraction property. 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. *What information must be included in an aspect-aware interface?*

Our abstraction result shows that strong modular reasoning is possible if an aspect-aware interface contains *only* a conventional interface plus pointcuts describing internal calls that are advised by external aspects.

The aspect-aware interfaces proposal [11] suggests that an aspect-aware interface should contain not only pointcuts describing internal events that are advised, but also information about the specific advice that is applied to these pointcuts. Our model suggests that this additional information is not necessary for modular reasoning—which is fortunate in the separate development case, because it would be impossible for a component provider to anticipate the exact aspects that clients might apply to a module. In the case of local de-

velopment, where tools compute the aspect-aware interface, this extra information might be useful to programmers even if it is not strictly necessary according to our theory.

2. ***What kind of modular analysis is possible once an aspect-aware interface has been computed?***

Our abstraction theorem implies that it is possible 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.

While our theorem does not explicitly cover non-functional characteristics like performance or reliability, it suggests ways of reasoning about these characteristics. For example, to reason about meeting a real-time performance, one could use assume-guarantee reasoning of the form "assuming all advice to pointcuts in an interface runs in some time bound, we can guarantee that the various functions in the interface will run in some other time bound." Future work includes developing modular aspect-oriented analyses for non-functional and partial correctness properties.

3. ***Given an aspect-aware interface, what kinds of changes can be made to a module without affecting clients?***

Our abstraction theorem provides a precise answer to that question: any changes that result in a new module that is logically equivalent to the old one according to the rules in Figure 13, will not affect clients.

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.

The most important contribution of the equivalence rules is conceptual, however: they show how engineers can 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.

4-5. ***How can developers specify an interface for a library or component that permits as many uses of aspects as possible, while still ensuring critical properties of the component implementation, and while still allowing the component to be changed in meaningful ways without affecting clients?***

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)

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.

With respect to separate development, our model suggests that IDEs could be built to warn when client code declares a pointcut that depends on implementation characteristics of a third-party component, helping programmers to write code that is more robust to change. Sometimes such implementation-dependent pointcuts are necessary, however, and in these cases an IDE could remind the programmer to recheck the semantics of these pointcuts whenever a new version of the component is delivered.

Language Design. Although our formal model is limited to a few core AOP constructs, it offers insights into the modularity of more complicated constructs. For example, the requirement that modules behave in a bisimilar way with respect to pointcuts implies that when a language contains `cflow` pointcuts, the modularity rules are exactly the same ***except*** that semantics-preserving changes to a module must preserve control-flow relationships between calls to and from a module. Handing off control to another thread to make a call, for example, is an example of an implementation change that could break `cflow` client aspects.

Our module system's distinction between internal and external calls to the interface to a module implies that some pointcuts—namely, pointcuts advising internal calls or executions within a module—are more "invasive" than others in terms of depending on implementation rather than interfaces. Designers of aspect-oriented programming languages could distinguish these pointcuts syntactically, making it easy to program using only non-invasive pointcuts where this is possible, and making it clear when invasive pointcuts are used so that programmers can make an informed decision about whether this is the right choice or not.

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 systems for aspect-oriented programming, Lämmel provides a big-step semantics for method-call interception in object-oriented languages [12]. Wand et al. give an untyped, denotational semantics for advice advice and dynamic join points [20]. 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 functional languages to provide static and dynamic aspects [18].

Aspects and Modules. Dantas and Walker are currently extending 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, providing more flexibility compared to the second-class pointcuts in `TinyAspect`. This design choice breaks our abstraction theorem, 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 the function declaration explicitly permits this, and so their system is not oblivious is 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 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 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].

Our module system is based on that of standard ML [15]. `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] and related systems [2].

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

[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 and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. Submitted for publication, 2004.

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

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

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

[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*, To Appear 2003.