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

Jonathan Aldrich
Institute for Software Research, International
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA
jonathan.aldrich@cs.cmu.edu

First version March 2004
Revised July 2004

CMU-ISRI-04-108

Abstract

Aspect-oriented programming (AOP) is a new programming paradigm whose goal is to more cleanly modularize crosscutting concerns such as logging, synchronization, and event notification which would otherwise be scattered throughout the system and tangled with functional code. However, while AOP languages provide promising ways to separate crosscutting concerns, they can also break conventional encapsulation mechanisms, making it difficult to reason about code without the aid of external tools.

We investigate modular reasoning in the presence of aspects through `TinyAspect`, a small functional language that directly models aspect-oriented programming constructs. We define Open Modules, a module system for `TinyAspect` that enforces Reynolds' abstraction theorem, a strong encapsulation property. Open Modules are "open" in that external aspects can advise functions and pointcuts in their interface, providing significant aspect-oriented expressiveness that is missing in non-AOP systems. In order to guarantee modular reasoning, however, our system places limits on advice: external aspects may not advise function calls internal to a module, except for calls explicitly exposed through pointcuts in the module's interface. The abstraction property of our system ensures that a module's implementation can be changed without affecting clients, and provides insight into formal reasoning, modular analysis, and tool support for aspect-oriented programming.

Keywords: Open Modules, modular reasoning, abstraction, aspect-oriented programming, TinyAspect, types, modules, logical relations, equivalence, bisimulation

1. Introduction

In his seminal paper, Parnas laid out the classic theory of information hiding: developers should break a system into modules in order to hide information that is likely to change [19]. Thus if change is anticipated with reasonable accuracy, the system can be evolved with local rather than global system modifications, easing many software maintenance tasks. Furthermore, the correctness of each module can be verified in isolation from other modules, allowing developers to work independently on different sub-problems.

Aspect-Oriented Programming (AOP) developed from the insight that conventional modularity and encapsulation mechanisms are not flexible enough to capture many concerns that are likely to change [13]. These concerns cannot be effectively hidden behind information-hiding boundaries, because they *crosscut* the underlying functional or datatype-based decomposition of the program. As a result, code implementing these concerns tends to be *scattered* in many placed throughout the system and *tangled* together with unrelated code.

For example, consider the problem of enforcing a global constraint in a simulation framework, ensuring that objects in the simulation do not overlap. One way of enforcing this constraint is to check it every time some object in the simulation moves. In an conventional system, the code for moving is likely to be spread out among many different functions in the system. Each function involved in moving objects must call constraint-checking code, meaning that these calls will be scattered across these methods, and tangled together with the code for moving objects. This scattering and tangling of the constraint-checking concern makes evolving it more difficult and error-prone, because an engineer must find all of the calls to the constraint checker and change them in a consistent way. If any constraint-checking calls are missed, a defect is likely to result.

Aspect-oriented programming systems provide mechanisms for modularizing crosscutting concerns like constraint checking. For example, Figure 1 shows how simulation invariants could be checked in a more modular way using the constructs in AspectJ, the most widely-used aspect-oriented programming language [12][2]. This example assumes some kind of graphical simulation, and shows two classes, one representing points and another representing rectangles. Both classes have a method `moveBy`, which moves the shapes on the screen.

In a separate package, a constraint-checking aspect is defined to check certain invariants of the simulation every time a shape moves. A *pointcut* is defined to show the set of places in the base simulation code where the constraint-checking should be applied. In this case, the `moves` pointcut refers to all calls to the `moveBy` method in all `shape` classes. Whenever this pointcut is triggered, the *advice* at the bottom of the aspect is invoked. This advice is `after` advice, meaning that it is run after the call to `moveBy` completes. The body of the advice simply invokes the `checkInvariants` method (not shown) which checks that the invariants of the simulation still hold.

The use of aspect-oriented technology to capture the constraint checking concern makes the code for the concern easier to understand and evolve. All the code for the concern is either in the constraint-checking aspect, or in the `checkInvariants` function, so the programmer can easily understand the code just by looking at these two places. In contrast, if this concern had been implemented in a conventional language, calls to `checkInvariants` would be scattered among all the `moveBy` methods of the shape classes. In the aspect-oriented implementation, the developer can change the implementation of the advice or the pointcut showing where it applies in one place, whereas without aspect-oriented technology this change would be spread out, making it more likely that the developer might miss some relevant places in the code.

```
package shape;

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

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


package constraints;

aspect CheckSimulationConstraints {
  pointcut moves():
      call(void shape.*.moveBy(..));

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

Figure 1: This code shows the definition of shapes in a graphical simulation framework, together with an aspect that checks global simulation invariants when the scene changes. The aspect defines a pointcut representing all calls to the `moveBy` methods of shapes, and advice that checks the simulation invariants after each shape moves.

1.1 AOP Definition

Filman and Friedman defined aspect-oriented programming as quantification and obliviousness [8]. *Quantification* is "the idea that one can write...statements that have effect in many, non-local places in a programming system [7]." For example, the pointcut in Figure 1 quantifies over all implementations of the `moveBy` method in the system. *Obliviousness* is the idea that "the places these quantifications applied did not have to be specifically prepared to receive these enhancements [7]." For example, the `moveBy` methods did not have to be specifically prepared (e.g., by calling the advice directly or through a callback) in order to be affected by the advice in Figure 1.

1.2 Aspects and Information Hiding

The example above illustrates how aspect-oriented tech-

[2]The example in Figure 1 illustrates AspectJ syntax, which is object-oriented. The remainder of this paper will focus on aspects in the setting of functional languages.

nology can provide better information hiding for crosscutting concerns, making it easier to understand and evolve the code dealing with these concerns. Unfortunately, aspect technology is a two-edged sword: it can also make software evolution more difficult by coupling aspects tightly to the code that they advise. AspectJ and many other systems allow aspects to reach across encapsulation boundaries, breaking information hiding principles.

For example, the constraint checking aspect in Figure 1 is is tightly coupled to the implementation details of the `shape` package, and will break if these implementation details are changed. Consider what happens if the rectangle is modified to store its coordinates as a pair of points, rather than two pairs of integer values. The body of `Rectangle.moveBy` would be changed to read:

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

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

The aspect in Figure 1 violates the information hiding boundary of the `shape` package by placing advice on method calls within the package. This means that the implementor of `shape` cannot freely switch between semantically equivalent implementations of `Rectangle`, because the external aspect may break if the implementation is changed. Because the aspect violates information hiding, evolving the `shape` package becomes more difficult and error prone.

Although the AspectJ language, considered in isolation, violates information hiding principles, tool support such as the AspectJ plugin for Eclipse (AJDT) can help to address the problem described above[1]. In this paper, we investigate a purely language-based solution to the information hiding problem. Although our solution gives up some of the expressiveness of AOP, it provides insight into why tool support is essential in pure AOP, how it can be improved, and what its limitations are.

1.3 Contributions

This paper makes two contributions to reasoning about aspect-oriented programming systems. First, we define `TinyAspect`, a functional core language for aspect-oriented programming. `TinyAspect` is the first formal model of aspects that is extremely small, models aspect-oriented constructs directly, and is defined using a standard small-step operational semantics. In combination, these properties make it easy to investigate aspect-oriented language extensions and prove theorems about them.

The second contribution of this paper is an extension of `TinyAspect` with Open Modules, a module system that is open to some forms of aspect-oriented extension yet guarantees Reynold's abstraction theorem, a strong encapsulation property [20]. Open Modules's design is based on the principle that a module may choose to expose internal semantic events as pointcuts to clients, but clients may not depend on implementation details that are not part of the semantics of the module interface.

In our design, the interface of an open module exposes a set of values, functions and pointcuts. The pointcuts represent internal events that are semantically important; clients can advise these pointcuts, and so if two modules implement the same interface, they must use these pointcuts in the same way. Clients can also advise functions that are in the interface of a module; however, this advice affects only *external* calls to these functions, not calls from within the module. Thus, clients cannot observe or depend on the way that the implementation of a module uses functions that are exposed in its interface. For example, in Figure 1, our module system would require that the pointcut apply only to calls that are not within the `shape` module itself.

To make the abstraction property precise, we define a bisimulation relation between programs. `TinyAspect`'s module system ensures that if two modules implement the same module interface, and if the implementations obey a bisimulation relation with respect to that interface, then no matter what client code is written against that interface, it will behave the same way no matter which module implementation is used. Because of this property, developers can reason separately about the correctness of the implementation and the clients of a module.

1.4 Outline

The outline of the rest of the paper is as follows. In the next section, we introduce the `TinyAspect` language through a series of examples and the formal static and dynamic semantics. In the following section, we extend `TinyAspect` with Open Modules. In Section 4, we define an equivalence relation between programs and show that the module system guarantees abstraction. Section 5 discusses the implications of the abstraction property for formal methods, modular analysis, and tool support for AOP. Section 6 discusses related work, and Section 7 concludes.

2. Formally Modelling Aspects

We would like to use a formal model of aspect-oriented programming in order to study language extensions like the module system discussed in the next section. While other researchers have used denotational semantics [23], big-step operational semantics [14], and translation systems [16, 21] to study the semantics of aspect-oriented programming, small-step operational semantics have the advantage of providing a simple and direct semantics that is amenable to syntactic proof techniques.

Jagadeesan et al. have proposed an operational semantics for the core of AspectJ, incorporating several different kinds of pointcuts and advice in an object-oriented setting [11]. These features are ideal for modeling AspectJ, but the complexity of the model makes it tedious to prove properties about the system.

Walker et al. propose a much simpler formal model incorporating just the lambda calculus, advice, and labeled hooks that describe where advice may apply [22]. Their foundational calculus does not fulfill the obliviousness requirement of AOP in that advice can only be applied to specifically labeled locations, and indeed it is not intended to model source-level AOP constructs directly. Instead, the calculus provides a set of primitives into which source-level, oblivious AOP constructs can be translated, and is thus useful for

[1]Thanks to Gregor Kiczales for pointing this out

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

Figure 2: `TinyAspect` Source Syntax

studying compilation strategies for AOP languages, and for defining AOP languages by translation.

Because the calculus of Walker et al. is considerably lower level than existing languages like AspectJ, properties that might be true at the source level of a language may not hold in the foundational calculus. Thus, a formal model that models source-level aspect constructs directly is a more effective way to investigate many properties of AOP languages.

2.1 TinyAspect

We have developed a new functional core language for aspect-oriented programming called `TinyAspect` that is intended to make proofs of source-level properties as straightforward as possible. As the name suggests, `TinyAspect` is tiny, containing only the lambda calculus with units, declarations, pointcuts, and `around` advice. `TinyAspect` directly models AOP constructs similar to those found in AspectJ, making source-level properties easy to specify and prove using small-step operational semantics and standard syntactic techniques. Although we are working in an aspect-oriented, functional setting, our system's design is inspired by that of Featherweight Java [10], which has been successfully used to study a number of object-oriented language features.

Figure 2 shows the syntax of `TinyAspect`. Our syntax is modeled after ML [17], so that `TinyAspect` programs are easy to read and understand. Names in `TinyAspect` are simple identifiers; we will extend this to paths when we add module constructs to the language. Expressions include the monomorphic lambda calculus – names, functions, and function application. To this core, we add a primitive unit expression, so that we have a base case for types. We could add primitive booleans and integers in a completely standard way. Since these constructs are orthogonal to aspects, we omit them.

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

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

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

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

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

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

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 $\tau_1 \to \tau_2$, and pointcut types representing calls to a function of type $\tau_1 \to \tau_2$.

2.2 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 and booleans have been added to illustrate the example.

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

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

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

$$
\begin{array}{llcl}
\text{Expression values} & v & ::= & \texttt{()} \mid \textbf{fn}\; x{:}\tau \texttt{ => } e \mid \ell \\
\text{Pointcut values} & p_v & ::= & \textbf{call}(\ell) \\
\text{Declaration values} & d_v & ::= & \bullet \\
 & & \mid & \textbf{val}\; x \equiv v\;\; d_v \\
 & & \mid & \textbf{pointcut}\; x \equiv p_v\;\; d_v \\
\text{Evaluation contexts} & C & ::= & \Box\, e_2 \mid v_1\, \Box \mid \textbf{val}\; x = \Box\;\; d \\
 & & \mid & \textbf{val}\; x \equiv v\;\; \Box \\
 & & \mid & \textbf{pointcut}\; x \equiv p_v\;\; \Box
\end{array}
$$

Figure 4: `TinyAspect` Values and Contexts

for a given value, looking up an argument in the cache, and storing a new argument-result pair in the cache.

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

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

2.3 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 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 ad-

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

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

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

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

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

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

Figure 5: `TinyAspect` Operational Semantics

vice 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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: `TinyAspect` Typechecking

context as defined in Figure 4.

2.4 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 is analogous to store typings in languages with references.

2.5 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 a standard substitution and weakening lemmas.

■

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

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 [22].

3. Open 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	$::=$	$\ldots \mid$ **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** $x = \Box\ d$
Module values	m_v	$::=$	**struct** d_v **end**
		$\mid$	**functor**$(x{:}\sigma)$ => m
Contexts	C	$::=$	$\ldots \mid$ **structure** $x = \Box\ d$
		$\mid$	**structure** $x \equiv m_v\ \Box$
		$\mid$	**struct** $\Box$ **end** $\mid \Box$ **:>** σ
		$\mid$	$\Box\ m_2 \mid m_v\ \Box$

Figure 7: Module System Syntax, Values, and Contexts

of ML, providing a familiar concrete syntax and benefitting from the design of an already advanced module system.

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

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

Our module system does not include abstract types, and so the abstraction property we enforce is one of implementation independence, not representation independence. The underlying problem is the same in both cases: external aspects should not be able to observe the internal behavior of module functions. Thus, we conjecture that our solution to the implementation independence problem will also enforce representation independence once abstract types are added in standard ways [17].

3.1 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.

3.2 Sealing

Our module sealing operation has an effect both at the type system level and at the operational level. At the type level, it

```
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

```
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

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

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

3.3 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.

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.

This solution, called *pointcut interfaces*, was originally proposed by Gudmundson and Kiczales as an engineering technique that can ease software evolution by decoupling an aspect from the code that it advises [9]. It is also related to the Demeter project's use of *traversal strategies* to isolate an aspect from the code that it advises [18].

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

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

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

3.4 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`,

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

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

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

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

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

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

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

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

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

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

Figure 10: Module System Operational Semantics

`pointcut`, or `struct`) to be omitted from the signature, so that clients cannot see it at all. The rule for sealing a value declaration generates a fresh label ℓ, maps that to the old value of the variable binding in η, and returns a declaration mapping the variable to ℓ. Client advice to the new label ℓ will affect only external calls, since internal references still refer to the old label which clients cannot change. The rule for pointcuts passes the pointcut value through to clients unchanged, allowing clients to advise the label referred to in the pointcut. Finally, the rules for structure declarations recursively seal any internal struct declarations, but leave functors unchanged.

3.5 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.

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

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

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

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

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

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

Figure 11: Open Modules Typechecking

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

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

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

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

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

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

Figure 12: Signature Subtyping

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.

3.6 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.

3.7 Expressiveness

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

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

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

Experiment. In a companion paper, we performed a microexperiment applying the ideas of Open Modules to Space-War, a small demonstration application distributed with AspectJ. The experiment was far too small to provide definitive results. However, we found that Open Modules support nearly all of the aspects in this program with no changes or only minor changes to the code [1].

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

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

- While our formalism supports only simple pointcuts, the design of Open Modules is compatible with the *quantification* [8] constructs of languages like AspectJ, allowing many functions within a module to be advised with a single declaration. Implementing similar functionality with conventional wrappers–without quantification–is far more tedious because a wrapper must be explicitly applied to each function.
- In Open Modules, a locally-defined aspect can implement a crosscutting concern by obliviously extending the interface of a number of modules. Wrappers cannot capture these concerns in a modular way, because each target module must be individually wrapped.
- Callbacks are invasive with respect to the implementation of a module because the implementation must explicitly invoke the callback at the appropriate points. In contrast, pointcut interfaces are non-invasive in that the pointcut is defined orthogonally to the rest of the

module's implementation, thus providing better support for separation of concerns.

These advantages illustrate how the quantification and oblivious extension provided by Open Modules distinguish our proposal from solutions that do not use aspects [8].

4. Abstraction

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

Reynolds' *abstraction* property [20] 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 [19].

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.

4.1 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.

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

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

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

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

Figure 14: Coinductive Definition of Logical Equivalence for Expressions

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.

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

$$
\begin{array}{rll}
\Lambda \vdash (\eta_1, v) \simeq (\eta_2, v) : \texttt{unit} & & \\[1em]
\Lambda \vdash (\eta_1, \texttt{fn}\ x{:}\tau \texttt{ => } e_1) \simeq (\eta_2, \texttt{fn}\ x{:}\tau' \texttt{ => } e_2) : \tau' \rightarrow \tau & \text{iff} & \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 \\[1em]
\Lambda \vdash (\eta_1, \ell) \simeq (\eta_2, \ell) : \tau & \text{iff} & \ell \notin \Lambda \\[1em]
\Lambda \vdash (\eta, \bullet) \simeq (\eta', \bullet) : (\bullet) & & \\[1em]
\Lambda \vdash (\eta, \texttt{val}\ x \equiv \ell\ \ d_v) \simeq (\eta', \texttt{val}\ x \equiv \ell\ \ d_v') : (x{:}\tau, \beta) & \text{iff} & \ell \notin \Lambda \text{ and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta \\[1em]
\Lambda \vdash (\eta, \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v) \simeq & \text{iff} & \ell \notin \Lambda \text{ and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta \\
(\eta', \texttt{pointcut}\ x \equiv \texttt{call}(\ell)\ \ d_v') : (x{:}\texttt{pc}(\tau), \beta) & & \\[1em]
\Lambda \vdash (\eta, \texttt{structure}\ x \equiv m_v\ \ d_v) \simeq & \text{iff} & \Lambda \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma, \\
(\eta', \texttt{structure}\ x \equiv m_v'\ \ d_v') : (x{:}\sigma, \beta) & & \text{and } \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta \\[1em]
\Lambda \vdash (\eta, \texttt{struct}\ d_v\ \texttt{end}) \simeq (\eta', \texttt{struct}\ d_v'\ \texttt{end}) : \texttt{sig}\ \beta\ \texttt{end} & \text{iff} & \Lambda \vdash (\eta, d_v) \simeq (\eta', d_v') : \beta \\[1em]
\Lambda \vdash (\eta_1, m_v^1) \simeq (\eta_2, m_v^2) : \sigma' \rightarrow \sigma & \text{iff} & \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 \\[1em]
\Lambda \vdash \eta_1 \simeq \eta_2 : \Sigma & \text{iff} & \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

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*
$\Lambda \vdash (\eta, \texttt{structure}\ x = m_v\ \ d) \cong$
$(\eta', \texttt{structure}\ x = m_v'\ \ d) : (x{:}\sigma, \beta)$

4.2 Proving Abstraction

In this section we outline the proof of abstraction for `TinyAspect`.

In order to prove the Abstraction theorem we will need a definition of equivalence that includes logical equivalence as a special case but explicitly relates structurally equivalent expressions such as the set of client declarations d in the Abstraction theorem. We define the structural bisimilarity relation, written $\approx_n$, to be structural equality of expressions except that closed expressions or values embedded at corresponding places in the expressions may only be logically equivalent. A bisimilarity judgment is labeled with its *complexity* n, roughly (but not exactly) denoting the number of steps in the derivation.

The formal definition is largely straightforward and is shown in Figure 15. A variable is bisimilar to itself at complexity 1, while a value is bisimilar to a logically equivalent value, also at complexity 1. Equivalent (non-value) expressions are also equivalent at complexity 1. We allow bisimilar expressions to be nested within logically equivalent contexts. It is important that the complexity of judgment be the same as the complexity of the underlying expression bisimilarity judgment (rather than adding something for the equivalent contexts) because as two bisimilar expressions execute in parallel, we will need to add additional contexts without increasing the overall complexity of the judgment.

The rest of the bisimilarity definitions are straightforward; for example, two functions are bisimilar if their bodies are, etc.

An important lemma in the proof shows that only the labels in Λ matter in the definitions of bisimilarity and logical equivalence. Intuitively, this property allows clients to apply advice to any label not in Λ, or to add new labels to the environment, without affecting equivalence relations on the underlying program.

Lemma 4 (Extension)
If $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$*, then for all* η_1', η_2' *such that*
$\ell \in domain(\eta_1) \cap \Lambda \implies \eta_1[\ell] = \eta_1'[\ell]$ *and*
$\ell \in domain(\eta_2) \cap \Lambda \implies \eta_2[\ell] = \eta_2'[\ell]$*,*
we have $\Lambda \vdash (\eta_1', E_1) \approx_n (\eta_2', E_2) : T$

Proof: [Extension] By induction on the structure of $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$. Cases *b-var* and *b-empty* are trivially satisfied. All the other cases except *b-equiv* and *b-ctx* follow directly from the induction hypothesis. The remaining two cases depend on the corresponding property holding true for logical equivalence of values and contexts, respectively.

We prove this by considering the set S of expression pairs

$$\frac{}{\Lambda \vdash (\eta, x) \approx_1 (\eta', x) : T}\ \textit{b-var}$$

$$\frac{\Lambda \vdash (\eta, V) \simeq (\eta', V') : T}{\Lambda \vdash (\eta, V) \approx_1 (\eta', V') : T}\ \textit{b-val}$$

$$\frac{\Lambda \vdash (\eta, E) \cong (\eta', E') : T \qquad E, E' \textit{ are not values}}{\Lambda \vdash (\eta, E) \approx_1 (\eta', E') : T}\ \textit{b-equiv}$$

$$\frac{\begin{array}{c}\Lambda \vdash (\eta, C) \cong (\eta', C') : T' \to T \\ \Lambda \vdash (\eta, E) \approx_n (\eta', E') : T' \qquad E, E' \textit{ are not values}\end{array}}{\Lambda \vdash (\eta, C[E]) \approx_n (\eta', C'[E']) : T}\ \textit{b-ctx}$$

$$\frac{\Lambda \vdash (\eta, e) \approx_n (\eta', e') : \tau' \qquad fv(e) = fv(e') \supset \{x\}}{\Lambda \vdash (\eta, \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e) \approx_{n+1} (\eta', \texttt{fn}\ x{:}\tau\ \texttt{=>}\ e') : T}\ \textit{b-fn}$$

$$\frac{\begin{array}{c}\Lambda \vdash (\eta, E_1) \approx_{n_1} (\eta', E_1') : T_2 \to T_1 \\ \Lambda \vdash (\eta, E_2) \approx_{n_2} (\eta', E_2') : T_2\end{array}}{\Lambda \vdash (\eta, E_1\ E_2) \approx_{n_1+n_2} (\eta', E_1'\ E_2') : T_1}\ \textit{b-app}$$

$$\frac{\Lambda \vdash (\eta, e) \approx_{n_1} (\eta', e') : \beta' \qquad \Lambda \vdash (\eta, d) \approx_{n_2} (\eta', d') : \beta'}{\Lambda \vdash (\eta, \textit{bind}\ x = e\ d) \approx_{n_1+n_2} (\eta', \textit{bind}\ x = e'\ d') : \beta}\ \textit{b-bind}$$

$$\frac{}{\Lambda \vdash (\eta, \bullet) \approx_1 (\eta', \bullet) : \bullet}\ \textit{b-empty}$$

$$\frac{\Lambda \vdash (\eta, m) \approx_n (\eta', m') : \sigma'}{\Lambda \vdash (\eta, m.x) \approx_{n+1} (\eta', m'.x) : \sigma}\ \textit{b-path}$$

$$\frac{\Lambda \vdash (\eta, d) \approx_n (\eta', d') : \sigma' \qquad d, d' \textit{ are not values}}{\Lambda \vdash (\eta, \texttt{struct}\ d\ \texttt{end}) \approx_{n+1} (\eta', \texttt{struct}\ d'\ \texttt{end}) : \sigma}\ \textit{b-struct}$$

$$\frac{\Lambda \vdash (\eta, m) \approx_n (\eta', m') : \sigma'}{\Lambda \vdash (\eta, m\ \texttt{:>}\ \sigma) \approx_{n+1} (\eta', m'\ \texttt{:>}\ \sigma) : \sigma}\ \textit{b-seal}$$

$$\frac{\Lambda \vdash (\eta, m) \approx_n (\eta', m') : \sigma' \qquad fv(m) = fv(m') \supset \{x\}}{\begin{array}{c}\Lambda \vdash (\eta, \texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m) \\ \approx_{n+1} (\eta', \texttt{functor}(x{:}\sigma)\ \texttt{=>}\ m') : \sigma \to \sigma'\end{array}}\ \textit{b-ftor}$$

Figure 15: Bisimulation Relation

(E, E') and context pairs (C, C') such that $\Lambda \vdash (\eta_1, E) \cong (\eta_2, E') : T$ (and similarly for C and C'). We wish to show that $\Lambda \vdash (\eta_1', E) \cong (\eta_2', E') : T$ for all η_1', η_2' satisfying the conditions above. We do so by showing that the rules for logical equivalence of expressions (and values and contexts) are closed with respect to the cross product of these expression pairs and environment pairs. This means that the cross product is a fixed point of the rules, and thus by the coinduction principle is within their greatest fixed point. The proof is by a case analysis on the rule used to conclude that $\Lambda \vdash (\eta_1, E) \cong (\eta_2, E') : T$.

Most of the cases are easy because they do not depend on η_1 or η_2. In fact, the only rule that does is the rule that allows evaluation according to the $\overset{\Lambda}{\mapsto}^*$ relation. But this relation only allows dependencies on the labels in Λ, which are assumed to be identical in the new environments η_1', η_2', so the proof is complete.

■

We next show that the bisimulation is preserved by substitution of logically equivalent values.

Lemma 5 (Substitution)
If $\Lambda \vdash (\eta_1) \simeq (\eta_2) : \Sigma$, $\quad \Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$, $x : T', \Sigma \vdash E_1 : T$, $\quad x : T', \Sigma \vdash E_2 : T$, *and* $\Lambda \vdash (\eta_1, V_1) \simeq (\eta_2, V_2) : T'$, *then* $\Lambda \vdash (\eta_1, \{V_1/x\}E_1) \approx_m (\eta_2, \{V_2/x\}E_2) : T$, *where* $m \leq n$.

Proof: [Substitution]
By induction on the complexity of the judgment $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$, as defined by the subscript n. The induction is simultaneous with the corresponding inductions for the pointcut substitution, preservation and equivalence lemmas, below. This simultaneous induction is necessary because the lemmas are interdependent; however, each lemma only depends on the other lemmas at complexity $n-1$, except for the equivalence lemma which depends on the preservation lemma at complexity n. Thus the overall induction is well-founded.

Base case: complexity = 1. We do a nested induction on the derivation of the judgment, with a case analysis on each rule that could result in a complexity of 1:

Subcase *b-var*: If the substitution variable is equal to the variable in the expression, the case holds because the values being substituted are equivalent, and the judgment remains at complexity 1. Otherwise, the expressions stay the same and thus remain bisimilar at complexity 1.

Subcase *b-val*: Since the values are closed, the expressions stay the same and remain bisimilar at complexity 1.

Subcase *b-equiv*: All $\cong$ judgments relate closed expressions, since otherwise the expressions would not be values or would not be able to take a step. This implies that expressions E_1 and E_2 are closed and thus are unaffected by the substitution, remaining bisimilar at complexity 1.

Subcase *b-ctx*: By the nested induction hypothesis, the substitution for the underlying expression must preserve equivalence at complexity 1. The contexts themselves must be closed and therefore unaffected by the substitution, so we can apply the *b-ctx* rule to show that after the substitution the expressions are still bisimilar at complexity 1.

Inductive case: Assume the complexity of the judgment is $n > 1$. We assume the truth of the substitution *and* preservation lemmas for size $n-1$, and prove the substitution lemma for size n with a nested induction on the derivation of $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$, with a case analysis on the last rule used.

Subcase *b-ctx*: Same as in the *b-ctx* subcase of the base case, above.

Subcase *b-fn,b-ftor*: The induction hypothesis implies that the bodies of the functions or functors remain bisimilar after substitution, with complexity $n-1$ or less. If the functions or functors are not closed values after the substitution, we can apply the rules *b-fn* or *b-ftor* to show that the functors are bisimilar with complexity n or less.

If the functions or functors become closed values as a result of the substitution, then we must show that the values are logically equivalent. By the definition of logical equivalence for function or functor values, we must show that they execute in a logically equivalent way when invoked with any pair of logically equivalent argument values of the appropriate type.

We use the induction hypothesis again to show that substitution of the argument values preserves the bisimilarity of the function bodies, at complexity $n-1$ or less. We then apply the Bisimilarity implies Equivalence lemma at complexity $n-1$ to show that the subsituted function bodies are logically equivalent. Therefore, the functions themselves are logically equivalent and thus bisimilar at complexity 1.

Other cases: All other cases hold trivially by applying the induction hypothesis. ■

There is also a version of the substitution lemma for pointcuts:

Lemma 6 (Pointcut Substitution)
If $\Lambda \vdash (\eta_1) \simeq (\eta_2) : \Sigma$, $\quad \Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$,
$x : \textbf{pc}(\tau), \Sigma \vdash E_1 : T$, $\ell \notin \Lambda$, *and* $x : \textbf{pc}(\tau), \Sigma \vdash E_2 : T$,
then $\Lambda \vdash (\eta_1, \{\texttt{call}(\ell)/x\}E_1) \approx_n (\eta_2, \{\texttt{call}(\ell)/x\}E_2) : T$.

Proof: [Pointcut Substitution] By induction on the complexity of the judgment $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$, as defined by the subscript n. The cases are the same as the proof above. ■

The most critical lemma in the proof of abstraction states that structural bisimilarity is preserved by reduction:

Lemma 7 (Bisimilarity Preservation)
If $\Lambda \vdash (\eta_1) \simeq (\eta_2) : \Sigma$, $\quad \Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$,
$\bullet, \Sigma \vdash E_1 : T$, $\quad$ *and* $\bullet, \Sigma \vdash E_2 : T$
then either:

- *E_1 and E_2 are both values such that* $\Lambda \vdash (\eta_1, E_1) \simeq (\eta_2, E_2) : T$, *or*
- *there exist* η_1', E_1', η_2', *and* E_2' *such that* $(\eta_1, E_1) \overset{\Lambda}{\mapsto}^* (\eta_1', E_1')$, $\quad (\eta_2, E_2) \overset{\Lambda}{\mapsto}^* (\eta_2', E_2')$, $\Lambda \vdash (\eta_1', E_1') \approx_m (\eta_2', E_2') : T$, *and* $\Lambda \vdash \eta_1' \simeq \eta_2' : \Sigma'$, *for some* $\Sigma' \supseteq \Sigma, m \leq n$.
- *there exist* E_1' *and* E_2' *such that* $(\eta_1, E_1) \mapsto (\eta_1, E_1')$, $(\eta_2, E_2) \mapsto (\eta_2, E_2')$ *and* $\Lambda \vdash (\eta_1, E_1') \approx_m (\eta_2, E_2') : T$ *for some* $m \leq n$.

Proof: [Bisimilarity Preservation]

By induction on the complexity n of the judgment $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$. The induction is simultaneous with the corresponding induction for the substitution lemma, above.

Base case: complexity = 1. We do a nested induction on the derivation of the judgment, with a case analysis on each rule that could result in a complexity of 1:

Subcase *b-var*: Does not apply, since E_1 and E_2 are well-typed in an environment that has no variable bindings.

Subcase *b-val*: Then E_1 and E_2 are both values such that $\Lambda \vdash (\eta_1, E_1) \simeq (\eta_2, E_2) : T$.

Subcase *b-equiv*: Perform another case analysis on the definition of $\cong$. The first subsubcase does not apply because by assumption E and E' are not values.

The second subsubcase corresponds exactly to the second case of bisimilarity preservation, observing that the resulting expressions are logically equivalent and therefore bisimilar at complexity 1.

The third subsubcase states that the two expressions will each take one step–corresponding to the third case of bisimilarity preservation–and result in logically equivalent (and therefore bisimilar) expressions at complexity 1.

Subcase *b-ctx*: By the nested induction hypothesis, bisimilarity preservation must hold for the subexpressions E and E'. We perform a case analysis on the cases of bisimilarity preservation for the subexpressions. As with the case analysis of *b-equiv*, the first subsubcase does not apply because E and E' are not values.

In the last two applicable subsubcases, we make possibly repeated application of the rule r-context to show that the entire expression takes the same reduction steps as the subexpressions, except with the contexts in place. If the resulting nested expressions are values, then they must be equivalent by the induction hypothesis, and thus we know from the definition of logically equivalent contexts that when the values are substituted into the contexts we have a logical equivalence. If the resulting nested expressions are not values, then we know the *b-ctx* rule still applies, and so the whole resulting expressions are bisimilar.

Inductive case: Assume the complexity of the judgment is $n > 1$. We assume the truth of the preservation *and* substitution lemmas for any size $m < n$, and prove the lemma for size n by nested induction on the derivation of $\Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$, with a case analysis on the last rule used.

Subcase *b-ctx*: Same as in the *b-ctx* subcase of the base case, above.

Subcase *b-fn,b-ftor*: Does not apply, since by assumption the expressions were closed and these rules apply only to open expressions.

Subcase *b-app*: If both of the subexpressions E_1 and E_2 are values, then they are logically equivalent to the corresponding values E_1' and E_2'. If E_1 is a label, then the labels refer to logically equivalent values, so bisimilarity is preserved at complexity n through this reduction step. If E_1 is a function, the definition of logical equivalence implies that function application will result in logically equivalent expressions, which are bisimilar at complexity 1. Similar analysis applies if E_1 is a functor.

If one of the subexpressions is not a value, then rule r-context must apply. We apply the same logic as in the rule *b-ctx*, using the induction hypothesis to show that the subexpressions remain bisimilar and concluding that the applications remain bisimilar as well at complexity n.

Subcase *b-bind*: If the subexpressions are not values, then the analysis used in rule *b-ctx* applies. If they are values, then one of the binding rules applies. We do a case analysis on which one applies:

Subsubcase *r-val*: The values being bound must be logically equivalent by the definition of bisimilarity. We apply the Extension lemma together with the logical equivalence of the values to show that extending the environments with the bound values preserves bisimilarity and the logical equivalence of the environments. We can then apply the Substitution lemma to show that substitution of the new label into the following definitions preserves bisimilarity. The subcase is completed by noting that the resulting val declarations are bisimilar according to rule *b-bind*.

Subsubcase *r-pointcut*: Here we apply the Pointcut Substitution lemma to show that the substituted declarations remain bisimilar. The subcase is completed by noting that the resulting pointcut declarations are bisimilar by rule r-bind.

Subsubcase *r-structure*: The substitution lemma shows that the resulting expressions remain bisimilar.

Subsubcase *r-around*: $\ell \notin \Lambda$ since it is free in the around expression, and so the values that ℓ maps to in η_1 and η_2 must be logically equivalent. Therefore, by the substitution lemma, the expressions e in the around declaration remain bisimilar after the substitution. By the Bisimilarity implies Equivalence lemma (below) at complexity $n_1 < n$, the constructed functions v' are logically equivalent. We apply the Extension lemma together with the logical equivalence of the original and constructed functions to show that extending the environments preserves bisimilarity and the logical equivalence of the environments.

Subcase *b-path*: If the module subexpression is not a value, then the analysis used in rule *b-ctx* applies. If it is a value, then by assumption the corresponding values in the structure are logically equivalent, so we apply rule r-path and the case holds.

Subcase *b-struct*: By assumption, the declarations are not values. Thus, rule r-context applies, and we can use the analysis in rule *b-app*.

Subcase *b-seal*: If the module subexpression is not a value, then the analysis used in rule *b-app* applies. If it is a value, then rule r-seal applies. It is easy to see that these new labels are logically equivalent, since they are only pointers to the corresponding old labels which are known to be logically equivalent by assumption. By applying the extension lemma we see that the extended environments remain logically equivalent, and through simple applications of the *b-struct* and *b-bind* rules we can see that the generated module values are bisimilar at a complexity that is less than or equal to n.

■

Next we show that structural bisimilarity implies logical equivalence:

Lemma 8 (Bisimilarity implies Equivalence)
If $\Lambda \vdash (\eta_1) \simeq (\eta_2) : \Sigma$, $\quad \Lambda \vdash (\eta_1, E_1) \approx_n (\eta_2, E_2) : T$,
$\bullet, \Sigma \vdash E_1 : T$, $\quad$ *and* $\bullet, \Sigma \vdash E_2 : T$
then $\Lambda \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T$

Proof: [Bisimilarity implies Equivalence]
By induction on n, simultaneous with the above lemmas. We then apply the bisimulation preservation theorem to show that bisimilar expressions at complexity n remain bisimilar under application of one of the rules in Figure 14, and the type preservation theorem to show that the result is well-typed. Since Figures 13 and 14 coinductively define logical equivalence (i.e., logical equivalence is the greatest fixpoint of these rules), and since the set of bisimilar expressions at complexity n is a fixpoint of these rules, it follows by the coinduction principle that set of bisimilar expressions at complexity n is within the set of logically equivalent expressions (i.e., within the greatest fixpoint of the rules). Therefore we have $\Lambda \vdash (\eta_1, E_1) \cong (\eta_2, E_2) : T$.

■

A final lemma stating that identical expressions are bisimilar is necessary for the final abstraction proof.

Lemma 9 (Reflexivity of Bisimulation)
If $\Lambda \vdash \eta_1 \simeq \eta_2 : \Sigma$ *and* $fl(E) \cap \Lambda = \emptyset$
then $\Lambda \vdash (\eta_1, E) \approx (\eta_2, E) : T$.

Proof: [Reflexivity of Bisimulation]
By induction on the structure of E, with a case analysis on the last syntactic construct used. Each case is trivial because there is a bimsimulation rule for each syntactic construct.

■

We now prove the abstraction theorem:

Proof: [Abstraction]
Since $\Lambda \vdash (\eta, m_v) \simeq (\eta', m'_v) : \sigma$, we know by the definition of $\approx$ and the Reflexivity of Bisimulation that
$\bullet \vdash (\bullet, \texttt{structure}\ x = m\ \ d) \approx (\bullet, \texttt{structure}\ x = m'\ \ d) : (x{:}\sigma, \beta)$.
We complete the proof by applying the Bisimilarity implies Equivalence lemma to show that the original expressions must also have been logically equivalent.

■

4.3 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.

5. Discussion

Formal Methods. The abstraction theorem and the definition of logical equivalence for Open Modules have broader implications for modular reasoning about aspect-oriented programming. For example, we might ask how one might specify the required behavior of a module and how we might prove the module meets that behavior in the presence of aspects. The definition of logical equivalence implies that a complete behavioral specification must include not just pre- and post-conditions for the functions in the module interface, but also the ordered trace of pointcuts that are triggered by calling each function in the interface. This trace must include the argument that will be passed to advice on the pointcut, as well as the specification of `proceed` for that pointcut. Abstraction states that if we can show that an implementation is bisimilar to the trace specification, then that implementation will be indistinguishable from other implementations that meet the specification.

Modular Analysis. Another interesting question is how one might perform modular analysis in the presence of Open Modules. Typically a modular analysis will run on a module, producing a summary that can be used when analyzing other modules. The definition of logical equivalence suggests that the analysis summary should describe the properties of the functions in the interface in terms of how they interact with the exposed pointcuts. For example, a modular escape analysis might conclude that a function `f` in the interface of a module does not capture its argument, provided that advice to a pointcut `p` does not capture *its* argument.

Tool Support. The AspectJ plugin for Eclipse provides integrated development environment support for programming in AspectJ. Perhaps the most important feature is showing, for each function, which aspects might apply to that function. This means that a programmer can more effectively predict the composed behavior of the function and its aspects, and can ensure that changes to the code preserve that behavior.

Open Modules and the abstraction theorem provide insight into why IDE support is so helpful for AspectJ. The description of which aspects apply to which functions is like a pointcut in the interface of the module. Just as in Open Modules a developer must make sure the semantics of the pointcut are maintained as a module evolves, so an AspectJ developer must ensure that changes to a package do not adversely affect the aspects that apply to it.

Open Modules also suggest a way to improve this tool support. In order to support more effective software evolution, the tool should project an editable view of each pointcut into the interface of the modules to which it applies. That way, when a developer makes a change to a module, she can also locally change affected pointcuts so that their semantics are maintained. These local changes will then be propagated to the original pointcut definition. Thus the module developer can reason about and evolve that module exactly as she would in the Open Module system, while the aspect developer can take advantage of the full expressiveness of AspectJ.

This scenario assumes that the tool can see and edit all aspects that might apply to a piece of base code. Thus, a tool can only provide a full solution to the information hiding problem in a setting where a monolithic application or library is being developed by a tightly integrated team.

In the more general setting of component-based development, no tool can predict all possible aspects that might be used by the client of a component. Thus it is very difficult to be sure that changes to the component will not affect is clients, or to ensure that safety properties of the component will hold no matter what aspects a client might apply. Here, Open Modules can be used to describe the ways in which a component may be extended by clients, while hiding implementation-specific details and protecting the component properties. Clients that obey the Open Modules specification of the component get a guarantee that future changes to the component will not break their programs, and also that their aspects will not violate important invariants of the component. Clients can turn off module checking and bypass the Open Modules interface, but will lose these guarantees.

Thus, Open Modules complement tool support for aspect-oriented programming, providing the benefits of information hiding in the setting of component-based development, while allowing developers in an integrated team to gain the full benefits of aspects.

6. Related Work

Formal Models. The most closely related formal models are the foundational calculus of Walker et al. [22], and the model of AspectJ by Jagadeesan et al. [11], both of which were discussed in the beginning of Section 2.

In other work on formal systems for aspect-oriented programming, Lämmel provides a big-step semantics for a method-call interception extension to object-oriented languages [14]. Wand et al. give an untyped, denotational semantics for advice advice and dynamic join points [23]. Masuhara and Kiczales describe a general model of crosscutting structure, using implementations in Scheme to give semantics to the model [16]. Tucker and Krishnamurthi show how scoped continuation marks can be used in untyped higher-order functional languages to provide static and dynamic aspects [21].

Aspects and Modules. Dantas and Walker are currently extending the calculus of Walker et al. to support a module system [6]. Their type system includes a novel feature for controlling whether advice can read or change the arguments and results of advised functions. In their design, pointcuts are first-class, providing more flexibility compared to the second-class pointcuts in `TinyAspect`. This design choice breaks 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 [15]. 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 [4].

Our module system is based on that of standard ML [17]. `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 [5].

7. 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.

8. Acknowledgments

I thank Gregor Kiczales, Mira Mezini, 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.

9. REFERENCES

[1] J. Aldrich. Open Modules: Reconciling Extensibility and Information Hiding. In *AOSD workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT '04)*, March 2004.

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

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

[6] D. S. Dantas and D. Walker. Aspects, Information Hiding and Modularity. Princeton University Technical Report TR-696-04, 2004.

[7] R. E. Filman. What is Aspect-Oriented Programming, Revisited. In *Advanced Separation of Concerns*, July 2001.

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

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

[12] 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.

[13] 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.

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

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

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

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

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

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

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

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

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

[23] 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.