Open Modules: A Foundation for Modular Aspect-Oriented Programming

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

ABSTRACT

Aspect-oriented programming systems provide powerful mechanisms for separating concerns, but understanding how these concerns interact can be challenging. In particular, many aspect-oriented programming constructs can violate encapsulation, making it difficult to reason about a module’s correctness in isolation.

In this paper, we introduce Open Modules, a mechanism for enforcing a strong form of encapsulation while supporting much of the extensibility provided by languages like AspectJ. Open Modules provide extensibility by allowing clients to advise the interface of a module, but enforce encapsulation by protecting internal function calls made within a module. A module can expose semantically important internal events to clients through pointcuts in its interface. The module’s implementation can change without affecting client advice as long as the semantics of the pointcuts in its interface are preserved.

Using TinyAspect, a formally defined language modeling core aspect-oriented programming constructs, we define the semantics of Open Modules and prove type soundness. We use a notion of bisimulation to show that Open Modules enforce Reynold’s abstraction theorem, a strong encapsulation property.

1. INTRODUCTION

Aspect-oriented programming languages provide powerful constructs that enable programmers to separate concerns more cleanly in their code. This separation of concerns makes it easier to understand and evolve programs, because all the code that deals with a particular issue is modularized.

However, the power of aspect-oriented programming constructs comes at a cost in reasoning separately about code. For example, Parnas argues that a primary goal of decomposing systems into separate modules is to hide design decisions that are likely to change [15]. Module systems in conventional languages exist in order to enforce abstraction, the property that clients cannot depend on the implementation details of a module [16]. Thus, a module’s implementation can be changed without affecting clients.

Unfortunately, aspect-oriented programming constructs can break the abstraction property enforced by existing module systems. For example, consider the AspectJ code in Figure 1. The List class keeps track of a linked list of objects. New objects can be added with the add method, and the addAll method allows the user to add the elements of one list to another list. Because calculating the size of a linked list takes linear time, the Size aspect was written to cache the list’s size. This aspect introduces the size field into the List class. It also declares after advice on both methods of the list, which increments the size field after add executes.

The problem with this code is that the aspect code depends on the implementation of the list, not just the list’s interface. For example, if the implementation of List is changed so that addAll calls add in order to add each element to the list, then advice will be executed both on the original call to addAll and on the recursive call to add, and size will be incremented by twice the number of elements actually added.

This is a simple example, and in fact the Size aspect could...
have been written in a way that does not depend on the implementation of List. However, it illustrates that in AspectJ, client code can be written that depends on the internal implementation details of a class, violating the abstraction boundary of the class. Because abstraction is violated, programmers cannot reason separately about the correctness of a module and its clients. The absence of separate reasoning makes it more difficult for programmers to build, debug, and change programs, and also makes it more difficult for compilers and tools to effectively analyze and compile aspect-oriented programs.

1.1 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. As the name suggests, TinyAspect is tiny, containing only the lambda calculus with units, declarations, pointcuts, and around advice. Its small size makes TinyAspect ideal for investigating the semantics of new features aspect-oriented languages (such as the module system described in this paper). TinyAspect has a source-level syntax similar to languages like ML, making the system more accessible compared to type-theoretic approaches. The meaning of TinyAspect programs is defined by a small-step operational semantics along with syntax-directed typing rules, and we show that the type system is sound. Finally, advice is second-class and declarative as in AspectJ, permitting stronger reasoning compared to systems with first-class advice.

The second contribution of this paper is an extension of TinyAspect with a parameterized module system that supports separate reasoning about code. TinyAspect’s module system relies on a key insight about modularity in aspect-oriented programming systems: 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 a 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 in Size apply only to calls that are not within the List itself.

Our module system supports many use cases of aspect-oriented programming: unanticipated advice to the interface of a module, advice to anticipated pointcuts in the implementation of a module (through pointcuts in the module’s interface), and unanticipated advice to a module whose source code can be modified (by adding a pointcut to the module’s interface). Our system does not support use cases where the aspect is unanticipated and involves the implementation details of a module whose source code cannot be changed. We hypothesize that most use cases for aspects fall into the categories supported by our system. In a companion paper [1] we describe a principled language extension and methodology for handling the remaining cases.

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 have functionally equivalent semantics 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.

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 a module system. In Section 4, we define an equivalence relation between programs and show that the module system guarantees abstraction. Section 5 discusses related work, Section 6 discusses future work, and Section 7 concludes.

2. THE TINYASPECT LANGUAGE

The goal of TinyAspect is to capture the essence of AOP languages like AspectJ in as simple a setting as possible, in order to make it easy to prove properties about programs. Although we are working in an aspect-oriented, functional setting, our system’s design is inspired by that of Featherweight Java [6], 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 [14], 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.

We follow real aspect-oriented languages such as AspectJ [9] by making advice declarative and second-class. A declaration is either the empty declaration, or a value binding, a pointcut binding, or advice followed by another declaration. The val declaration gives a static name to a function or 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

![Figure 2: TinyAspect Source Syntax](image-url)
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 form n is just an alias for a previous pointcut declaration n.

Finally, 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 τ1 → τ2, and pointcut types representing calls to a function of type τ1 → τ2.

2.1 Fibonacci Caching Example

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

Figure 3 shows the TinyAspect code for the Fibonacci function. Because our language has around advice, we do not need recursion as a primitive in the language, although a more realistic language would probably provide it. We assume integers and booleans have been added to illustrate the example. The base case of the fib function simply returns 1.

The fib function has no way to call itself, so it cannot include the recursive part of the definition. Instead, we define around advice on calls to 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 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 naive way.

One of the advantages of organizing a program as a sequence of declarations rather than an unordered set of classes (as in Java and AspectJ) is that the ordering of declarations provides a natural precedence relation between aspects, as this example illustrates.

2.2 Operational Semantics

We define the semantics of TinyAspect more precisely as a set of small-step reduction rules. These rules translate a series of source-level declarations into the values shown in Figure 4.

Expression-level values include the unit value and 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 ⊑ 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 \((\eta, e)\) of an advice environment \(\eta\) (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 \(\eta\), and we denote the functional update of an environment as \(\eta' = [\ell \mapsto v']\eta\). The reduction judgment is of the form \((\eta, e) \rightarrow (\eta', e')\), read, “In advice environment \(\eta\), expression \(e\) reduces to expression \(e'\) with a new advice environment \(\eta'\).”

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 \(\ell\) 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 \(r\text{-lookup}\) to look up the label’s value in the current environment.

The next three rules reduce declarations to “declaration values.” The \(\text{val}\) 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) \(d\). The \(\text{pointcut}\) declaration simply substitutes the pointcut value for the variable \(x\) in subsequent declaration(s).

The \(\text{around}\) declaration looks up the advised declaration \(\ell\) in the current environment. It places the old value for the binding in a fresh label \(\ell'\), and then re-binds the original \(\ell\) to the body of the advice. Inside the advice body, any references to the special variable \(\text{proceed}\) are replaced with \(\ell'\), 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 \(\text{proceed}\).

The last rule shows that reduction can proceed under any context as defined in Figure 4.

2.3 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 \(\Gamma\) and declaration context \(\Sigma\) expression \(e\) has type \(\tau\).” Here \(\Gamma\) maps variable names to types, while \(\Sigma\) 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 \(\Gamma\) and \(\Sigma\), 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 \(\beta\) to declarations, where \(\beta\) is a sequence of variable to type bindings. The base case of an empty declaration has an empty signature. For \(\text{val}\) bindings, we ensure that the expression is well-typed at some point. Pointcuts are similar, but the rule ensures that the expression \(p\) is well-typed.
module signatures of the form $m : sig x \Rightarrow \sigma \Rightarrow m$. 

First-order module expressions include a name, a $struct$ with a list of declarations, and an expression $m ::> \sigma$ 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 $\sigma$ 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 [14].

3.1 Fibonacci Revisited

The Fibonacci function is now encapsulated inside the $Math$ module. The module implements caching by instantiating a 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 hides all members of a module that are not in the signature $\sigma$—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.

This module system design also means that if an aspect (such as $Cache$) depends on the implementation details of a module, it cannot be applied from outside of a sealed module unless that module exports an appropriate pointcut. For example, if $Cache$ were applied from the outside of $Math$, we would not be able to tell whether or not caching had been applied, even if $Cache$ placed advice on $Math.fib$.
structure Graphics = struct
  val createShape = fn ...
  val move = fn ...
  val animate = fn ...
...
  pointcut positionChange = call(move)
end

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

it would cache clients’ initial calls to the function, but not the recursive internal calls. Removing the sealing from Match would allow Cache to operate properly from outside the module, but it would also allow client code to observe the details of the implementation through advice. Thus, our sealing operation limits what a client aspect can observe, but in return guarantees that the implementation of a module can be changed without affecting clients. We argue that these limits are a small price to pay for separate reasoning; experience with real systems will be needed to understand the costs and benefits in practice.

3.3 Exposing Semantic Events with Pointcuts

Sometimes a module may want to expose internal semantic events to clients. A module can do this in a principled way by exporting a pointcut in its signature. For example, Figure 9 describes a graphics library that includes functions to create shapes, move and animate them, etc. Clients of the library might be interested in being notified whenever shapes move. In a language like AspectJ, clients would have to advise internal functions, and thus would break if implementation details changed (As in the List example from Figure 1, the implementation of animate may or may not rely on move).

In TinyAspect, clients 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 positionChange pointcut. Because this pointcut is part of the graphics library’s interface, anyone who changes the implementation of the library must ensure that semantics of the pointcut remain the same. For example, if animate initially uses move to change the location of shapes, but is modified to use some internalMove function, then the positionChange pointcut must be modified to include calls to internalMove.

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.

3.4 Operational Semantics

Figure 10 shows the operational semantics for the module constructs in TinyAspect. 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, \( \text{seal} \), 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 \( \eta \), a list of declarations \( \ell \), and the sealing declaration signature \( \beta \). The operation computes a new environment \( \eta' \) and new list of declarations \( \ell' \). 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 environment, resulting in another empty list of declarations and an unchanged environment $\eta$. The second rule allows a declaration $\text{bind } x \equiv v$ (where $\text{bind}$ represents one of $\text{val}$, $\text{pointcut}$, or $\text{struct}$) 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 $\ell$, maps that to the old value of the variable binding in $\eta$, and returns a declaration mapping the variable to $\ell$. Client advice to the new label $\ell$ 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 apply 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 $\sigma$ of the bound module. The rule for $\text{struct}$ simply puts a $\text{sig}$ wrapper around the declaration signature. The rules for sealing and functor application allow a module to be passed into a context where a supertype of its signature is expected.

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

3.6 Type Soundness

We now state progress and preservation theorems for $\text{TinyAspect}$. The theorems quantify over expressions, declarations, and modules using the metavariable $E$, and quantify over types, declaration signatures, and module 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 $\eta', E'$ 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. Because of signature subtyping, the type preservation theorem allows a module to reduce to a subtype of its original signature.

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 $\bullet, \Sigma' \vdash E' : T'$ where $T' \prec T$ and $\Sigma' \vdash \eta'$.

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

Together, progress and type preservation imply type soundness. Soundness means that there is no way that a well-typed $\text{TinyAspect}$ program can get stuck or “go wrong” because it gets into some bad state.

4. ABSTRACTION

The example programs in Section 3 are helpful for understanding the benefits of $\text{TinyAspect}$’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 [16] 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 [15].

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

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

### 4.1 Formalizing Abstraction

We can define abstraction formally using judgments for observational equivalence of values, written

$$\Lambda \vdash (\eta, V) \simeq (\eta', V') : T$$

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

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

The most interesting rule is the one for function values. Two function values are equivalent if for any observationally equivalent argument values $v_1$ and $v_2$, they produce equivalent results. A similar rule is used for observational equivalence of functors.

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

Figure 14 shows the rules for observational equivalence of expressions. Two expressions are equivalent if they are equivalent values. Otherwise, the expressions must look up the same sequence of labels in $\Lambda$ while either diverging or reducing to observationally equivalent values (since client arguments can use advice to observe lookups to labels in $\Lambda$).

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

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

**Theorem 3 (Abstraction)**

If $\Lambda \vdash (\bullet, m_\omega) \cong (\bullet, m'_\omega) : \sigma$, then for all $d$ such that $x:\sigma; \bullet \vdash d : \beta$ we have $\Lambda \vdash (\bullet, \text{structure } x = m_\omega, d) \cong (\bullet, \text{structure } x = m'_\omega, d) : (x:\sigma, \beta)$

### 4.2 Applying Abstraction

Figure 14: TinyAspect Observational Equivalence for Expressions

```haskell
structure Fib1 = struct
  val fib = fn x:int => helper x
  end

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

end :> sig
fib : int->int
end

structure Fib2 = struct
  val helper = fn x:int => 1
  end

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

end :> sig
fib : int->int
end
```

Figure 15: Two equivalent modules that define the Fibonacci function
The abstraction theorem can be used to show that two different implementations of a module are equivalent and thus interchangeable. For example, Figure 15 shows two definitions of the Fibonacci function. The first one uses recursion directly to compute the result, while the second one invokes a helper function. Since we have sealed both modules, it is easy to prove that they are equivalent. Clients can only advise the fresh label exported by the sealed modules, which doesn’t affect the internal semantics of the module at all. Therefore, we can prove that the modules are equivalent by showing that the fib functions always return the same value when passed the same argument. A simple proof by induction on the argument value will suffice.

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

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

The Fibonacci example is simplistic in that it does not export any pointcuts to clients. However, similar equivalence properties can be proven in the presence of pointcuts, if it can be shown that two modules always treat pointcuts in an identical way, despite other differences in implementation behavior.

4.3 Proving Abstraction

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

In order to prove the Abstraction theorem we will need a induction hypothesis that is stronger than the \( \equiv \) relation. We define the structural congruence relation, written \( \simeq \), to be

\[
\Lambda \vdash (\eta, v) \simeq (\eta', v') : \text{unit}
\]

\[
\Lambda \vdash (\eta, v_1) \simeq (\eta_2, v_2) : \tau' \rightarrow \tau
\]

\[
\Lambda \vdash (\eta, \text{val } x \equiv \ell \ d_v) \simeq (\eta', \text{val } x \equiv \ell \ d_v') : (x: \tau, \beta)
\]

\[
\Lambda \vdash (\eta, \text{pointcut } x \equiv \ell \ d_v) \simeq (\eta', \text{pointcut } x \equiv \ell \ d_v') : (x: \tau, \beta)
\]

\[
\Lambda \vdash (\eta, \text{structure } x \equiv m_v \ d_v) \simeq (\eta', \text{structure } x \equiv m_v' \ d_v') : (x: \sigma, \beta)
\]

\[
\Lambda \vdash (\eta, \text{struct } d_v \text{ end}) \simeq (\eta', \text{struct } d_v' \text{ end}) : \text{sig } \beta \text{ end}
\]

\[
\Lambda \vdash (\eta, m_v) \simeq (\eta', m_v') : \sigma_1 \rightarrow \sigma_2
\]

\[
\Lambda \vdash (\eta, V) \simeq (\eta', V') : T \quad V \text{ closed} \quad V' \text{ closed}
\]

\[
\Lambda \vdash (\eta, V) \approx (\eta', V') : T
\]

\[
\Lambda \vdash (\eta, E) \approx (\eta', E') : T
\]

\[
\Lambda \vdash \text{if } (\eta, e) \approx (\eta', e') : \tau' \rightarrow \tau
\]

\[
\Lambda \vdash \text{if } (\eta, E_1) \approx (\eta', E_1') : T_2 \rightarrow T_3
\]

\[
\Lambda \vdash \text{if } (\eta, E_2) \approx (\eta', E_2') : T_2
\]

\[
\Lambda \vdash \text{if } (\eta, E_1 \ E_2) \approx (\eta', E_1' \ E_2') : T_1
\]

\[
\Lambda \vdash (\eta, \text{bind } x = e \ d) \approx (\eta', \text{bind } x = e' \ d') : \beta
\]

\[
\Lambda \vdash (\eta, m) \approx (\eta', m') : \sigma'
\]

\[
\Lambda \vdash (\eta, m, x) \approx (\eta', m', x) : \sigma
\]

\[
\Lambda \vdash (\eta, d) \approx (\eta', d') : \sigma'
\]

\[
\Lambda \vdash (\eta, \text{struct } d \text{ end}) \approx (\eta', \text{struct } d' \text{ end}) : \sigma
\]

\[
\Lambda \vdash (\eta, m) \approx (\eta', m') : \sigma'
\]

\[
\Lambda \vdash (\eta, m : > \sigma) \approx (\eta', m' : > \sigma) : \sigma
\]

\[
\Lambda \vdash (\eta, m) \approx (\eta', m') : \sigma'
\]

\[
\Lambda \vdash (\eta, \text{functor}(x: \sigma) \Rightarrow m) \\
\approx (\eta', \text{functor}(x: \sigma) \Rightarrow m') : \sigma \rightarrow \sigma'
\]

Figure 13: TinyAspect Observational Equivalence for Values

Figure 16: TinyAspect Structural Congruence
fuction bodies implies observational equivalence of the functions:

Lemma 4 (Congruence implies Function Equivalence)
If \( \Lambda \vdash (\eta_1, e_1) \approx (\eta_2, e_2) : T \) and \( \text{fn } x : T \Rightarrow e_1 \) and \( \text{fn } x : T \Rightarrow e_2 \) are closed values, then \( \Lambda \vdash (\eta_1, \text{fn } x : T \Rightarrow e_1) \approx (\eta_2, \text{fn } x : T \Rightarrow e_2) : T \). Likewise, if \( \Lambda \vdash (\eta_1, m_1) \approx (\eta_2, m_2) : \sigma \) and \( \text{functor}(x : \sigma) \Rightarrow m_1 \) and \( \text{functor}(x : \sigma) \Rightarrow m_2 \) are closed values, then \( \Lambda \vdash (\eta_1, \text{functor}(x : \sigma) \Rightarrow m_1) \approx (\eta_2, \text{functor}(x : \sigma) \Rightarrow m_2) : \sigma \rightarrow \sigma' \).

Proof: [Congruence implies Function Equivalence]
By induction on the complexity of \( e_1 \) and \( e_2 \), where complexity is the size of the structurally equal parts of \( e_1 \) and \( e_2 \) (that is, not including the size of embedded observationally equivalent values).

For primitive expressions \( x \) and \( () \), the theorem holds trivially.

For more complex expressions, we apply two observationally equivalent values \( v_1 \) and \( v_2 \) to \( \text{fn } x : T \Rightarrow e_1 \) and \( \text{fn } x : T \Rightarrow e_2 \), respectively. When the reduction step occurs, the bodies of all closed function values within the resulting expressions are less complex than \( e_1 \) and \( e_2 \) were (since their non-value parts are substructures of \( e_1 \) and \( e_2 \)), and thus we can assume by the induction hypothesis that all closed function values embedded in the substituted expressions are observationally equivalent (including, of course, \( v_1 \) and \( v_2 \)).

We can show that the theorem is valid by showing that all execution steps obey the definition of observational equivalence, and in addition preserve the invariant that the two expressions are structurally congruent. The two expressions take corresponding steps until a function application is reached. We know these functions are observationally equivalent, as are the argument values; thus the functions execute in an observationally equivalent way until they either both diverge or both result in another observationally equivalent value. At this point, the two executions once again take corresponding steps, and so on.

A similar argument proves the lemma for functors.

We next show that congruence is preserved by substitution.

Lemma 5 (Substitution)
If \( \Lambda \vdash (\eta_1, E_1) \approx (\eta_2, E_2) : T \) and \( x : T' \) is a free value in \( E_1 \) and \( E_2 \), and \( \Lambda \vdash (\eta_1, V_1) \approx (\eta_2, V_2) : T' \), then \( \Lambda \vdash (\eta_1, \{V_1/x\}E_1) \approx (\eta_2, \{V_2/x\}E_2) : T \).

Proof: [Substitution]
By induction on the structure of \( E_1 \) and \( E_2 \).

Every case in the induction is trivial except for the case for functions or functors, which get tricky if the function/functor becomes closed due to the substitution. In this case we appeal to the value equivalence lemma above to show that the new values are observationally equivalent.

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

Lemma 6 (Congruence Preservation)
If \( \Lambda \vdash (\eta_1, E_1) \approx (\eta_2, E_2) : T \) then either \( (\eta_1, E_1) \) and \( (\eta_2, E_2) \) both diverge, or else there exist \( \eta'_1, E'_1, \eta'_2, \) and \( E'_2 \) such that \( (\eta_1, E_1) \rightarrow^\ast (\eta'_1, E'_1) \) and \( (\eta_2, E_2) \rightarrow^\ast (\eta'_2, E'_2) \) and \( \Lambda \vdash (\eta_1, E_1) \approx (\eta_2, E_2) : T \).

Proof: [Congruence implies Equivalence]
By induction on the derivation of the reductions applicable to \( (\eta_1, E_1) \) and \( (\eta_2, E_2) \), with a case analysis on the rule to be applied. Since reduction cannot take place within a value, and since \( E_1 \) and \( E_2 \) are structurally equivalent except for values, the applicable reduction rules in each case must be the same.

Case r-app: By the definition of \( \approx \) the function and argument values at the corresponding application points in \( E_1 \) and \( E_2 \) must be observationally equivalent. By the definition of observational equivalence for function values and expressions, either both applications must diverge in an observationally equivalent way or they must reduce to observationally equivalent values.

Case r-lookup: By the definition of \( \approx \) the labels refer to observationally equivalent values, so structural congruence is preserved through this reduction step.

Case r-path: Follows immediately by the induction hypothesis.

Case r-context: Follows immediately by the induction hypothesis.

Next we show that structural congruence implies observational equivalence:

Lemma 7 (Congruence implies Equivalence)
If \( \Lambda \vdash (\eta_1, E_1) \approx (\eta_2, E_2) : T \) then \( \Lambda \vdash (\eta_1, E_1) \approx (\eta_2, E_2) : T \).
By Congruence Preservation we know that the congruent expressions will take a sequence of steps and result in another congruent expression. By inspection we observe that the sequence of steps developed in that lemma is consistent with the definition of observational equivalence. We complete the proof by observing that execution must either continue indefinitely or result in values, which must be observationally equivalent by the definition of congruence.

We now prove the abstraction theorem:

**Proof:** [Abstraction]

Since \( \Lambda \vdash (\eta, m_s) \approx (\eta', m'_s) : \sigma \), we know by the definition of \( \approx \) that \( \bullet \vdash (\bullet, \text{structure } x = m \ d) \approx (\bullet, \text{structure } x = m' \ d) : (x \sigma, \beta) \).

We complete the proof by noting that structural congruence implies observational equivalence, and therefore the original expressions must also have been observationally equivalent.

5. RELATED WORK

Related work falls into three major categories: formal models of aspect-oriented programming languages, module systems for aspect-oriented programming languages, and advanced module system research.

5.1 Formal Models of Aspects

Walker et al. model aspects using an expression-oriented functional language that includes the lambda calculus, labeled join points, and advice [20]. They show that their model is type-safe, but they model around advice using a low-level exception construct and so their soundness theorem includes the possibility that the program could terminate with an uncaught exception error. TinyAspect guarantees both type safety and a lack of run-time errors because it models advice with high-level constructs similar to those in existing aspect-oriented programming languages. In addition, the declarative nature of TinyAspect allows us to easily explore modularity and prove an abstraction result.

In concurrent work, Dantas and Walker extend this calculus to support a module system [5]. Their type system includes a novel feature for controlling whether advice can read or change the arguments and results of advised functions. In their design, pointcuts are first-class, providing more flexibility compared to the second-class pointcuts in TinyAspect. This design choice breaks abstraction and thus separate reasoning, however, because it means that a pointcut can escape from a module even if it is not explicitly exported in the module’s interface. In their system, functions can only be advised if this is planned in advance; in contrast, TinyAspect allows advice on all function declarations, providing unplanned extensibility without compromising abstraction.

Jagadeesan et al. describe an untyped object-oriented aspect calculus modeling many of the features of AspectJ [8]. Their formal model is much richer than ours, capturing complex pointcuts and different forms of advice in a rich subset of Java. In concurrent work, they are extending their calculus to a typed setting and proving a type soundness theorem [7]. TinyAspect is intentionally much more minimal than their aspect calculus, so that it is easy to investigate language extensions such as a module system and prove properties such as abstraction.

In other work on formal systems for aspect-oriented programming, Lämmel provides a big-step interception extension to object-oriented languages [10]. Wand et al. give an untyped, denotational semantics for advice and dynamic join points [21]. Masuhara and Kiczales describe a general model of crosscutting structure, using implementations in Scheme to give semantics to the model [12]. Tucker and Krishnamurthi show how scoped continuation marks can be used in untyped higher-order functional languages to provide static and dynamic aspects [19].

5.2 Aspects and Modules

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 [11]. 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 abstract property.

AspectJ [9] extends Java’s encapsulation mechanisms, protecting private methods from access by external aspects. However, AspectJ does not enforce abstraction, because internal calls to public methods may still be advised by external aspects. Furthermore, an aspect can get around even the limited encapsulation mechanism by declaring itself to be privileged. Thus, AspectJ’s design provides only minimal encapsulation, but gives programmers maximum flexibility in writing aspects.

A different approach to reasoning about code interactions in aspect-oriented programs is to provide an analysis that shows how aspects might affect source code or each other. For example, the Eclipse plugin for AspectJ includes a view showing which aspects affect each line of source code, and researchers have studied more sophisticated analyses [17, 18]. These analyses, however, do not prevent abstraction violations in the way that our module system does.

Clifton and Leavens propose to modularize reasoning about aspects using the concepts of observers and assistants [3]. Observers can observe a module, but not change its semantics, while assistants can change a module’s behavior, but only with that module’s permission. TinyAspect’s abstraction property enforces a stronger barrier between a module and its clients, because even “observer aspects” cannot depend on the internal implementation details of a module. Observers and assistants have a complementary advantage, supporting stronger reasoning about how different aspects might interact.

Some aspect-oriented systems support open classes, allowing programmers to add methods to a class from the outside. Clifton et al. describe how to add open classes to Java [4], while McDirmid et al. show how recursive, parameterized modules can be used in a pattern to achieve the same goal [13]. Our work complements these systems by showing how to support advice in a modular way.

5.3 Module System Research

Our work applies research in advanced module systems to aspect-oriented programming languages. Our abstraction property is based on Reynolds’ abstraction theorem [16]. Our
module system is based on that of standard ML [14]. Our sealing construct is similar to the freeze operator in the module calculus of Ancona and Zucca, which closes a module to extension [2].

6. FUTURE WORK

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

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 extended the language with a parameterized module system, 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. REFERENCES