Aspects, Information Hiding and Modularity

Daniel S. Dantas
Princeton University
ddantas@cs.princeton.edu

David Walker*
Princeton University
dpw@cs.princeton.edu

ABSTRACT

Aspect-oriented programming languages such as AspectJ provide a new way to separate out and consolidate code for debugging, profiling, distribution and other tasks that would otherwise become tangled with the main-line computation. Without aspects, this code can be difficult to understand and maintain. Unfortunately, while aspects purport to provide a new form of modularity, they also defeat the purpose of existing information hiding and modularity mechanisms.

We have developed a new aspect-oriented programming language, AspectML, that allows programmers to control information hiding and access to the internals of a module through a simple static type system. Using our mechanisms, programmers can prevent aspects from interfering with local invariants or reading information that should be kept private to a module. Consequently, AspectML is the first aspect-oriented language that interoperates safely and effectively with rich module systems.

1. INTRODUCTION

Aspect-oriented programming languages (AOPL) such as AspectJ [6] allow programmers to specify both *what* computation to perform as well as *when* to perform it. For example, AspectJ makes it easy to implement a profiler that records statistics concerning the number of calls to each method: The *what* in this case is the computation that does the recording and the *when* is the instant of time just prior to execution of each method body. In aspect-oriented terminology, the specification of *what* to do is called *advice* and the specification of *when* to do it is called a *point cut*. A collection of point cuts and advice organized to perform a coherent task is called an *aspect*.

The profiler described above could be implemented without aspects by inserting the profiling code into the body of each method directly. However, when the programmer does the insertion manually, at least two problems can occur. First, it is no longer easy to adjust *when* the appropriate advice should be run, requiring the programmer to explicitly relocate the profiling code. Second, the profiled code becomes "tangled." In other words, the main computation source code is interleaved with profiling code, making the program difficult to read and maintain. The problem gets much worse when code for several different tasks such as profiling, debugging, distribution, access control and others is all mixed together in the same place. Aspects allow maintaining program code with debugging, profiling, and other extensions late in the development cycle with less risk of entanglement.

Aspect-oriented programming already has a significant following in software engineering circles, has recently been featured in Communications of the ACM [2], has its own annual conference (AOSD), a workshop on foundations (FOAL), and is a significant new focus of a variety of traditional object-oriented programming language conferences including ECOOP and OOPSLA. However, despite the recent popular success of AOPL, they suffer from some very serious drawbacks. In fact, some researchers feel that aspects may end up costing the software industry millions of dollars if they become widely used in their current form. The central problem is that although AOPL like AspectJ purport to deliver a new form of modularity, they also undermine existing modularity and abstraction mechanisms. For instance, AspectJ allows aspects, through the use of the *privileged* keyword, to access the private components of a class or package with potentially destructive consequences. More specifically:

- A client can determine the existence of fields and methods in an implementation that would otherwise be kept private. Using this information, the client can develop code with unwanted dependencies on the underlying implementation. If the implementation changes, the client code may break. In short, it becomes impossible for a implementation to establish and enforce representation independence properties.
- Clients can also read fields of objects and data that flows between methods in an otherwise private implementation. Not only are there ramifications for representation independence as above, but also for security as no data value can be kept secret.
- Most dangerous of all, client code can make arbitrary

*This research was supported in part by ARDA Grant no. NBCHC030106 on *Assurance-Carrying Components* and National Science Foundation CAREER grant No. CCR-0238328 on *Programming Languages For Secure and Reliable Component Software*.

PLDI 2004

changes to the fields of objects, as well as the arguments to and results from methods. Such changes can disrupt any and all local invariants a programmer may try to maintain. The result is that client code and implementation code can become too closely coupled to be maintained as separate abstractions.

In summary, current aspect-oriented programming languages do not provide sufficient mechanisms to protect one piece of code from another. Consequently, debugging and maintaining aspect-oriented code can be extraordinarily difficult and extremely costly. Moreover, current aspect-oriented programming languages are a threat to software security, and particularly insider attacks, as it is trivial for malicious components to subvert trusted ones.

The goal of our research is to develop aspect-oriented programming mechanisms that provide support for representation independence and information hiding, the fundamental mechanisms necessary for reliable and secure programming-in-the-large. In order to achieve this goal, we are developing a new functional, aspect-oriented programming language called AspectML. AspectML is a conservative extension of SML/NJ [1] with features that allow programmers to define point cuts and advice. As in other aspect-oriented programming languages, these tools can be used to separate out orthogonal concerns from the main-line computation and thereby untangle code. However, unlike other aspect-oriented languages, AspectML programmers have fine-grained control over access to their data. Access rights, including the ability to determine the existence of module components and to read or write data that flows between functions, are specified through a convenient syntax. AspectML propagates these access rights through its type system and prevents client code from violating the information hiding and abstraction policy established by the implementer. When the core language is combined with ML's rich module system, the programmer has tight control over which regions of code have access to which set of rights.

In the next section of this paper, we describe the main features of AspectML through a series of examples. First, we cover the basic mechanisms for defining AspectML's version of point cuts and advice. Many of these basic design points are inspired by earlier work by Walker, Zdancewic and Ligatti [9], although the specifics of our design are new. Next, we introduce mechanisms for specifying access rights and explain how to integrate access rights into the ML type system. In addition, we show how to use the ML module system to help control propagation of access rights. Finally, we summarize our design philosophy and discuss some of the trade-offs of our design with respect to AspectJ. In section 3, we specify the semantics of a simplified and idealized core language, including typing rules and execution behavior. We have proven our idealization is sound. Section 4 discusses our implementation. As of writing this paper, we have implemented a type checker for the language and are working on completing the back end. Section 4 also mentions future and related work.

2. ASPECTML

2.1 Preliminary Concepts

AspectML extends Standard ML with two central features, *first-class control-flow pairs* and *first-class advice*, that work together to allow programmers to untangle code.

First-class Control-flow Pairs. AspectML allows programmers to declare, store and manipulate data structures that represent pairs of program control-flow points. These control-flow pairs, or CFPs, are created whenever a programmer declares a function in AspectML. The first control-flow point in the CFP represents the instant in time just prior to execution of the function body. The second in the pair represents the instant in time just after execution of the function body but before the return of its result.

Specifically, when a programmer declares the function `fun f x = ...`, they implicitly declare a CFP that is bound to the variable `$f`. The variable `$f` may be used in the scope following the function declaration and follows all of the ordinary scoping rules for ML variables. For instance, after defining functions `fact` and `double`, we might collect the CFPs for these two functions together in a list called `pts`:

```
fun fact x = if x <= 1 then 1 else fact (x-1) * x

fun double x = x * 2

val pts = [$fact, $double]
```

The CFPs in the example above will later be used to specify *when* advice is triggered.

First-class Advice. AspectML allows programmers to define several different forms of advice, but the simplest has this shape: `advice` t *pat* `=` e. The expression e is the body of the advice. It is ordinary ML code that will execute when the advice is triggered. The first two parameters of the declaration, t and *pat*, collaborate to specify when the advice will be run. The time t is either `before` or `after`. It specifies which of the control-flow points in the CFP should be selected–the point before or after the function body is executed. The pattern *pat* specifies which CFPs will trigger the advice.

For example, suppose we wanted to add some debugging support to the program we began above. If we decided to print out some information both before and after the function `fact`, we would declare two pieces of advice:

```
val traceEntry =
  advice before <| pts as p(arg) |> @ _ = ...

val traceExit =
  advice after <| pts as p(res) |> @ _ = ...
```

The two patterns specify that advice will be triggered when execution reaches any of the CFPs designated by the list `pts` defined earlier (*i.e.* when control reaches the functions `fact` or `double`). When the advice is triggered, the particular CFP that triggered it (`$fact` or `$double`) is bound to the variable `p`, which may be used in the body of the advice. Moreover, in the `traceEntry` advice, the variable `arg` is bound to the argument of the function that is being advised and in the `traceExit` advice, the variable `res` is bound to the result of the function that is being advised.

As mentioned above, the body of advice is ordinary ML code. Hence, to complete our tracing advice we might write the following.

```
fun fact x = if x <= 1 then 1 else fact (x-1) * x

fun double x = x * 2

val pts = [$fact, $double]

val traceEntry =
  advice before <| pts as p(arg) |> @ _ =
    print ("enter:" ^ (label p) ^ ":"
           ^ (Int.toString arg))

val traceExit =
  advice after <| pts as p(res) |> @ _ =
    print ("leave:" ^ (label p) ^ ":"
           ^ (Int.toString res))

val runTrace =
  fn _ => traceEntry >> traceExit >> ()
```

Figure 1: A simple AspectML program

```
val traceEntry =
  advice before <| pts as p(arg) |> @ _ =
    print ("enter:" ^ (label p) ^ ":"
           ^ (Int.toString arg))

val traceExit =
  advice after <| pts as p(res) |> @ _ =
    print ("leave:" ^ (label p) ^ ":"
           ^ (Int.toString res))
```

The only element of these definitions we have not seen before is the function `label`, which is a built-in AspectML operator that extracts a string from a CFP. Currently, the string is the name of the function that corresponds to the CFP.

The above declarations introduce our first-class advice, but that advice has no effect until it is *installed*. The expression `e1 >> e2` installs advice `e1` after any other advice that has already been installed and then executes the expression `e2`. The expression `e1 << e2` installs `e1` before other advice. For instance, if we write `e1 >> e2 >> ()` then whenever both `e1` and `e2` are triggered by the same control-flow point, the effects of `e2` will occur after the effects of `e1`. On the other hand, if we write `e1 >> e2 << ()` then the effects of `e1` will occur after the effects of `e2`.

In the current example, both pieces of advice operate over disjoint sets of control-flow points. Therefore it does not matter whether we use `>>` or `<<`. Figure 1 presents our entire AspectML program in one piece, including the function `runTrace` that can be called to install the tracing advice.

2.2 Advanced Advice

In addition to the simple before and after advice described in the previous section, AspectML provides facilities that allow programmers to write *context-sensitive* advice that can takes the calling context of a function into consideration, a mechanism that allows programmers to create advice with similar functionality to AspectJ's *around* advice, and *volatile* advice that can modify function arguments and results.

Context-sensitive Advice. As we have already seen, the body of an advice expression may reference the CFP associated with the function that triggered it. This feature was used above to print the label of the function that was called. In addition, advice may also depend upon the CFPs associated with the functions on the current control stack. Following the tradition of functional programming, CFP stacks are first-class, immutable data structures like any other. Programmers may analyze them and extract their components through pattern matching.

```
fun fact x = ...

val unwindPts = ...

fun unwind stack =
  scase stack of
    NilStack =>
      print "done"
  | <| unwindPts as f(_) |> @ stack' =>
      print ((label f) ^ ":");
      unwind stack'
  | _ @ stack' =>
      unwind stack'

val traceStack =
  fn pts condition =>
    advice before <| pts as f(x) |> @ stack =
      if condition (f,x) then
        (print ((label f) ^ ":");
         unwind stack)
      else
        ()

val _ =
  (traceStack [$fact] (fn (_,x) => x < 0)) >> ()
```

Figure 2: Context-sensitive advice

Figure 2 presents an example of debugging advice to illustrate the concept. In this example, the function `unwind` prints the labels on a CFP stack one by one. The main element of interest is the *scase*, or stack-case, statement, which uses three patterns to match a CFP stack. The first pattern matches the empty stack and the next two match non-empty stacks involving a "head" (at the top of the stack) and, after the "`@`", a "tail" that matches the rest of the stack. The head of the second pattern is a CFP pattern, similar to the ones we saw in the previous subsection.

Continuing with the example in Figure 2, the function `traceStack` takes two arguments, a list of CFPs and a predicate, and generates advice to print the stack of CFPs when the predicate holds. The initial pattern in this advice declaration binds the variable `stack` to the CFP stack at the time the advice is triggered. The last declaration in the figure applies the function to the specified CFP list and predicate and finally installs the generated advice.

Around Advice. AspectJ allows users to declare advice *around* a function `f`. The body of this sort of advice has the effect of entirely replacing the body of the function `f`. AspectML includes an alternative mechanism, the command `return e1 to e2`, that supports around advice.

The returnto expression evaluates its arguments `e1` and

e2 to obtain values v1 and v2. It then crawls down the call stack to the nearest enclosing activation record for the function associated with the CFP v2. At this point, the value v1 is substituted for what would have been the result of the function and execution continues. To simulate around advice using this mechanism, the programmer writes before advice that terminates with the use of returnto:

```
advice before <| pts as p(x) |> @ _ =
  ... return e to p ...
```

Volatile Advice. *Stable* advice is advice executed exclusively for its effect; it does not modify function arguments or results when it is invoked. Our previous tracing and debugging examples have been stable advice.

Volatile advice, on the other hand, does modify function arguments or results. Using returnto results in one form of volatile advice since the value returned replaces a function result. Programmers can also add the keyword `volatile` to before and after advice to allow them to modify arguments and results respectively. For instance, to modify the argument to the `fact` function, one may write the following.

```
advice before volatile <| [$fact] as p(x) |> @ _ =
  if x < 0 then 0 else x
```

The default *volatility* for advice is stable. Optionally, the programmer can add the keyword `stable` to before or after advice to emphasize that the advice will not change the argument (or result).[1]

2.3 Typing and Access Control

The various sorts of advice provided by AspectML allow a client component written by a programmer A to determine all kinds of information about an implementation component written by another programmer B: Stable advice written by A can determine the existence of functions written by B; stable advice written by A can read data that flows between functions written by B; and volatile advice written by A can modify the data that flows between functions written by B. In the first two cases, A's code will depend directly upon code written by B, and if B wishes to change her code, the changes may break A's code. In the last case, the code written by A can disrupt local invariants that are important to the proper functioning of B's code. The bottom line is that without protection mechanisms that can create a boundary between A and B, AspectML code can become *conceptually entangled* even if it is not *physically entangled*, and conceptually entangled code is at least as difficult to debug as physically entangled code.

In order to protect their code from outside interference, AspectML programmers decorate their function declarations with access controls. For instance, if we wish to restrict access to our familiar `fact` and `double` functions, we might declare them as follows.

```
fun[r,n] fact x =
  if x <= 0 then 1 else fact (x-1) * x

fun[r,n] double x = x * 2
```

[1]If the argument or result of a function is ordinarily a mutable object (a reference or array), the advice will be able to assign to the object, changing the value it points to, even in stable advice.

The access control specification `[r,n]` decorating both function declarations specifies that the argument of the function is *readable* but not *writeable* (designated by `r`) and the result of the function is *neither* readable nor writeable (designated by `n`). The two other access controls, not used in this example, are *writeable* (`w`) and *read-write* (`rw`).

A readable argument or result can be bound to a pattern and used within stable advice. A writable argument or result can be computed by invoking volatile advice or (in the case of a result) a returnto expression. For instance, reconsider our two declarations of advice from above:

```
val pts = [$fact, $double]

val traceEntry =
  advice before <| pts as p(arg) |> @ _ =
    print ("enter:" ^ (label p) ^ ":"
           ^ (Int.toString arg))

val traceExit =
  advice after <| pts as p(res) |> @ _ =
    print ("leave:" ^ (label p) ^ ":"
           ^ (Int.toString res))
```

With the new access controls, `traceEntry` is well typed, but `traceExit` is not. In the latter case, the advice accesses the result of the function, which is not allowed. To obtain a type-correct program we must modify `traceExit` so that it uses the wildcard pattern:

```
val traceExit =
  advice after <| pts as p(_) |> @ _ =
    print ("leave:" ^ (label p))
```

Notice that the declarations above implicitly permit programmers to use the `label` operation on the respective CFPs. To deny client code any access whatsoever to a CFP, one simply uses an ordinary ML function declaration: `fun f x = ...`. Our default access control choice is one that favors safety and is faithful to ML semantics.

Types. To integrate access control checking smoothly with the rest of ML typechecking, we include access control specifications in the types of CFPs. In general, these types resemble record types and have the form

$$\{\{\texttt{arg:}a_1\ \tau_1\texttt{,res:}a_2\ \tau_2\texttt{,lab:}a_3\}\}$$

where a_1, a_2, and a_3 are the access control specifications, and τ_1 and τ_2 are the types of the arguments and results of the respective functions. When no access is allowed for one of the components, the component is dropped from the collection.

As an example, consider the declaration of the factorial function (`fun[r,n] fact x = ...`) given above. The type of the related CFP `$fact` is {{`arg:r int,lab:r`}}. This type allows the argument of `fact`, which is an integer, to be read within stable before advice. It also allows the `label` operation to be applied to `$fact`. If we extended the privileges for `$fact` by declaring it with privileges `[r,rw]` then `$fact` would have the type

$$\{\{\texttt{arg:r int,res:rw int,lab:r}\}\}$$

and we would be able to write volatile after advice for `$fact` and use the operation `return e to $fact` provided that `e` has integer type.

Subtyping. The types of CFPs are ordered by a natural subtyping relation: a CFP with more access rights is a subtype of a CFP with fewer access rights. For instance, the type that *does* allow access to the result of `fact` is a subtype of the type that *does not* allow access to the result of `fact`:

$$\{\{\texttt{arg:r int,res:rw int,lab:r}\}\} \leq \{\{\texttt{arg:r int,lab:r}\}\}$$

This subtyping relation is extremely useful to programmers who wish to provide different sets of rights to different client components. For instance, a programmer may allow his or her own local code to have quite liberal access rights and then to reduce access rights to ensure that external components do not depend on or interfere with local invariants.

AspectML programmers manage subtyping through explicit coercions. For instance, if `e` is an expression with a CFP type, then the expression `e <-` τ is an upcast that can reduce the access rights associated with `e`. We do not support coercions for higher types (pairs, functions, lists) as they are not particularly useful in our context.[2]

Although explicit coercions have more syntactic overhead than implicit subtyping, we have chosen the former as it is simpler to integrate with (Standard) ML type inference. However, we could have used row polymorphism and its associated type inference techniques.[3]

2.4 Aspects and Modules

AspectML makes no changes to ML's rich module system. However, the core language was carefully designed with this module system in mind. In particular, AspectML programmers are expected to use the module system in conjunction with access controls and subtyping to raise abstraction boundaries within their programs where appropriate.

The code below presents an open implementation of an arithmetic structure containing our familiar `fact` and `double` functions. The signature `ARITHOPEN` is sufficiently liberal that clients may write any sort of advice, stable or volatile, to manipulate these operations.

```
signature ARITHOPEN =
  sig
    val fact    : int -> int
    val double  : int -> int
    val $fact   : {{arg:rw int,res:rw int,lab:r}}
    val $double : {{arg:rw int,res:rw int,lab:r}}
  end
structure arithopen :> ARITHOPEN =
  struct
    fun[rw,rw] fact = ...
    fun[rw,rw] double = ...
  end
```

The ML module system makes it easy to create different interfaces to the arithmetic package that vary the access rights provided to different components of the system. For instance, in the code below, we define two additional structures `arith`, which provides no CFPs that can be used to create advice, and `arithProf`, which summarizes the CFPs important for profiling and limits the rights associated with these CFPs to label-only status. Now, debugging infrastructure, which requires liberal access to the underlying representations in order to be effective, can be defined over the `arithopen` structure; profiling infrastructure, which only needs access to the names of functions to construct and print profiling data, can be defined over `arithProf`; and the bulk of the code can be defined over `arith` and its purely functional interface.

```
signature ARITH =
  sig
    val fact   : int -> int
    val double : int -> int
  end
structure arith :> ARITH = arithopen

signature PROF =
  sig
    val profilePts : {{lab:r}} list
  end
structure arithProf :> PROF =
  struct
    type t = {{lab:r}}
    val profilePts =
      [arithopen.$fact<-t, arithopen.$double<-t]
  end
```

2.5 Summary of Design Philosophy

One of AspectML's design goals is to provide programmers with as much choice as possible concerning the degree of abstraction and information hiding in their programs. Just as ordinary ML allows programmers to declare types transparent, translucent or abstract, AspectML allows programmers to create modules in which all functions are completely open to modification, some functions are partially open, or no functions are open at all. When programming a more open style, one can simulate the features of more traditional aspect-oriented programming languages. Such a style can be useful in smaller systems or in localized regions of a larger system. At the same time, traditional ML programmers can transition to AspectML without fear that any of the properties of their programs will be disrupted. In other words, unlike any other aspect-oriented design proposal that we are aware of, our proposal is a *conservative* extension of a language with strong modularity properties.

While AspectML admits many choices, it *encourages* safer, more modular aspect-oriented programming through careful construction and a choice of the defaults. For instance, by default, function declarations are closed, and when they are left open, a concise local annotation reminds programmers that they may be modified. Similarly, when CFPs are left out of an interface, access is denied. In order to provide external access, a programmer must explicitly include a CFP in an interface, and as above, the interface documents the possibility that external components may influence internal semantics.

The fundamental trade-off between a language like AspectJ and a language like AspectML is whether or not the implementer of a module is allowed to set an access control

[2] In the uncommon case a programmer needs coercions for higher types, they may build them explicitly For instance, a programmer may coerce a function `f` through the standard eta-expansion trick: `fn x:`τ_1`.(f (x <-` τ_2`)) <-` τ_3. This coercion will have some non-zero run-time overhead.

[3] As you will see, our formal CFP types are very similar to records, with each field a single access control right. Subtyping for CFP types is width subtyping, and therefore row polymorphism applies directly.

$$
\begin{array}{llll}
\text{types} & \tau & ::= & \texttt{string} \mid \texttt{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \texttt{list} \\
& & \mid & \texttt{stack} \mid \texttt{advice} \mid \{a_i^{i\in 1..n}\} \\
\\
\text{ctxts} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\
\\
\text{ac specs} & a & ::= & \texttt{lab}_r \mid \texttt{arg}_r : \tau \mid \texttt{arg}_w : \tau \\
& & \mid & \texttt{res}_r : \tau \mid \texttt{res}_w : \tau \\
\\
\text{terms} & e & ::= & x \mid s \mid () \\
& & \mid & \texttt{let}\, ds\, \texttt{in}\, e \mid \lambda x{:}\tau.e \mid e_1\, e_2 \\
& & \mid & nil \mid \texttt{cons}(e_1, e_2) \\
& & \mid & \texttt{lcase}\, e (nil \Rightarrow e_1 | \texttt{cons}(x, y) \Rightarrow e_2) \\
& & \mid & \texttt{scase}\, e (nil_s \Rightarrow e_1 | pt\ \texttt{@}\ x \Rightarrow e_2 |\_\ \texttt{@}\ y \Rightarrow e_3) \\
& & \mid & \texttt{return}\, e_1\, \texttt{to}\, e_2 \mid \texttt{label}\, e \\
& & \mid & \texttt{advice}\, t\, vol (pt \,@\, y) = e \\
& & \mid & e_1 \texttt{ >> } e_2 \mid e_1 \texttt{ << } e_2 \\
\\
\text{decls} & ds & ::= & \cdot \\
& & \mid & (\texttt{val}\, x = e)\, ds \\
& & \mid & (\texttt{fun}\, f (p{:}\{a_i^{i\in 1..n}\})(x{:}\tau_1){:}\tau_2 = e)\, ds \\
\\
\text{times} & t & ::= & \texttt{before} \mid \texttt{after} \\
\\
\text{vol's} & vol & ::= & \texttt{stable} \mid \texttt{volatile} \\
\\
\text{cfp pats} & pt & ::= & e\, \texttt{as}\, p(x)
\end{array}
$$

Figure 3: Formal Syntax

policy for the module. AspectML allows implementers to set policy; AspectJ does not, allowing unrestricted access using the *privileged* keyword.. The primary disadvantage of AspectML's design choice is that programmers might disallow access and later wish they had not, forcing them to restructure their interfaces. The primary disadvantage of AspectJ's design choice is that to understand the semantics of *any* method, programmers must *always* examine *all* of their code. We believe the latter disadvantage outweighs the former, particularly since in AspectML, there is a simple way to correct an overly restrictive design, but in AspectJ, there are no mechanisms to build and check abstractions.

AspectML also has more notational overhead than AspectJ as programs must specify CFPs and their types in signatures. This overhead may be viewed as an inconvenience to programmers. However, we believe that it provides an important form of machine-checkable documentation that specifies access points and privileges and that it will help make aspect-oriented programs more reliable.

3. CORE LANGUAGE SEMANTICS

3.1 Syntax

The formal syntax is presented in figure 3. For the most part, it resembles the examples given in previous sections. One minor difference appears in the syntax of access control specifications (a). Rather than allowing composite access controls such as the `rw` designator, every privilege is split out into its own individual specifications. These specifications are then collected together in a type for CFPs ($\{a_i^{i\in 1..n}\}$). Presence of a particular specification within a CFP type grants access; absence of a specification prevents access. Types for CFPs that differ only in the order of their access control specifications are considered to be identical. Otherwise, type equality is purely syntactic. Type contexts (Γ) are finite partial maps used for type checking.

The syntax of terms is largely self-explanatory. We use metavariables x, y, z, f and p to range over variables. We use f to emphasize that the variable has function type and p to emphasize that the variable has CFP type. The metavariable s ranges over strings. There are two forms of function in the calculus. The first ($\lambda x : \tau.e$) is an anonymous function that has no associated CFP. The second ($\texttt{fun}\, f (p{:}\{a_i^{i\in 1..n}\})(x{:}\tau_1){:}\tau_2 = e$) is a combined declaration for both a recursive function f and its associated CFP p with access rights $\{a_i^{i\in 1..n}\}$. In the previous section's examples, the CFP p was always implicitly defined as \$f where f was the name of the function.

The `lcase` operation destructs a list whereas the `scase` operation destructs a stack. We will use the abbreviation $e_1; e_2$ for $\texttt{let}(\texttt{val}\, x = e_1)\texttt{in}\, e_2$ when x is not free in e_2. We also write $[e_1, \ldots, e_n]$ for $\texttt{cons}(e_1, \cdots \texttt{cons}(e_n, nil))$. As usual, we treat objects that only differ in the names of their bound variables as identical.

3.2 Static Semantics

The rules for typechecking terms are presented in Figure 4. Most of the rules should be familiar: unit, strings, (anonymous) functions and lists all have standard typing rules. In the rules for checking functions and elsewhere, we implicitly alpha-convert bound variables where necessary to be sure they do not overlap with variables in the domain of the typechecking context.

The rules for `scase` must determine how the information in a CFP pattern ($e\, \texttt{as}\, p(x)$) should be extracted and used. The first step in the process is to find the type of e, which must have the form $\{a_i^{i\in 1..n}\}\ \texttt{list}$. Then, a function $\texttt{ctxt}(t, \{a_i^{i\in 1..n}\}, p, x) \Rightarrow \Gamma$ constructs Γ based upon a time t and access controls $\{a_i^{i\in 1..n}\}$ (as well as p and x). The idea is that Γ will assign p the given access rights and it will only contain x if permitted by those access rights. The `ctxt` function for constructing Γ given arguments t, $\{a_i^{i\in 1..n}\}$, p and x follows.

- $p{:}\{a_i^{i\in 1..n}\} \in \Gamma$
- If $t = \texttt{before}$ and $\texttt{arg}_r{:}\tau \in \{a_i^{i\in 1..n}\}$ then $x{:}\tau \in \Gamma$
- If $t = \texttt{after}$ and $\texttt{res}_r{:}\tau \in \{a_i^{i\in 1..n}\}$ then $x{:}\tau \in \Gamma$
- No other assumptions appear in Γ

When `ctxt` is used in checking stack pattern matching, its temporal argument is `before` since the objects on the stack are function arguments not results.

The two operations, `returnto` and `label`, have simple typing rules. Importantly, each rule checks to make sure that the appropriate access rights are available.

The rules for typechecking advice use the `ctxt` function, just as in the rules for typechecking the stack-case operation.

Checking advice also requires a second function, $\texttt{body}(t, vol, \{a_i^{i\in 1..n}\}) \Rightarrow \tau$, to determine the expected type of the body of advice. The partial function `body` is defined as follows:

- If $vol = \texttt{stable}$ then $\tau = \texttt{unit}$

- If vol = `volatile` and t = `before` and $\texttt{arg}_w : \tau' \in \{a_i^{i\in 1..n}\}$ then $\tau' = \tau$
- If vol = `volatile` and t = `after` and $\texttt{res}_w : \tau' \in \{a_i^{i\in 1..n}\}$ then $\tau' = \tau$

If the advice body type checks appropriately, the advice is given type `advice`. The elimination forms for advice, `<<` and `>>` , expect objects with type `advice` as the primary arguments.

The last rule in the first group in Figure 4 is the standard subsumption rule. We use implicit subtyping in our formal work rather than the explicit coercions of the implementation. There are only two rules for subtyping in our calculus:

$$\frac{}{\{a_i^{i\in 1..n+k}\} \le \{a_i^{i\in 1..n}\}} \qquad \frac{}{\tau \le \tau}$$

The first rule allows access control rights to be dropped, thereby tightening control over access to a CFP. The second rule simply states that equal types are subtypes of one another. Transitivity of subtyping can easily be derived from these two rules.

The bottom section of Figure 4 presents three rules for checking a sequence of declarations followed by an expression. The first rule checks the empty sequence. The second rule checks a value declaration followed by a sequence. The last rule checks the combined function and CFP declaration. The subtyping judgment in that last rule checks that the access control specification is compatible with the type of the function. For instance, it prevents a programmer from granting the $\{\texttt{arg}_r : \texttt{bool}\}$ access right when the argument type of the function is `int`.

3.3 Operational Semantics

This section describes the abstract machine that executes AspectML programs. Figure 5 introduces the additional syntactic elements we need. At the bottom of the figure is the schema for machine states M. They are constructed from a code store C, advice store A and term to be evaluated e. The code store is a finite partial map from locations l to pairs of code and access control specification for that code. The advice store is a sequence of advice declarations (order is important).

In order to state the operational semantics, we extend the language of terms to include references to functions ($\texttt{fun}(l)$) and access controls ($\texttt{cfp}(l)$) found in the code store. The token nil_s represents the empty stack and $l[v_1] :: v_2$ represents a stack with an activation record for function l and argument v_1 on top and v_2 following.[4] The meta variables v, v_1, v_2, etc. range over values.

The last new term is $\texttt{store}\, l[x] = e_1\, \texttt{in}\, e_2$, a command to store a value (the result of evaluating e_1) on the stack, in the activation record for the function associated with l. The code e_2 can read the value off the stack by referring to x. To indicate the constraint that the bound variable in the store expression does not appear free in e_2, we write $\texttt{store}\, l[\_] = e_1\, \texttt{in}\, e_2$

In order to determine the complete, current stack given an expression e that contains many `store` instructions, we break that expression into a redex and an evaluation context E and then apply the following function to E (where `@` is intended to be a function that concatenates two stacks):

[4]Stacks are fairly list-like. However, the list values are nil and $\texttt{cons}(v_1, v_2)$.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash s : \texttt{string}} \qquad \frac{}{\Gamma \vdash () : \texttt{unit}}$$

$$\frac{\Gamma \vdash ds; e : \tau}{\Gamma \vdash \texttt{let}\, ds\, \texttt{in}\, e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}$$

$$\frac{}{\Gamma \vdash nil : \tau\ \texttt{list}}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau\ \texttt{list}}{\Gamma \vdash \texttt{cons}(e_1, e_2) : \tau\ \texttt{list}}$$

$$\frac{\Gamma \vdash e : \tau\ \texttt{list} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau, y{:}\tau\ \texttt{list} \vdash e_2 : \tau_1}{\Gamma \vdash \texttt{lcase}\, e (nil \Rightarrow e_1 | \texttt{cons}(x, y) \Rightarrow e_2) : \tau_1}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{stack} \qquad \Gamma \vdash e_1 : \tau_1 \qquad (x \notin \texttt{Dom}(\Gamma)) \\ \Gamma \vdash e' : \{a_i^{i\in 1..n}\}\ \texttt{list} \qquad \texttt{ctxt}(\texttt{before}, \{a_i^{i\in 1..n}\}, p, x) \Rightarrow \Gamma' \\ \Gamma, \Gamma', y{:}\texttt{stack} \vdash e_2 : \tau_1 \qquad \Gamma, z{:}\texttt{stack} \vdash e_3 : \tau_1\end{array}}{\Gamma \vdash \texttt{scase}\, e (nil_s \Rightarrow e_1 | e'\, \texttt{as}\, p(x)\, @\, y \Rightarrow e_2 |\_\, @\, z \Rightarrow e_3) : \tau_1}$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \{\texttt{res}_w{:}\tau\}}{\Gamma \vdash \texttt{return}\, e_1\, \texttt{to}\, e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \{\texttt{lab}_r\}}{\Gamma \vdash \texttt{label}\, e : \texttt{string}}$$

$$\frac{\begin{array}{c}(x \notin \texttt{Dom}(\Gamma)) \\ \Gamma \vdash e_1 : \{a_i^{i\in 1..n}\}\ \texttt{list} \quad \texttt{ctxt}(t, \{a_i^{i\in 1..n}\}, p, x) \Rightarrow \Gamma' \\ \texttt{body}(t, vol, \{a_i^{i\in 1..n}\}) \Rightarrow \tau_2 \quad \Gamma, \Gamma', y{:}\texttt{stack} \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash \texttt{advice}\, t\, vol (e_1\, \texttt{as}\, p(x)\, @\, y) = e_2 : \texttt{advice}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{advice} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ \texttt{>>}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{advice} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ \texttt{<<}\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \le \tau_2}{\Gamma \vdash e : \tau_2}$$

$\boxed{\Gamma \vdash ds; e : \tau}$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash .; e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x{:}\tau_1 \vdash ds; e_2 : \tau_2}{\Gamma \vdash (\texttt{val}\, x = e_1)\, ds; e_2 : \tau_2}$$

$$\frac{\begin{array}{c}\Gamma, f{:}\tau_1 \to \tau_2, x{:}\tau_1 \vdash e_1 : \tau_2 \\ \Gamma, f{:}\tau_1 \to \tau_2, p{:}\{a_i^{i\in 1..n}\} \vdash ds; e : \tau \\ \{label_r, \texttt{arg}_r{:}\tau_1, \texttt{arg}_w{:}\tau_1, \texttt{res}_r{:}\tau_2, \texttt{res}_w{:}\tau_2\} \le \{a_i^{i\in 1..n}\}\end{array}}{\Gamma \vdash (\texttt{fun}\, f (p{:}\{a_i^{i\in 1..n}\})(x{:}\tau_1){:}\tau_2 = e_1)\, ds; e : \tau}$$

Figure 4: Term Typing

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \texttt{fun}(l) \mid \texttt{cfp}(l) \mid nil_s \mid l[v_1] :: v_2 \\
 & \mid & \texttt{store}\, l[x] = e_1 \,\texttt{in}\, e_2 \\
 & \mid & \texttt{store}\, l[\_] = e_1 \,\texttt{in}\, e_2 \\
v & ::= & s \mid () \mid \texttt{fun}(l) \mid \texttt{cfp}(l) \mid \lambda x{:}\tau.e \\
 & \mid & nil \mid \texttt{cons}(v_1, v_2) \mid nil_s \mid l[v_1] :: v_2 \\
 & \mid & \texttt{advice}\, t\, vol\, (pt_v \mathbin{@} y) = e \\
pt_v & ::= & v \,\texttt{as}\, p(x) \\
E & ::= & [] \mid E\ e \mid v\ E \\
 & \mid & \texttt{let}\,(\texttt{val}\, x = E)\, ds \,\texttt{in}\, e \\
 & \mid & nil \mid \texttt{cons}(E, e) \mid \texttt{cons}(v, E) \\
 & \mid & \texttt{lcase}\, E\, (nil \Rightarrow e_1 | \texttt{cons}(x, y) \Rightarrow e_2) \\
 & \mid & \texttt{return}\, E\, \texttt{to}\, e \mid \texttt{return}\, v\, \texttt{to}\, E \mid \texttt{label}\, E \\
 & \mid & \texttt{scase}\, E\, (nil_s \Rightarrow e_1 | e\, \texttt{as}\, p(x)\, \texttt{@}\, y \Rightarrow e_2 | \_\, \texttt{@}\, z \Rightarrow e_3) \\
 & \mid & \texttt{scase}\, v\, (nil_s \Rightarrow e_1 | E\, \texttt{as}\, p(x)\, \texttt{@}\, y \Rightarrow e_2 | \_\, \texttt{@}\, z \Rightarrow e_3) \\
 & \mid & \texttt{advice}\, t\, vol\, (E\, \texttt{as}\, p(x) \mathbin{@} y) = e \\
 & \mid & E \gg e \mid E \ll e \\
 & \mid & \texttt{store}\, l[x] = E \,\texttt{in}\, e \mid \texttt{store}\, l[\_] = v \,\texttt{in}\, E \\
C & ::= & \cdot \mid C, l \rightarrow ((\texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2 = e), \{a_i^{i \in 1..n}\}) \\
A & ::= & \cdot \mid A, \texttt{advice}\, t\, vol\, (E\, \texttt{as}\, p(x) \mathbin{@} y) = e \\
M & ::= & < C, A, e >
\end{array}
$$

Figure 5: Run-time data

$$
\begin{array}{lcl}
\texttt{stack}([]) & = & nil_s \\
\texttt{stack}(\texttt{store}\, l[x] = v \,\texttt{in}\, E) & = & \texttt{stack}(E)\, \texttt{@}\, (l[v] :: nil_s) \\
\texttt{stack}(E) & = & \text{recursively apply } \texttt{stack} \text{ to nested context within } E
\end{array}
$$

Dynamic Access Control Checks. Our operational semantics will contain explicit (dynamic) access control checks. In well-typed programs, these access control checks never fail, and therefore, they need not be implemented. The access control check for converting a label to a string simply verifies that the $\texttt{lab}_r$ permission has been granted for that label. The access control checks for reading and writing arguments and results of functions are slightly more involved: $\texttt{readcheck}(t, \{a_i^{i \in 1..n}\}, x, e)$ is valid if

- $t = \texttt{before}$ and $x \in FV(e)$ implies $\texttt{arg}_r : \tau \in \{a_i^{i \in 1..n}\}$
- $t = \texttt{after}$ and $x \in FV(e)$ implies $\texttt{res}_r : \tau \in \{a_i^{i \in 1..n}\}$

$\texttt{writecheck}(t, \{a_i^{i \in 1..n}\})$ is valid if

- $t = \texttt{before}$ implies $\texttt{arg}_w : \tau \in \{a_i^{i \in 1..n}\}$
- $t = \texttt{after}$ implies $\texttt{res}_w : \tau \in \{a_i^{i \in 1..n}\}$

The `readcheck` predicate determines whether a value bound to a variable x is read in the expression e by examining the free variables of e. This check is sufficient, but even if e does contain x, e will not necessarily read it. We have specified the access control check in this manner as it appears to be the simplest and most elegant approach.[5]

[5]Refining the calculus with explicit substitutions might allow us specify necessary and sufficient conditions on read access, but this minor benefit does not justify the additional machinery we would need.

$\boxed{< C, A, e > \longmapsto_\beta < C', A', e' >}$

$$< C, A, \texttt{let} \cdot \texttt{in}\, e > \longmapsto_\beta < C, A, e >$$

$$< C, A, \texttt{let}\,(\texttt{val}\, x = v)\, ds \,\texttt{in}\, e > \longmapsto_\beta < C, A, (\texttt{let}\, ds \,\texttt{in}\, e)[v/x] >$$

$$< C, A, \texttt{let}\,(\texttt{fun}\, f\, (p{:}\{a_i^{i \in 1..n}\})(x{:}\tau_1){:}\tau_2 = e_1)\, ds \,\texttt{in}\, e_2 > \longmapsto_\beta < C, l \longmapsto ((\texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2 = e_1), \{a_i^{i \in 1..n}\}), A, (\texttt{let}\, ds \,\texttt{in}\, e_2)[\texttt{fun}(l)/f][\texttt{cfp}(l)/p]) >$$

$$< C, A, (\lambda x{:}\tau.e)\ v > \longmapsto_\beta < C, A, e[v/x] >$$

$$\frac{C(l) = (..., \{\texttt{lab}_r, \ldots\}) \quad \texttt{labelToString}(l) = s}{< C, A, \texttt{label}\, \texttt{cfp}(l) > \longmapsto_\beta < C, A, s >}$$

$$< C, A, \texttt{store}\, l[x] = v_1 \,\texttt{in}\, e_2 > \longmapsto_\beta < C, A, \texttt{store}\, l[\_] = v_1 \,\texttt{in}\, e_2[v_1/x] >$$

$$< C, A, \texttt{store}\, l[\_] = v_1 \,\texttt{in}\, v_2 > \longmapsto_\beta < C, A, v_2 >$$

$$\frac{l \notin \texttt{stack}(E) \quad C(l) = (..., \{a_i^{i \in 1..n}\}) \quad \texttt{writecheck}(\texttt{after}, \{a_i^{i \in 1..n}\})}{< C, A, \texttt{store}\, l[\_] = v_1 \,\texttt{in}\, E[\texttt{return}\, v_2 \,\texttt{to}\, \texttt{cfp}(l)] > \longmapsto_\beta < C, A, v_2 >}$$

$$< C, A, \texttt{lcase}\, nil\, (nil \Rightarrow e_1 | \texttt{cons}(x, y) \Rightarrow e_2) > \longmapsto_\beta < C, A, e_1 >$$

$$< C, A, \texttt{lcase}\, \texttt{cons}(v_1, v_2)\, (nil \Rightarrow e_1 | \texttt{cons}(x, y) \Rightarrow e_2) > \longmapsto_\beta < C, A, e_2[v_1/x][v_2/y] >$$

$$< C, A, \texttt{scase}\, nil_s\, (nil_s \Rightarrow e_1 | pt_v\, \texttt{@}\, y \Rightarrow e_2 | \_\, \texttt{@}\, z \Rightarrow e_3) > \longmapsto_\beta < C, A, e_1 >$$

$$\frac{v_1 \models pt_v \Rightarrow \theta \qquad C(l) = (..., \{a_i^{i \in 1..n}\}) \qquad pt_v = v\, \texttt{as}\, p(x) \qquad \texttt{readcheck}(\texttt{before}, \{a_i^{i \in 1..n}\}, x, e)}{< C, A, \texttt{scase}\, l[v_1] :: v_2\, (nil_s \Rightarrow e_1 | pt_v\, \texttt{@}\, y \Rightarrow e_2 | \_\, \texttt{@}\, z \Rightarrow e_3) > \longmapsto_\beta < C, A, \theta(e_2)[v_2/y] >}$$

$$\frac{v_1 \not\models pt_v}{< C, A, \texttt{scase}\, v_1 :: v_2\, (nil_s \Rightarrow e_1 | pt_v\, \texttt{@}\, y \Rightarrow e_2 | \_\, \texttt{@}\, z \Rightarrow e_3) > \longmapsto_\beta < C, A, e_3[v_2/z] >}$$

$$< C, A, (\texttt{advice}\, t\, vol\, (pt_v \mathbin{@} y) = e_1 \gg e_2) > \longmapsto_\beta < C, (A, \texttt{advice}\, t\, vol\, (pt_v \mathbin{@} y) = e_1), e_2 >$$

$$< C, A, (\texttt{advice}\, t\, vol\, (pt_v \mathbin{@} y) = e_1 \ll e_2) > \longmapsto_\beta < C, (\texttt{advice}\, t\, vol\, (pt_v \mathbin{@} y) = e_1, A), e_2 >$$

$\boxed{< C, A, e > \longmapsto < C', A', e' >}$

$$\frac{< C, A, e > \longmapsto_\beta < C', A', e' >}{< C, A, E[e] > \longmapsto < C', A', E[e'] >}$$

$$\frac{l \notin \texttt{stack}(E)}{< C, A, E[\texttt{return}\, v_2 \,\texttt{to}\, \texttt{cfp}(l)] > \longmapsto < C, A, \texttt{return}\, v_2 \,\texttt{to}\, \texttt{cfp}(l) >}$$

$$\frac{\begin{array}{c} C(l) = ((\texttt{fun}\, f\, (x{:}\tau_1){:}\tau_2 = e), \{a_i^{i \in 1..n}\}) \\ \texttt{compose}(\texttt{before}, A, l{:}\tau_1, \texttt{stack}(E), \{a_i^{i \in 1..n}\}) \Rightarrow \texttt{bef} \\ \texttt{compose}(\texttt{after}, A, l{:}\tau_2, \texttt{stack}(E), \{a_i^{i \in 1..n}\}) \Rightarrow \texttt{aft} \\ e''' = \left(\begin{array}{l} \texttt{store}\, l[x] = \texttt{bef}\ v \,\texttt{in} \\ \texttt{let}\,(\texttt{val}\, y = e[\texttt{fun}(l)/f]) \,\texttt{in} \\ \texttt{aft}\ y \end{array} \right) \end{array}}{< C, A, E[\texttt{fun}(l)\ v] > \longmapsto < C, A, E[e'''] >}$$

Figure 6: Operational semantics

Evaluation Rules. Figure 6 specifies program evaluation using two judgments. The first judgment ($\longmapsto_\beta$) specifies the operation of basic commands that may appear within some evaluation context. The second judgment ($\longmapsto$) specifies top-level evaluation.

The first three rules for the $\longmapsto_\beta$ relation specify execution of `let` declarations. The most interesting is the rule for processing function declarations. It adds the function together with its access control specification to the code store and then substitutes `fun(`l`)` for the function variable and `cfp(`l`)` for the CFP variable in the rest of the declarations.

The next interesting rule is the rule for evaluating the `label` operation. It depends upon an unspecified operator (`labelToString`) that generates a string from a label l (the implementation generates a string from the source text). It also checks the access controls associated with the CFP to ensure that access has been granted.

The `store` and `returnto` instructions are implemented by three rules. First, when evaluation of the primary argument of `store` results in a value v_1, this value is substituted for x throughout e_2. Second, since `store` $l[\_] = v_1$ `in` E is an evaluation context, evaluation can proceed underneath a `store` instruction.[6] When evaluation of the body of the `store` instruction produces a value v_2, the function that produced the `store` expression is deemed to "return" and the `store` expression is eliminated, leaving only v_2. When a `returnto` expression appears in the context of a `store` expression, the value returned replaces the entire `store` expression.

The rules for `lcase` are ordinary but the rules for `scase` involve some additional definitions. The second rule for `scase` depends upon a partial function, written $l[v] \models pt_v \Rightarrow \theta$ for matching a CFP pattern and generating a substitution θ. The function is defined as follows.

$$\frac{l \in l_1, \ldots, l_n}{l[v] \models [\texttt{cfp}(l_1), \ldots, \texttt{cfp}(l_n)]\ \texttt{as}\ p(x) \Rightarrow [\texttt{cfp}(l)/p][v/x]}$$

We write $l[v] \not\models pt_v$ if $l[v]$ does not match the CFP pattern. This second `scase` rule also checks for violation of the access control policy for l using the `readcheck` predicate.

The rules for the $\longmapsto$ relation include a rule that allows a basic reduction rule to be applied in any evaluation context and a rule to terminate execution abnormally with a `return` v_1 `to cfp(`l`)` instruction. The last rule explains how to evaluate a recursive function application. Intuitively, recursive function application is processed in the following steps:

- Look up the code for the function in the code store C.
- Determine the code to run before the function due to before advice (`bef`).
- Determine the code to run after the function due to after advice (`aft`).
- Use the `store` instruction to store the function argument on the stack and then put the three things above together to form the code that will be run as the body of the recursive function.

Determining the code to run before and after a function call due to advice is done with the help of the `compose` function, which is presented in Figure 7. While it looks somewhat complicated, it simply runs across the the advice store selecting all of the advice that are triggered by the current function and composes their bodies together. It performs the appropriate access control checks along the way.

[6]The use of _ here indicates that the first step, substitution for the bound variable, has been completed.

Properties. We have proven that our type system is sound with respect to our operational semantics using Progress and Preservation theorems. This strategy requires that we extend the typing relation to cover all of the run-time in terms in the language as well as the other elements of the abstract machine (*i.e.,* the code store and aspect store). We omit the the details due to lack of space. The final judgment defining the well-typed abstract machine states has the form $\vdash M$ `ok`. The statement of Progress and Preservation follows.

THEOREM 3.1 (PROGRESS). *If* $\vdash < C, A, e >$ `ok` *then either* e *is a value, or* e *is* (`return` v `to cfp(`l`)`)*, or there exists a machine state* M *such that* $< C, A, e > \longmapsto M$.

THEOREM 3.2 (PRESERVATION). *If* $\vdash M$ `ok` *and* $M \longmapsto M'$ *then* $\vdash M'$ `ok`.

4. DISCUSSION

4.1 Implementation

We are in the process of implementing our language as an extension to SML/NJ. Our implementation parses programs written in the syntax explored in Section 2. They are elaborated into an extension of SML/NJ's ABSYN internal language, where typechecking occurs. Our extensions of ABSYN correspond closely with the formal language described in Section 3. We compile away the aspect-oriented features during the type-preserving translation from ABSYN to the main intermediate representation, a typed lambda calculus called LAMBDA. One of the advantages of our design is that all the implementation work occurs in the core language. The module system and its sophisticated semantics remain intact. The compilation manager does as well since, unlike some aspect-oriented languages, AspectML supports separate compilation. At the time of writing, we have implemented the parser and typechecker for the language and are working on the translation. We hope to complete it shortly.

4.2 Future Work

There are many directions for future research; the most interesting are the design choices concerning advice for polymorphic functions. Consider the identity function `id` with type $\forall\alpha.\alpha \to \alpha$. What type can we give a (read-only) CFP for this function? We might try $\forall\alpha.\{\texttt{arg}_r : \alpha, \texttt{res}_r : \alpha\}$. In this case, we could instantiate α with the type `int` and write well-typed advice with the form

```
advice before <| [$id[int]] as p(x:int) |> @ _ =
  print (Int.toString x)
```

In order for this scheme to be sound, this advice must only be triggered when the identity function is used at type `int`. Consequently, this strategy requires that we interpret ML polymorphism using a type-passing semantics.

A second alternative is to generalize the type of CFPs to allow type variables to be bound within the constructor: $\{\forall\alpha.(\texttt{arg}_r : \alpha, \texttt{res}_r : \alpha)\}$. In this case, the usual type instantiation would be disallowed; instead, programmers would be able to declare advice with bodies that operate parametrically with respect to α. Currently, we only support the `label` operation on CFPs from polymorphic functions.

$$\boxed{\texttt{compose}(t, A, l{:}\tau, v, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda z{:}\tau.e}$$

$$\frac{}{\texttt{compose}(t, \cdot, l{:}\tau, v, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda z{:}\tau.z}$$

$$\frac{\texttt{cfp}(l) \in v \quad \texttt{compose}(t, A, l{:}\tau, v_s, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda z{:}\tau.e' \quad \texttt{readcheck}(t, \{a_i^{i\in 1..n}\}, x, e)}{\texttt{compose}(t, \texttt{advice}\, t\, \texttt{stable}(v\, \texttt{as}\, p(x)\, @\, y) = e, A, l{:}\tau, v_s, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda x{:}\tau.(e[\texttt{cfp}(l)/p][v_s/y]; (\lambda z{:}\tau.e')\ x)}$$

$$\frac{\texttt{cfp}(l) \in v \quad \texttt{compose}(t, A, l{:}\tau, v_s, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda z{:}\tau.e' \quad \texttt{readcheck}(t, \{a_i^{i\in 1..n}\}, x, e) \quad \texttt{writecheck}(t, \{a_i^{i\in 1..n}\})}{\texttt{compose}(t, \texttt{advice}\, t\, \texttt{volatile}(v\, \texttt{as}\, p(x)\, @\, y) = e, A, l{:}\tau, v_s, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda x{:}\tau.((\lambda z{:}\tau.e')\, (e[\texttt{cfp}(l)/p][v_s/y]))}$$

$$\frac{\texttt{cfp}(l) \notin v \quad \texttt{compose}(t, A, l{:}\tau, v_s, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda z{:}\tau.e'}{\texttt{compose}(t, \texttt{advice}\, t\, vol(v\, \texttt{as}\, p(x)\, @\, y) = e, A, l{:}\tau, v_s, \{a_i^{i\in 1..n}\}) \Rightarrow \lambda z{:}\tau.e'}$$

Figure 7: Operational semantics: Aspect Composition

4.3 Related Work

Our language design is inspired by earlier work by Walker, Zdancewic and Ligatti [9], which was in turn influenced by earlier work on the semantics of aspect-oriented languages [10, 3, 7]. WZL define a low-level, typed calculus with first-class advice and CFPs. They then show how some conventional, high-level features from aspect-oriented languages like AspectJ can be compiled into the low-level calculus. The design in this paper is derived from elements of both the high-level (before and after advice) and low-level languages (the returnto statement). We then added useful variations of the ideas (*e.g.,* stable and volatile advice; our stack patterns), developed the access controls and their type system, and studied the interactions between modules, information hiding and advice. Another difference is that we give a semantics to our language directly rather than defining it by a translation from high-level language to low-level language.

The only other functional aspect-oriented language that we are aware of is the Scheme derivative developed by Tucker and Krishnamurthy [8]. Among other things, they explain how functional programmers can make effective use of first-class advice. However, since their language is untyped, programmers have no mechanisms (beyond static scoping) to protect their code from unwanted interference by aspects, which is the central topic of this paper.

Jagadeesan, Jeffrey and Riely have developed a type system for an object- and aspect-oriented language [4], which extends their earlier work on untyped languages [5]. They show that weaving, the translation that implements aspect-oriented features in terms of object-oriented features, preserves typing. This work is orthogonal to our main research results concerning access controls. However, it would be interesting to know how to transfer our results from a functional to an object-oriented setting and Jagadeesan's calculus might be a good place to start this investigation.

5. REFERENCES

[1] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In M. Wirsing, editor, *Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13, New York, Aug. 1991. Springer-Verlag. Volume 528 of *Lecture Notes in Computer Science*.

[2] Aspect-oriented programming. In T. Elrad, R. E. Filman, and A. Bader, editors, *Special Issue of Communications of the ACM*, volume 40. ACM Press, Oct. 2001.

[3] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Third International Conference on Metalevel architectures and separation of crosscutting concerns*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Sept. 2001. Springer-Verlag.

[4] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of typed aspect-oriented programs. Unpublished manuscript., 2003.

[5] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *European Conference on Object-Oriented Programming*, Darmstadt, Germany, July 2003.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.

[7] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002.

[8] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167, 2003.

[9] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ACM International Conference on Functional Programming*, Uppsala, Sweden, Aug. 2003.

[10] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In G. T. Leavens and R. Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop*, pages 17–25, Apr. 2002. Iowa State University University technical report 02-06.