AspectML: A polymorphic aspect-oriented functional programming language

DANIEL S. DANTAS and DAVID WALKER
Princeton University
{ddantas,dpw}@cs.princeton.edu
and
GEOFFREY WASHBURN and STEPHANIE WEIRICH
University of Pennsylvania
{geoffw,sweirich}@cis.upenn.edu

This paper defines AspectML, a typed functional, aspect-oriented programming language. The main contribution of AspectML is the seamless integration of polymorphism, run-time type analysis and aspect-oriented programming language features. In particular, AspectML allows programmers to define type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points. AspectML also comes equipped with a type inference algorithm that conservatively extends Hindley-Milner type inference. To support first-class polymorphic point-cut designators, a crucial feature for developing aspect-oriented profiling or logging libraries, the algorithm blends the conventional Hindley-Milner type inference algorithm with a simple form of local type inference.

We give our language operational meaning via a type-directed translation into an expressive type-safe intermediate language. Many complexities of the source language are eliminated in this translation, leading to a modular specification of its semantics. One of the novelties of the intermediate language is the definition of polymorphic labels for marking control-flow points. When a set of labels is assembled as a pointcut, the type of each label is an instance of the type of the pointcut.

Categories and Subject Descriptors: D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features—*abstract data types, polymorphism, control structures*; F.3.3 [**LOGICS AND MEANINGS OF PROGRAMS**]: Software—*type structure, program and recursion schemes, functional constructs*; F.4.1 [**MATHEMATICAL LOGIC AND FORMAL LANGUAGES**]: Mathematical Logic—*Lambda calculus and related systems*

General Terms: Design, Languages, Security, Theory

Additional Key Words and Phrases: Aspect-oriented programming, Functional languages, Parametric and ad-hoc polymorphism, Type systems, Type inference

---

This research was supported in part by ARDA Grant no. NBCHC030106, National Science Foundation grants CCR-0238328, CCR-0208601, and 0347289 and an Alfred P. Sloan Fellowship. This work does not necessarily reflect the opinions or policy of the federal government or Sloan foundation and no official endorsement should be inferred.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0164-0925/20YY/0500-0001 $5.00

1. INTRODUCTION

Aspect-oriented programming languages allow programmers to specify *what* computations to perform as well as *when* to perform them. For example, AspectJ [Kiczales et al. 2001] makes it easy to implement a profiler that records statistics concerning the number of calls to each method. The *what* in this example 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 *pointcut designator*. A collection of pointcut designators and advice organized to perform a coherent task is called an *aspect*.

The profiler described above could be implemented without aspects by placing the profiling code directly into the body of each method. However, at least four problems arise when the programmer does the insertion manually.

—First, it is no longer easy to adjust when the advice should execute, as the programmer must explicitly extract and relocate calls to profiling functions. Therefore, for applications where the "when" is in rapid flux, aspect-oriented languages are clearly superior to conventional languages.

—Second, there may be a specific convention concerning how to call the profiling functions. When calls to these functions are spread throughout the code base, it may be difficult to maintain these conventions correctly. For example, IBM [Colyer and Clement 2004] experimented with aspects in their middleware product line, finding that aspects aided in the consistent application of cross-cutting features such as profiling and improved the overall reliability of the system. Aspect-oriented features added structure and discipline to IBM's applications where there previously was none.

—Third, when code is injected directly into the body of each method, the code becomes "scattered," in many cases making it difficult to understand. This problem is particularly relevant to the implementation of security policies for programs. Many security experts have argued convincingly that security policies for programs should be centralized using aspects. Otherwise, security policy implementations are spread among many modules, making it impossible for a security expert to audit them effectively. Several researchers have implemented security systems based on this principle (though many of the experts did not use the term "aspect-oriented") and presented their ideas at prestigious conferences including POPL, PLDI and IEEE Security and Privacy [Evans and Twyman 1999; Kim et al. 1999; Lee et al. 1999; Colcombet and Fradet 2000; Erlingsson and Schneider 1999; 2000; Bauer et al. 2005].

—Fourth, in some situations, the source code is unavailable to be examined or modified. For example, the source code may belong to a third party who is not willing to provide this proprietary information to its users. Consequently, manual insertion of function calls is not possible. In these cases, aspects can be used as a robust form of external software patching [Fiuczynski et al. 2005].

To date there has been much success integrating aspects into object-oriented languages, but much less research on the interactions between aspects and typed functional languages. One of the central challenges is constructing a type system that

ensures safety, yet is sufficiently flexible to fit aspect-oriented programming idioms. In some situations, typing is straightforward. For instance, when defining a piece of advice for a single monomorphic function, the type of the argument to, and result of, the advice is directly connected to the type of the advised function. However, many aspect-oriented programming tasks, including the profiling task mentioned above, are best handled by a single piece of advice that executes before, after, or around many different function calls. In this case, the type of the advice is not directly connected with the type of a single function, but with a whole collection of functions. To type check advice in such situations, one must first determine the type for the collection and then link the type of the collection to the type of the advice. Normally, the type of the collection (the pointcut) will be highly polymorphic, and the type of each element will be less polymorphic than the collection's type.

In addition to finding polymorphic types for pointcuts and advice, it is important for advice to be able to change its behavior depending upon the type of the advised function. For instance, profiling advice might be specialized so that on calls to functions with integer arguments, it tracks how often that argument is the value 0. This and other similar examples require that the advice can determine the type of the function argument. In AspectJ and other object-oriented languages where subtype polymorphism is predominant, downcasts are used to determine types. However, in ML, and other functional languages, parametric polymorphism is predominant, and therefore run-time type analysis is the appropriate mechanism.

In this paper, we develop a typed functional programming language with first-class, polymorphic pointcuts, run-time type analysis and a conservative extension of ML's Hindley-Milner type inference algorithm. The language we define contains before, after, and around advice and is *oblivious* [Filman and Friedman 2005]. In other words, programmers can add functionality to a program "after-the-fact" in the typical aspect-oriented style. Furthermore, pointcuts in our language are first-class objects, an important feature for building effective aspect-oriented libraries. To provide support for stack-inspection-like security infrastructure, and to emulate AspectJ's CFlow, our language also includes a general mechanism for analyzing metadata associated with functions on the current call stack.

To specify the dynamic semantics of our language, we give a type-directed translation from the source into a type-safe core calculus with its own operational semantics. This strategy follows previous work by Walker, Zdancewic and Ligatti (WZL) [2003], who define the semantics of a monomorphic language in this way. Defining the operational semantics of a complex language via a translation to a simpler one has considerable precedent. Our translation could be seen as providing a denotational semantics for AspectML. Harper and Stone developed an improved semantics for Standard ML by elaboration into a simpler language [1998]. More recently, Avgustinov, et al. have given the first rigorous semantics for the AspectJ pointcut language via a translation to Safe Datalog [2007].

Our use of a translation helps to modularize the semantics for the source language by unraveling complex source-language objects into simple, orthogonal core calculus objects. Indeed, as in WZL, we have worked very hard to give a clean semantics to each feature in this language, and to separate unrelated concerns. We believe this will facilitate further exploration and extension of the language.

Furthermore, we believe that defining the semantics of AspectML this way makes it easier to understand the language. It is possible to focus on comprehending the core language and the translation separately, rather than the composition of the two. This modular design also makes formal reasoning about the language considerably easier. This is important as mechanically verified proofs become the norm. For example, it is unlikely that Lee, Crary, and Harper would have been able to provide a mechanically verified semantics for Standard ML if they had worked directly from the *The Definition of Standard ML* [2007; 1997]. We have not mechanically formalized the definition of AspectML, but our design has made it possible for us to prove important and non-trivial properties of the language.

Our core calculus, though it builds on WZL, is itself an important contribution of our work. One of the novelties of the core calculus is its first-class, polymorphic labels, which can be used to mark any control-flow point in a program. Unlike in WZL, where labels are monomorphic, polymorphism allows us to structure the labels in a tree-shaped hierarchy. Intuitively, each internal node in the tree represents a group of control-flow points, while the leaves represent single control-flow points. Depending upon how these labels are used, there could be groups for all points just before execution of a function or just after; groups for all labels in a module; groups for getting or setting references; groups for raising or catching exceptions, etc. Polymorphism is crucial for defining these groups since the type of a parent label, which represents a group, must be a polymorphic generalization of the type of each member of the group.

This paper expands and builds upon the contributions of our earlier work presented at the 10th ACM SIGPLAN International Conference on Functional Programming [Dantas et al. 2005a].

—We formally define a surface language, called idealized AspectML,[1] that includes three novel features essential for aspect-oriented programming in a strongly-typed functional language: polymorphic pointcuts, polymorphic advice and polymorphic analysis of metadata on the current call stack. In addition, we add run-time type analysis, which, though not a new feature, is seamlessly integrated into the rest of the language.

—We define a conservative extension of the Hindley-Milner type inference algorithm for idealized AspectML. In the absence of aspect-oriented features and run-time type analysis, type inference works as usual; inference for aspects and run-time type analysis is integrated into the system smoothly through a novel form of local type inference. Additionally, we believe the general principles behind our type inference techniques can be used in other settings.

—We define an explicitly-typed core calculus, $\mathbb{F}_{\mathsf{A}}$, that carefully separates mechanisms for polymorphic first-class function definition, polymorphic advice definition, and run-time type analysis. This core calculus introduces a new primitive notion of polymorphic labeled control flow points, to specify pointcuts in an orthogonal manner. We prove that this core calculus is type safe.

[1] In our earlier version of this work, we called our language PolyAML [Dantas et al. 2005b]. We have since chosen to rename it to emphasize the aspect-oriented nature of the language and to avoid confusion with the implementation of Standard ML called Poly/ML [Matthews 2005].

```
(polytypes)         s  ::= <ā> t
(pointcut type)     pt ::= (<ā> t1 ~> t2)
(monotypes)         t  ::= a | Unit | String | Stack | t1 -> t2 | pc pt
(trigger time)      tm ::= before | after | around
(terms)             e  ::= x | () | c | e1 e2 | let d in e | stkcase e1 (p=>e |_=> e2)
                        |  typecase<t> a (t=>e |_=> e) | #x:pt# | any | e:t
(stack patterns)    p  ::= x | [] | f::p
(frame patterns)    f  ::= _ | (|e|)<ā> (x:t,y)
(declarations)      d  ::= fun x1 <ā> (x2:t1):t2 = e
                        |  advice tm (|e1|) <ā> (x:t1,y,z):t2 = e2
                        |  case-advice tm (|e1|) (x:t1,y,z):t2 = e2
```

Fig. 1. Syntax of idealized AspectML

—We define the semantics of AspectML by a translation into $\mathbb{F}_A$. We prove that the translation is type-preserving, and therefore that the surface language is also type safe.

—We have implemented an extended version of idealized AspectML for use as a research language. Therefore we included a number of advanced features necessary for studying the properties and expressiveness of a modern, functional, aspect-oriented language.[2] Our implemented source language is similar in scope to core Standard ML.

—We have used the implementation to write several example programs to demonstrate the usefulness of AspectML, including a security case study. The case study examines an AspectML implementation of the Java stack inspection security mechanism.

In the remaining sections of this paper, we define and critically analyze our new polymorphic, functional and aspect-oriented programming language AspectML. Section 2 introduces the AspectML syntax and informally describes the semantics through a series of examples. Section 3 discusses our implementation and its use in a security case study of the Java stack inspection mechanism. Section 4 describes the formal semantics of the AspectML type system and type inference algorithm. Section 5 introduces the semantics of our polymorphic core calculus, $\mathbb{F}_A$. In Section 6, we give a semantics to AspectML in terms of $\mathbb{F}_A$. Finally, Sections 7 and 8 describe related work and conclusions.

2. PROGRAMMING WITH ASPECTS

AspectML is a polymorphic functional, aspect-oriented language based on the ML family of languages. Figure 1 presents the syntax of the idealized version of the language. However, all of the examples of this section are written in full AspectML, which extends this syntax with many common constructs following Standard ML.[3]

[2]Our implementation is available at http://www.cs.princeton.edu/sip/projects/aspectml/

[3]Details about the full language are available from the documentation that accompanies the implementation.

In Figure 1 and elsewhere, we use over-bars to denote lists of syntactic objects: $\overline{x}$ refers to a sequence $x_1 \ldots x_n$, and x_i stands for an arbitrary member of this sequence. Bold-faced text indicates program text, as opposed to mathematical meta-variables. We assume the usual conventions for variable binding and α-equivalence of types and terms.

As in ML, the type structure of AspectML is divided into *polytypes* and *monotypes*. Polytypes are normally written **<ā>** t where ā is a list of binding type variables and t is a monotype. However, when ā is empty, we abbreviate **<>** t as t.

In addition to type variables, a, simple base types like **Unit**, **String** and **Stack**, and function types t_1 **->** t_2, the monotypes include **pc** pt, the type of a pointcut, which in turn binds a list of type variables in a pair of monotypes. We explain pointcut types in more detail later. However, note that in AspectML, the word "monotype" is a slight misnomer for the syntactic category t as some of these types contain internal binding structure.

AspectML expressions include variables, x, constants like unit, **()**, and strings, c, function application and let declarations. New functions may be defined in let expressions. These functions may be polymorphic, and they may or may not be annotated with their argument and result types. Furthermore, if the programmer wishes to refer to type parameters within these annotations, they must specify a binding set of type variables **<ā>**. When the type annotations are omitted, AspectML will infer them. It is straightforward to extend idealized AspectML with other features such as integers, arithmetic, file and network I/O, tuples, and pattern matching, and we will make use of such constructs in our examples.

The most interesting features of our language are pointcuts and advice. Advice in AspectML is second-class and includes two parts: the body, which specifies what to do, and the *pointcut designator*, which specifies when to do it. In AspectML, a pointcut designator has two parts, a *trigger time*, which may either be **before**, **after**, or **around**, and a *pointcut* proper, which is a set of function names. The set of function names may be written out verbatim as **#f̄#**, or, to indicate all functions, a programmer may use the keyword **any**. In idealized AspectML, it is always necessary to provide a type annotation **#f̄:pt#** on a pointcut formed from a list of functions. In our implementation, this annotation is often not necessary.

Informally, the pointcut type, **(<ā>** t_1 **~>** t_2**)**, describes the I/O behavior of a pointcut. In AspectML, pointcuts are sets of functions, and t_1 and t_2 approximate the domains and ranges of those functions. For example, if there are functions **f** and **g** with types **String -> String** and **String -> Unit** respectively, the pointcut **#f,g#** has the pointcut type **pc (<a> String ~> a)**. Because their domains are equal, the type **String** suffices. However, they have different ranges, so we use a polytype that generalizes them both, **<a> a**. Any approximation is a valid type, so it would have also been fine to annotate the pointcut **#f,g#** with the pointcut type **pc (<a b> a ~> b)**. This latter type, is the most general pointcut type, and can be the type for any pointcut, including **any**. The semantics of pointcut types is given precisely in Section 4.

The pointcut designator **before (| #f# |)** represents the point in time immediately before executing a call to the function **f**. Likewise **after (| #f# |)** represents the point in time immediately after execution. The pointcut designator **around (| #f# |)**

wraps around the execution of a call to the function **f** – the advice triggered by the pointcut controls whether the function call actually executes or not.

The most basic kind of advice has the form:

$$\textbf{advice}\ \texttt{tm}\ \texttt{(|}e_1\texttt{|)}\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:}t_1\texttt{,y,z):}t_2 = e_2$$

Here, tm (|e_1|) is the pointcut designator. When the pointcut designator dictates it is time to execute the advice, the variable x is bound either to the argument (in the case of **before** and **around** advice) or to the result of function execution (in the case of **after** advice). The set of binding type variables, <a>, allows the types quantified by the pointcut to be named within the advice. However, the binding specification may be omitted if there are no quantified types, or if they are unneeded. The variable x may optionally be annotated with its type, t_1. The variable y is bound to the current call stack. We explain stack analysis in Section 2.2. The variable z is bound to metadata describing the function that has been called. In idealized AspectML this metadata is a string corresponding to the function name as written in the source text, but in the implementation it includes not just the name of the function, but the originating source file and line number. In the future it might also include security information, such as a version number or the name of the code signer. Since advice exchanges data with the designated control flow point, **before** and **after** advice must return a value with the same type as the first argument x. For **around** advice, x has the type of the argument of the triggering function, and the advice must return a value with the result type of the triggering function.

A common use of aspect-oriented programming is to add tracing information to functions. These statements print out information when certain functions are called or return. For example, we can advise the program below to display messages before any function is called and after the functions **f** and **g** return. The trace of the program is shown on the right in comments.

```
(* code *)                 (*  Output trace    *)
fun f x = x + 1            (*                  *)
fun g x = if x then f 1    (*  entering g      *)
               else f 0    (*  entering f      *)
fun h _ = False            (*  leaving f => 2  *)
                           (*  leaving g => 2  *)
                           (*  entering h      *)

(* advice *)
advice before (| any |) (arg, _, info) =
  (print ("entering " ^ (getFunName info) ^ "\n"); arg)

advice after (| #f,g# |) (arg, _, info) =
  (print ("leaving " ^ (getFunName info) ^
          " => " ^ (int_to_string arg) ^ "\n");
   arg)

val _ = h (g True)
```

Even though some of the functions in this example are monomorphic, polymorphism is essential. Because the advice can be triggered by any of these functions and they have different types, the advice must be polymorphic. Moreover, since the argument types of functions **f** and **g** have no type structure in common, the argument **arg** of the **before** advice must be completely abstract. On the other hand, the result types of **f** and **g** are identical, so we can fix the type of **arg** to be **Int** in the after advice.

In general, the type of the **after** advice argument may be the most specific type `t` such that the result types of all functions referenced in the pointcut are instances of `t`. Inferring `t` is not a simple unification problem; instead, it requires *anti-unification* [Plotkin 1970; 1971]. Our current implementation can often use anti-unification to compute this type, we describe when this is possible in Section 4.3.

Finally, we present an example of **around** advice. Again, **around** advice wraps around the execution of a call to the functions in its pointcut designator. The **arg** passed to the advice is the argument that would have been passed to the function had it been called. Finally, **around** advice introduces into the environment the **proceed** function. When applied to a value, **proceed** continues the execution of the advised function with that value as the new argument. Note that **proceed** is not a keyword and may be shadowed by variable binding.

In the following example, a cache is installed "around" the **f** function. First, a cache (**fCache**) is created for the **f** function with the externally-defined **cacheNew** command. Then, **around** advice is installed such that when the **f** function is called, the argument to the function is used as a key in a cache lookup (using the externally-defined **cacheGet** function). If a corresponding entry is found in the cache, the entry is returned as the result of the function. If the corresponding entry is not found, a call to **proceed** is used to invoke the original function. The result of this call is placed in the cache (using the externally-defined **cachePut** function) and is returned as the result of the **f** function.

```
val fCache : Ref List (Int,Int) = cacheNew ()

advice around (| #f# |) (arg, _, _) =
  case (cacheGet (fCache, arg))
    of Some res => res
     | None     => let
                     val res = proceed arg
                     val _ = cachePut (fCache, arg, res)
                   in
                     res
                   end
```

We note that we can transform this example into a general-purpose cache inserter by wrapping the cache creation and **around** advice code in a function that takes a first-class pointcut as its argument as described in Section 2.3. Finally, though not shown here, the **cacheGet** and **cachePut** functions are polymorphic functions that can be called on caches with many types of keys. As such, the key comparisons use

a polymorphic equality function that relies on the run-time type analysis described in the next section.

2.1 Run-time type analysis

We might also want a tracing routine to print not only the name of the function that is called, but also its argument. AspectML makes this extension easy with an alternate form of advice declaration, called **case-advice**, that is triggered both by the pointcut designator and the specific type of the argument. In the code below, the second piece of advice is only triggered when the function argument is an integer, the third piece of advice is only triggered when the function argument is a boolean, and the first and fourth pieces of advice are triggered by any function call. Advice is maintained as a stack, so all advice that is applicable to a program point is triggered in LIFO order.

```
advice before (| any |)(arg, _, info) = (print "\n"; arg)

case-advice before (| any |)(arg : Int, _, _) =
  (print (" with arg " ^ (int_to_string arg)); arg)

case-advice before (| any |)(arg : Bool, _, _) =
  (print (" with arg " ^ (if arg then "True" else "False")); arg)

advice before (| any |)(arg, _, info) =
  (print ("entering " ^ (getFunName info)); arg)
```

The code below and its trace demonstrates the execution of the advice. Note that even though **h**'s argument is polymorphic, because **h** is called with an **Int**, the third advice above triggers instead of the first.

```
(* code *)                (*  Output trace              *)
fun f x = x + 1           (*                            *)
fun g x = if x then f 1   (*  entering g with arg True  *)
               else f 0   (*  entering f with arg 1     *)
fun h _ = False           (*  entering h with arg 2     *)
val _ = h (g True)
```

This ability to conditionally trigger advice based on the type of the argument means that polymorphism is not parametric in AspectML– programmers can analyze the types of values at run-time. However, without this ability we cannot implement this tracing aspect and other similar examples. For further flexibility, AspectML also includes a typecase construct to analyze type variables directly. For example, consider the following advice:

```
advice before (| any |)<a b>(arg : a, _, info) =
  (print ("entering " ^ (getFunName info) ^ "with arg" ^
          (typecase<String> a
             of Int => (int_to_string arg) ^ "\n"
              | Bool => (if arg then "True\n" else "False\n")
              | _    => "<unprintable>\n"));
   arg)
```

This advice is polymorphic, and the argument type **a** is bound by the annotation **<a b>**.[4] Also note that in the example above, to aid typechecking the typecase expression, the return type is annotated with **<String>**. Finally, note that because this example is written in full AspectML, **typecase** operates over type patterns rather than types. However, the only difference between types and type patterns is that type patterns provide underscore as a wildcard pattern.

There is a nice synergy between aspects and run-time type analysis. Converting values to strings is an operation that is generally useful, so one might imagine implementing it as a library function **val_to_string** to be called by the above advice:

```
fun val_to_string <a>(v:a):String =
    typecase<String> a
      of Bool => bool_to_string v
       | String => v
       | Int => int_to_string v
       | (a, b) => "(" ^ (val_to_string (fst v)) ^ ", " ^
                         (val_to_string (snd v)) ^ ")"
       | _ => "<unprintable>"
```

Notice that this solution requires that **val_to_string** be revised every time a new data type is defined by the user (like ML, the full AspectML language allows the creation of new algebraic data types). Instead, we switch to using an **around case-advice** idiom:

```
fun val_to_string <a> (v:a):String = "<unprintable>"

case-advice around (| #val_to_string# |) (v:Bool, _, _) =
  bool_to_string v

case-advice around (| #val_to_string# |) (v:String, _, _) = v

case-advice around (| #val_to_string# |) (v:Int, _, _) =
  int_to_string v

case-advice around (| #val_to_string# |) (v:(a,b), _, _) =
  "(" ^ (val_to_string (fst v)) ^ ", " ^ (val_to_string (snd v)) ^ ")"
```

Each piece of **around case-advice** overrides the default behavior of the function when passed an argument of a particular type. Notice that there are no **proceed** calls – we do not want the function to continue to execute once the correct string conversion function has been selected.

Now, when a programmer defines a data type, they can also update the **val_to_string** function to be able to convert values of their new datatype to a string. For example,

[4]Currently this example requires the type variable `b` to be bound, even though it never occurs in the code fragment. The pointcut **any** is of type `pc (<ab> a ~> b)`, and if the programmer wishes to name one of quantified variables, in this case `a`, she must name all of the variables. This is because the quantified variables are unordered and the type inference algorithm cannot naïvely choose which of the two variables to name when only given a single name.

we can define a **List** data type and its output function in the same location[5]

```
datatype List = Cons : <a> a -> List a -> List a
              | Nil  : <a> List a

case-advice around (| #val_to_string# |) (v:List a, _, _):String =
    case v
      of Cons h t => (val_to_string h) ^ " :: " ^ (val_to_string t)
       | Nil => "Nil"
```

2.2 Reifying the context

When advice is triggered, often not only is the argument to the function important, but also the context in which it was called. Therefore, this context information is provided to all advice and AspectML includes constructs for analyzing it. For example, below we augment the tracing aspect so that it displays debugging information for the function **f** when it is called directly from **g** and **g**'s argument is the boolean **True**.

```
advice before (| #f# |)(farg, fstk, _) =
  ((case fstk
      of _ :: (| #g# |)(garg, _) :: _ =>
         if garg then
           print "entering f from g(True)\n"
         else ()
       | _ => ());
   farg)
```

The stack argument **fstk** is a list of **Frame**s, which may be examined using case analysis.[6] Each frame in the list **fstk** describes a function in the context and can be matched by a frame pattern: either a wild-card _ or the pattern **(|e|)**<ā>**(x,y)**. The expression e in a frame pattern must evaluate to a pointcut – the pattern matches if any function in the pointcut matches the function that frame describes. Like in advice, the type variable binders are used to optionally name types quantified by the pointcut. The variable x is the argument of that function, and y is the metadata of the function. The head of the list contains information about the function that triggered the advice (e.g. **f** in the example above).

Consider also the following example, that uses an aspect to implement a stack-inspection-like security monitor for the program. (We will expand the technique of using aspects to provide stack-inspection security in our case study in Section 3.) If the program tries to call an operation that has not been enabled by the current context, the security monitor terminates the program. Below, assume the function **enables:FunInfo -> FunInfo -> Bool** determines whether the first argument (a piece of function metadata) provides the capability for the second argument (another piece of function metadata) to execute. We also assume **abort:String -> Unit** terminates

[5]Note that AspectML has a different syntax for data type definition than Standard ML. Here a data type definition contains the name of the data type and then a list of the data type constructors, with their types.
[6]Idealized AspectML does not make lists primitive, but instead uses the primitive type **Stack** and primitive analysis **stkcase**.

the program with an error message. Note that **abort** is a system primitive, not a function, and cannot be advised.

```
fun walk (stk : List Frame, info : FunInfo) =
  case stk of [] => abort "Function not enabled"
            | (| any |)(_, info') :: rest =>
              if (enables info' info) then ()
              else
                walk (rest, info)

advice before (| #f,g,h# |)(arg, stk, info) = (walk (stk, info); arg)
```

2.3 First-class pointcuts

The last interesting feature of our language is the ability to use pointcuts as first-class values. This facility is extremely useful for constructing generic libraries of profiling, tracing or access control advice that can be instantiated with whatever pointcuts are useful for the application. For example, recall the first example in Section 2 where we constructed a logger for the **f** and **g** functions. We can instead construct an all-purpose logger that is passed the pointcut designators of the functions we intend to log with the following code (Recall that **val_to_string** is a function, defined in Section 2.1, that converts values of any type to a string)

```
fun startLogger (toLog:pc (<a b> a ~> b)) =
  let advice before (| toLog |)(arg, _, info) =
        ((print ("before " ^ (getFunName info) ^ ": " ^
               (val_to_string arg) ^ "\n")); arg)
      advice after (| toLog |) (res, _, info) =
        ((print ("after " ^ (getFunName info) ^ ":" ^
               (val_to_string res) ^ "\n")); res)
  in () end
```

Another example generalizes the "**f** within **g**" pattern presented above. This is a very common idiom; in fact, AspectJ has a special pointcut designator for specifying it. In AspectML we can implement the **within** combinator using a function that takes two pointcuts – the first for the callee and the second for the caller – as arguments. Whenever we wish to use the **within** combinator, we supply two pointcuts of our choice as shown below.

```
fun within (fpc : pc (<a b> a ~> b),
            gpc : pc (<c> Bool ~> c),
            body : Bool -> Unit) =
  let advice before (| fpc |)(farg, fstk, _) =
        (case fstk
           of _ :: (| gpc |)(garg, _) :: _ => body garg
            | _ => ();
         farg) in () end

fun entering x = if x then print "entering f from g\n" else ()

val _ = within (#f#, #g#, entering)
```

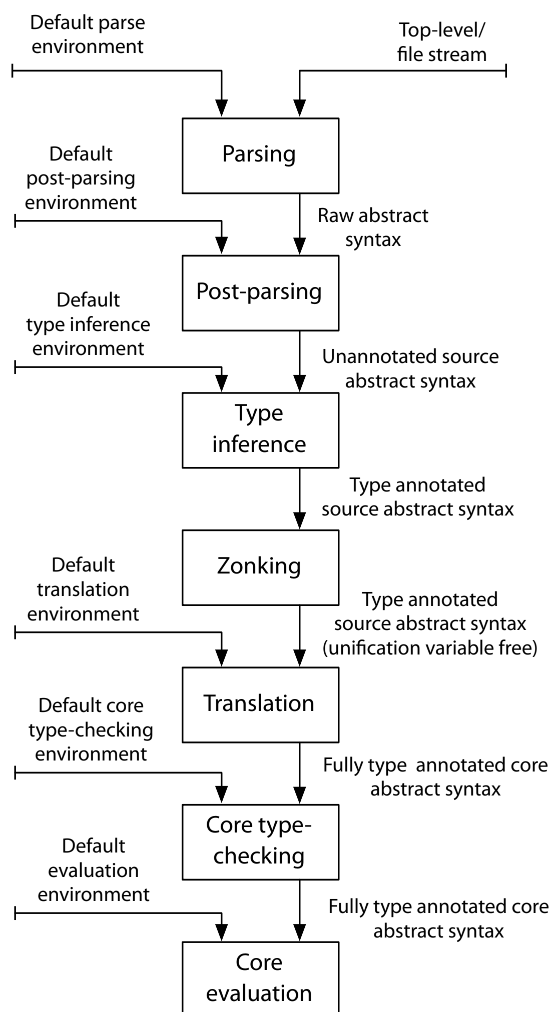

Fig. 2. AspectML implementation pipeline

Notice that we placed a typing annotation on the formal parameter of **`within`**. When pointcuts are used as first-class objects, it is not always possible to infer types of function arguments and results. The reason, described fully in Section 4, is that pointcut types have binding structure that cannot be determined via unification without the addition of type annotations.

2.4 AspectML Implementation

As we have described, our implementation of AspectML is very similar in scope to core Standard ML. Aside from a number of syntactic differences, AspectML currently lacks exceptions, type definitions, and records. Adding exceptions would require some thinking as we would likely want to add additional join points corresponding

with raising and catching exceptions. The other features would require additional engineering, but not much thought about language design.

However, full AspectML also includes a number of extensions beyond what we have described so far. Our implementation includes support for higher kinded type variables, data kinds in the style of Ωmega [Sheard 2005], polymorphic recursion, algebraic data types with existentials [Läufer and Odersky 1992], generalized algebraic data types (GADTs) [Peyton Jones et al. 2006], and a form of subsumption, inspired by first-class polymorphism [Peyton Jones et al. 2007], to allow pointcuts with a more specific type to be used when a more general one is required. Finally, AspectML includes a few specially designed features to aid polytypic programming, such as defining tuple types in terms of "list" kinds and functions for manipulating spine-based representations of algebraic data types [Hinze et al. 2006].

AspectML is implemented in SML/NJ. At the time of writing, the combined total of Standard ML code, ML-LEX and ML-YACC specifications, the AspectML basis library, and SML/NJ Compilation Manager configuration files totals approximately 15 100 lines of code.

Figure 2 provides an overview of our AspectML implementation. It is composed of a series of seven "processes" that consume a stream of data and potentially produce a stream of output.

Parsing. This stage transforms ASCII text, using standard regular-expression lexing and LALR(1) parsing techniques, into what we call *raw abstract syntax*. Raw abstract syntax captures the rough structure of program text, but does not disambiguate the lexical scope of variables or the application of prefix, postfix, and infix operators.

Post-parsing. This stage transforms raw abstract syntax into AspectML abstract syntax. The lexical scope of variables is established by incrementally building a finite map between the string representations of variables to our abstract representation of variables. The resulting abstract syntax tree follows the Barendregt naming convention as a consequence [1985]. An operator precedence parser resolves the application of prefix, postfix, and infix operators [Aho et al. 1986]. Additionally, the post-parser eliminates some syntactic sugar.

Type inference. This stage decorates the abstract syntax tree received from the post-parser with types, as inferred using an algorithm extending the one described in Section 4. The most significant difference between the full implementation and idealized AspectML is that instead of passing around a substitution, imperative type unification is used.

Zonking. This stage[7], eliminates the indirection created by the use of mutable references in unification, and instantiates unconstrained type unification variables to type **`Unit`**. The purpose of this stage is to ensure that the type of the program is fully determined and that translation does not encounter unification variables.

Translation. This stage transforms, using an extension of the algorithm in Section 6, an explicitly typed source language abstract syntax tree into an explicitly

[7]The colourful name for this stage is from Simon Peyton Jones [2007].

type core language aspect syntax tree. This core language is closely related to $\mathbb{F}_A$, described in Section 5.

Core type-checking. In full AspectML, this stage most closely matches what is formalized in this paper. This is because $\mathbb{F}_A$ is expressive enough to implement just about all of our extensions to idealized AspectML except defining and pattern matching on algebraic data types.

Core evaluation. Evaluation of the core calculus is quite different from the operational semantics described in Section 5. Firstly, we use an environment-based semantics rather than a substitution-based semantics. Secondly, while we retain a small-step semantics, instead of searching for the next redex, the actual implementation pushes *evaluation contexts* onto a stack and deterministically *focuses* upon the next reduction problem. As reduction problems are completed, evaluation contexts are popped off the stack. Evaluation then proceeds by repeatedly stepping the abstract machine until it reaches a finished or error state. This is reminiscent of Harper's C-machine [2005].

2.5 Design decisions

The **any** *pointcut.* Programming with the **any** pointcut takes some care. For example, we can modify our previous stack-inspection-like security example to use the **any** pointcut, instead of specifically stating which functions it is to be triggered on.

```
fun walk (stk : List Frame, info : FunInfo) =
  case stk of [] => abort ("Function not enabled")
            | (| any |)(_, info') :: rest =>
              if (enables info' info) then ()
              else
                walk (rest, info)

advice before (| any |)(arg, stk, info) =(walk (stk, info); arg)
```

Unfortunately, we discovered when we tested this program that that it will always diverge. The function calls in the body of the advice will trigger the advice itself.

This problem could be solved in a number of ways. One possibility would be to introduce a primitive, **disable** e, disables all advice while e is evaluated. The advice could then be rewritten as

```
advice before (| any |)(arg, stk, info) =
  (disable (walk (stk, info)); arg)
```

Another option would be to introduce subtractive pointcuts, such as e_1 **except** e_2, that behave here like set difference on names of functions. We could use this to rewrite the advice as

```
advice before (| any except #walk,enables# |) (arg, stk, info) =
 (walk (stk, info); arg)
```

This extension has the disadvantage that it the author of the advice must know the entire potential call tree for **walk** to properly specify the exception list.

Both of these extensions are straightforward to integrate into our type system, but the extensions would require some modifications to the core operational semantics we describe in Section 5.

Anonymous functions. Anonymous functions present several design choices for aspect-oriented languages. Because they are nameless, it is impossible to write explicit pointcuts for them. It is possible that generic pointcuts might implicitly advise them, such as **any**, or a new pointcut, **anon** that refers just to anonymous functions. There are no technical difficulties to this extension, but on the other hand, we do not yet see any compelling reasons for advising anonymous functions, either. Therefore, in our implementation, we have decided to make anonymous functions non-advisable until we have more experience with programming in AspectML.

Non-advisable functions. Programmers wary of allowing external aspects to modify the behavior of their functions may wish for the ability to define non-advisable functions. Adding non-advisable functions, perhaps by marking them with a **final** keyword, appears straightforward both in theory and practice – one need merely avoid creating a join point for such functions during translation.

Advising first-class functions. Another design choice we made is that explicit pointcuts, such as $\#\overline{f}\#$ may only refer to term variables that were let-bound to functions in the current lexical scope. In other words, as illustrated by the following example, variables corresponding to first-class functions cannot be used in a pointcut.

```
fun f x = x + 1
val ptc = #f#   (* allowed *)

fun h (g : Int -> Int) = #g#  (* not allowed *)
```

We have left this feature out because it increases the complexity of the semantics of AspectML without a corresponding increase in usefulness – we have not found any compelling examples that require advising first-class functions. There are no technical obstacles to advising first-class functions in AspectML. Furthermore, pointcuts are first-class in AspectML, so if the programmer needs access to **g**'s pointcut, **h** can be rewritten to take a pointcut as an additional argument.

More pointcuts. In a larger language, it might be desirable to extend the language of pointcuts to allow advising events other than function invocation. However, given the central importance of functions in AspectML, not as many extensions are necessary as might be needed an imperative language like AspectJ. For example, a Java programmer might find it useful to advise allocating, reading, and writing mutable state. However, all of these behaviors correspond to functions in AspectML that may be directly advised.

3. CASE STUDY

To demonstrate the usefulness of AspectML, we have implemented a dynamic security policy manager and stack inspection framework with most of the interesting components one finds in Java.

We have chosen to focus on security because it appears to be one of the best, most convincing applications of aspect-oriented programming technology. Indeed, many

previous researchers have argued that aspect-oriented programming mechanisms enable more modular implementation of access control infrastructure than standard programming languages. More specifically, since an aspect-oriented implementation can encapsulate not only the definition of *what* an access control check is supposed to do, but also the complete list of places *where* that access control check should occur, aspect-oriented policy specifications are easier to understand. This in turn makes policy specifications easier to audit – the security auditor need not search through thousands of lines of library or application code to find the relatively few lines of access control checks. In particular, analysis of the pointcut definitions used in a policy can often tell the auditor whether or not access control checks have been omitted. In addition, because all security code is centralized, when security vulnerabilities are identified, security policy updates can be made more easily. Moreover, to distribute the changed policy, a single new aspect can be deployed as opposed to an entire new library or set of libraries. Past systems that used this kind of aspect-oriented design (occasionally without acknowledging it as such) include Naccio [Evans and Twyman 1999], SASI [Erlingsson and Schneider 1999], Java MAC [Kim et al. 1999; Lee et al. 1999], PoET/Pslang [Erlingsson and Schneider 2000], and Polymer [Bauer et al. 2005]. All of these systems were situated in imperative or object-oriented languages and hence were substantially different from AspectML.

In the following sections, we first describe the Java security mechanism and then analyze our implementation of the algorithm in AspectML.

3.1 Permissions

The basic unit of protection is the permission. Java security defines many permissions including file system permissions, network socket permissions, system property permissions, and GUI permissions. A permission consists of a permission name and permission arguments that constitute the internal structure of the permission. For example, the file system permission name is **`FilePermission`**, and the permission arguments contain the access mask (read/write/delete) and the file path.

Permissions consist of granted permissions and requested permissions. Granted permissions represent a list of actions a function is allowed to perform. A requested permission represents a specific action that a function is attempting to perform. If the granted permissions of a function "imply" the requested permission, then the requested action can be performed. The specific mechanisms for granting, requesting, and testing permission implication are described in the following sections.

3.2 Policy Parsing

To determine what permissions will be granted to the executing source code, Java security reads from a policy file upon start-up. The policy file consists of a set of **`grant`** declarations, each of which specify a segment of source code and the list of permissions granted to that source code. Granted permissions are specified by the name of the permission followed by an list of arguments that describe the details of the permission. We use this same basic format to specify AspectML security files.

The following is the specification of Java and AspectML policy file syntax:

```
grant sourceCodeSpecifier {
  permission permissionName1 permissionArgs1;
  permission permissionName2 permissionArgs2;
  ...
};
...
```

The *sourceCodeSpecifier* selects the source code to which permissions are granted. A policy file writer is allowed to select code by specifying a **codebase** (where the code is located in the file system), a **signer** (the key that has cryptographically signed the code), and a **principal** (the entity that is executing the code). For the purposes of this case study, we have chosen to allow only the **codebase** specifier in AspectML policy files.

The *permissionName* specifies the name of the permission to grant to the selected source code, while the *permissionArgs* specifies the details of the permission. For the purposes of this case study, we have studied file system and network socket permissions in AspectML. A sample AspectML policy file follows.

```
grant codebase "example/*" {
  permission  FilePermission    "tmp/*", "read write";
  permission  SocketPermission  "*", "listen accept";
};
```

This would give all code in files in the **example/** directory permission to read and write to the **tmp/** directory. It also allows the code in the **example/** directory to start a network server that accepts connections from any host.

3.3 Permission Specification

As stated earlier, AspectML security currently specifies file system and network permissions. We have also provided functions in our security implementation that allow a user to add new permission types to the policy file and to the underlying security mechanism. The user must specify the name of the permission (used in policy file parsing), an **addPermission** function of type **String -> permission** that parses the permission arguments from the policy file and returns the resulting permission, and an **impliesPermission** function of type **(permission,permission) -> Bool** that takes in a granted permission and a requested permission and returns whether the first allows the second.

Accordingly, when we added a file system permission type to AspectML security, we specified the name **FilePermission** and an **addFilePermission** function which parses file paths and access mask strings like “read write” to an internal representation. Finally, we specified an **impliesFilePermission** function which takes a granted file system permission such as reading and writing to the **tmp/** directory and a requested read, write, or delete action and then determines whether the permission allows the action.

3.4 Stack Inspection

To determine whether a restricted action should be performed, the permissions granted by the policy file to the currently executing code are examined to determine

Java stack inspection pseudocode

```
public bool inspectStack (Stack currentStack) {
  for (StackFrame sf : currentStack) {
    if (stack frame sf does not allow the action) {
      return false;
    } else if (stack frame sf is marked as "privileged") {
      \\ note that stack frame sf allows the action
      return true;
    }
  }
  return true;
}
```

AspectML stack inspection code

```
fun checkStack (stk, requestedPermission) =
  case stk of
    (| #doPrivileged# |)(_,_) :: (|any|)(_,info) :: _ =>
      let
        val privFun = getFileName info
        val grantedPermissions = getPermissions (privFun, policyfile)
      in
        impliesPermissions (grantedPermissions, requestedPermission)
      end
  | (|any|)(_,info) :: stktail =>
      let
        val currFun = getFileName info
        val grantedPermissions = getPermissions (currFun, policyfile)
      in
        impliesPermissions (grantedPermissions, requestedPermission)
        andalso checkStack (stktail, requestedPermission)
      end
  | _ => True
```

Fig. 3. Stack inspection comparison: Java and AspectML

whether they imply the requested permission required by the restricted action. This test is not enough – if trusted code is called from untrusted code, the untrusted code may perform malicious actions by proxy. Therefore, Java security is a stack inspection mechanism, as described by the pseudo-code in Figure 3. The current stack frame and all subsequent frames must all have the required permission before an action is approved. Therefore, if the end of the stack is reached, which corresponds to wildcard pattern branch, `True` will be returned indicating that the requested action has been approved.

For example, if function **`f`** calls function **`g`** which calls function **`h`** which then attempts to read from a file, the call stack will look like this: **`[fileRead, h, g, f]`**. The Java security mechanism will ensure that **`h`**, **`g`**, and then **`f`** all have permission to read from the requested file.

Finally, if trusted code is certain that it can only be used in approved ways, it can disengage any further stack checks by performing a "privileged" action. This

is useful, for example, if a trusted function carefully checks its inputs or if the restricted action that it wishes to perform does not depend on data passed to it by its calling function. A "privileged" action is performed by calling a special security function **doPrivileged**, passing it the code to be run. When the security mechanism walks the stack, it will stop at whatever code called the **doPrivileged** function.

In the above example, if the function **g** is certain that it cannot be used inappropriately by code that calls it, it can perform a "privileged" call to the **h** function. The stack will appear as **[fileRead**, **h**, **doPrivileged**, **g**, **f]**. In this case, only **h** and **g** need to have the file read permission – **f** is not examined by the security mechanism.

The AspectML stack inspection algorithm is compared with its Java counterpart in Figure 3. As with Java stack inspection, there are three cases, a privileged stack frame, a regular stack frame, and the end of the stack.

3.5 Security Triggering

We now have described the policy parsing, permission implication, and stack inspection code. The final step is to trigger this security mechanism when a restricted action is performed. In Java security, every read-file system call in the source code is preceded by a call to **AccessController.checkPermission(requestedAction)**.

In our AspectML implementation, a pointcut is created that contains a list of the restricted actions that should trigger the aspect. For example, all of the system calls that will read from a file will trigger the **readPC** pointcut. Next, an aspect is created that, when the pointcut is triggered (before a restricted action is performed), creates the requested permission and checks that the stack allows this requested permission. In the following example, when code attempts to read from a file, the **readPC** pointcut is triggered, calling the **checkRead** function that creates the requested **FilePermission** and performs the security stack inspection check.

```
val readPC = #fileCanRead, fileExists, fileIsDirectory,
  fileIsFile, fileLastModified, fileLength, fileList,
  fileOpenRead#
advice before (| readPC |) (arg, stk, _) =
  if checkRead (stk, arg) then
    arg
  else
    abort "Failed security check"
```

Similar checks are performed when writing files, deleting files, and making and receiving network connections.

3.6 Issues

A difficulty we encountered during this stage was similar to a problem also described in another aspect-oriented case study [Sullivan et al. 2005] – occasionally the crosscutting location is not easily accessible with a pointcut. In the Java security mechanism, two security checks were in the middle of Java functions, not at the beginning or the end of the function. This disallowed easy use of **before** or **after** advice to institute the security check. Instead, we were required to split each affected function into two parts: a function which runs the pre-security-check code, and a

Pre-security getAllByName0 Java pseudocode

```
static InetAddress[] getAllByName0 (String host) {...}
```

Post-security getAllByName0 Java pseudocode

```
static InetAddress[] getAllByName0 (String host, boolean check) {
  if (check) { perform security check }
  ...
}

static InetAddress[] getAllByName0 (String host) {
  return getAllByName0(host, true);
}
```

AspectML getAllByName0 security

```
fun shouldCheck (stk) =
  case stk of
    (| #checkListen,checkAccept,checkConnect# |) (_,_) :: _ => False
  | (|any|) (_,_) :: stktail => shouldCheck stktail
  | _ => True

advice before (| #netGetAllByName0# |) (host, stk, _) =
    if (shouldCheck stk) andalso (not (checkConnect (stk, host, ~1))) then
      abort "Failed security check"
    else
      host
```

Fig. 4. Recursive security: Java and AspectML

second function which is called by the first and runs the post-security-check code. We trigger the security check as **before** advice on this post-security-check function.

Another issue emerged when we discovered that the network security code calls network I/O code which in turn triggers the Java network security code and so on. The Java security mechanism handles this by adding a flag to the argument list of the affected network I/O functions, indicating whether the function is being called from security code or not.

For example, in the top section of Figure 4, we display the pre-security pseudo-code for the **getAllByName0** function, which looks up the internet address that correspond to a given hostname string. Because this function both triggers the network security mechanism and is called by it, in the middle section of the figure, **getAllByName0** must be converted to a new function that takes both the hostname string and a flag marking whether the check should be performed or whether **getAllByName0** has been called from within the security mechanism. In addition, the function is overloaded so that calls to the old **getAllByName0** function (calls that are not from within the security mechanism) call the new function with a **check** flag value of **true**

We feel that this solution is suboptimal because it requires modifying the signature of the original network I/O code in order to add the security feature. In our AspectML implementation (displayed in the bottom section of the figure), there is no need to change the argument list of the function. Instead, we add a **shouldCheck** function

that performs stack analysis to determine whether the **getAllByName0** function has been called by the security mechanism or not. If it has, then no security check is performed by the advice to avoid infinite recursion.

3.7 History Inspection

As an aside, several researchers [Schneider 2000; Gordon and Fournet 2003; Abadi and Fournet 2003] have questioned whether stack inspection is the correct choice for enforcing common security policies. For example, in order to preserve the confidentiality of certain files, a user may wish to disallow making or receiving network connections after reading from the file system. As any file system reads will have occurred in the past and will no longer be on the stack when a network connection is attempted, a stack inspection mechanism will not suffice to enforce this policy. Instead, the entire execution history – the list of all the functions that have been run – must be examined. AspectML can enforce history-based policies just as easily as the stack-inspection based policies of the previous section. As an example, the following code implements the simple history-based policy for file confidentiality described above.

```
val didRead = ref false

advice after (| #fileOpenRead# |) (res, _, _) =
  ((didRead := true); res)

advice before (| #netConnect, netServerAccept# |) (arg, _, _) =
    if !didRead then
      abort "No network after file read\n"
    else
      arg
```

The first piece of advice sets a flag when any file is read. The second piece of advice disallows all network connections if the flag is set.

4. TYPE INFERENCE

Many modern statically typed languages use type inference to eliminate typing annotations that users must write. In Standard ML, type inference is so effective that users never need to provide any type annotations. The type system of AspectML is carefully designed to permit efficient type inference with an algorithm that is an extension of one used by many implementations of Standard ML, Damas and Milner's (DM) unification-based Algorithm $\mathcal{W}$ [1982]. Therefore, type inference in AspectML behaves exactly the same as ML for ML programs; all programs that do not include aspects, type analysis, or any other extensions will type check as they do in ML.

However, first-class polymorphic point cuts and runtime type analysis cause difficulties for unification-based type inference, which we discuss in more detail below. Programs that use these capabilities require annotation – AspectML does not permit complete inference. In our design, we have carefully balanced the number of required annotations with the complexity of type inference. We believe that programming with aspects is not tedious, yet it is easy for programmers to

understand where type annotations are required. Furthermore, many users provide typing annotations for function definitions in Standard ML as a form of checked documentation – most of the time such annotations are sufficient for AspectML.

The key idea of our type system is a distinction between two different modes of type inference: *local* and *global* typings. Local typings, which generalize the idea of type annotations, describe when the type of an expression can be determined without using unification. This process is a simplified version of local type inference [Pierce and Turner 1998]. Local typings always produce *rigid* types – that are guaranteed not to contain any uninstantiated unification variables. On the other hand, global typings make no such guarantees. They may use unification to determine the type of an expression, so the resulting type may not be rigid.

In the our implementation of AspectML there are only four places where some sort of type annotations are necessary.

(1) Advised pointcuts must have local typings so that the connection between the pointcut type and the argument of the pointcut is explicit. We discuss this issue in greater detail in Section 4.1.

(2) Pointcuts created from sets of function names often have local typings (making them suitable for advisement). However, if any of their member functions have types that are not rigid, this is not the case. The reason for this requirement is discussed in greater detail in Section 4.3. However, we expect that this should be a rare occurrence in practice. Furthermore, the AspectML implementation will explicitly tell the user when it does occur.

(3) The **typecase** expression must be annotated with a type for the entire expression. Furthermore, during type analysis, the types of some variables in the context may be *improved*, but only for variables with local types. These requirements will be explained in greater detail in Section 4.4.

(4) Type annotations inside polymorphic functions and advice may need to refer to type variables. Therefore, these type variables must be explicitly brought into scope with an annotation such as $<\bar{a}>$. Some languages, such as OCaml, Standard ML and Haskell, implicitly bring type variables into scope using *lexically scoped type variables* [Shields and Peyton Jones 2002], but we think that explicit binding is simpler to understand as it makes the binding site more apparent.

To better describe how type inference works in AspectML, in the rest of this section we precisely describe a type inference algorithm for idealized AspectML. However, before we do that, we discuss the problems with unification-based type inference and pointcut types that provide part of the motivation for the distinction between local and global typings.

4.1 Local and global typings

A key component of the DM type inference algorithm is the use of first-order unification. In idealized AspectML type inference, unification variables are notated by $X, Y, Z, \ldots$ and are only introduced by the type inference algorithm. Unification variables are distinct from rigid type variables, a, that result from programmer annotations and generalization.

Pointcut types, like **pc (<ā>** t_1 **~>** t_2**)**, are problematic for first-order unification because they bind type variables within monotypes.[8] For example, consider the following contrived, but illustrative, code fragment.

```
fun f p = let
            advice before (| p |) (x, _, _) = ...
          in
            ...
          end
```

In traditional DM type inference, while processing **f**, the variable **p** would have some unification variable, X, as its type. However, to give a type to **x** we need to know the domain component of the pointcut type. When the same problem occurs with any other type constructor, say function types, we would just construct a new function type with fresh unification variables, say Y **->** Z, unify X with Y **->** Z, and then use Y for the needed type. However, we cannot do the same for possibly polymorphic pointcut types for two reasons. First, it would be necessary to guess the correct set of binding variables **ā** to construct a pointcut type **pc (<ā>** Y **~>** Z**)**. Second, the final instantiation of the unification variables Y and Z may depend upon some of the variables in **ā** – a dependency that is not captured by first-order unification. Consequently, higher-order unification would be necessary to guess an appropriate pointcut type.

Higher-order unification is known to be undecidable, so to keep our inference algorithm for AspectML decidable, we solve the problem above by requiring that the type of **p** be locally known. That way, any binding in the pointcut type is apparent. For example, we can repair our example from above by annotating **p** at its binding site (an annotation at its use site would also suffice.)

```
fun f (p : pc (<a b> (a, b) ~> Int))  =
  let
    advice before (| p |) (x, _, _) = ...
  in
    ...
  end
```

4.2 The formal semantics of local and global typings

Our discussion of local and global inference so far has been very informal. We now flesh out the details precisely. All judgments used by type inference in this section can be found in Figure 5.

Both local and global type inference consume and produce Θ, an idempotent, ever-growing substitution of monotypes for unification variables. Substitutions have a composition operator, $- \circ -$, that is associative. This substitution is extended by unification. The unification judgment $\Theta \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta'$ is read as:

> "With input substitution Θ, types $\mathtt{t}_1$ and $\mathtt{t}_2$ unify, producing the extended substitution Θ'."

[8]Recall that ML carefully separates types (or monotypes) that may not bind type variables from type schemes (or polytypes), that may. Only monotypes are unified.

Contexts

(*term variable contexts*)	Γ	::=	$\cdot \mid \Gamma, x :: s \mid \Gamma, x : s$
(*function context*)	Φ	::=	$\cdot \mid \Phi, x$
(*type variable contexts*)	Δ	::=	$\cdot \mid \Delta, a$
(*substitutions*)	Θ	::=	$\cdot \mid \Theta, t/X$

Judgments

$\Theta \vdash t_1 = t_2 \Rightarrow \Theta'$	unification
$\Theta \vdash \texttt{<}\bar{a}\texttt{>}\ t_1 \preceq \texttt{<}\bar{b}\texttt{>}\ t_2 \Rightarrow \Theta'$	instance
$\Theta; \Delta; \Phi; \Gamma \overset{loc}{\vdash} e \Rightarrow t; \Theta'$	local inference
$\Theta; \Delta; \Phi; \Gamma \vdash e \Rightarrow t; \Theta'$	global inference
$\Theta; \Delta; \Phi; \Gamma \vdash d \Rightarrow \Theta'; \Phi'; \Gamma'$	declaration inference
$\Theta; \Delta; \Phi; \Gamma \vdash p \Rightarrow \Theta'; \Delta'; \Gamma'$	pattern inference

Fig. 5. Judgments used in type inference

Unification $\Theta \vdash t_1 = t_2 \Rightarrow \Theta'$

$$\frac{}{\Theta \vdash t = t \Rightarrow \Theta}\ \textsf{uni:eq} \qquad \frac{X \in \mathrm{dom}(\Theta) \qquad \Theta \vdash \Theta(X) = t \Rightarrow \Theta'}{\Theta \vdash X = t \Rightarrow \Theta'}\ \textsf{uni:uvar1}$$

$$\frac{X \notin \mathrm{dom}(\Theta) \qquad X \notin \mathrm{FUV}(t)}{\Theta \vdash X = t \Rightarrow \Theta, t/X}\ \textsf{uni:uvar2} \qquad \frac{\Theta \vdash X = t \Rightarrow \Theta'}{\Theta \vdash t = X \Rightarrow \Theta'}\ \textsf{uni:uvar3}$$

$$\frac{\Theta \vdash t_1 = t_3 \Rightarrow \Theta' \qquad \Theta' \vdash t_2 = t_4 \Rightarrow \Theta''}{\Theta \vdash t_1\ \texttt{->}\ t_2 = t_3\ \texttt{->}\ t_4 \Rightarrow \Theta''}\ \textsf{uni:arr}$$

$$\frac{\Theta \vdash \texttt{<}\bar{a}\texttt{>}\ t_1\ \texttt{->}\ t_2 \preceq \texttt{<}\bar{b}\texttt{>}\ t_3\ \texttt{->}\ t_4 \Rightarrow \Theta' \qquad \Theta' \vdash \texttt{<}\bar{b}\texttt{>}\ t_3\ \texttt{->}\ t_4 \preceq \texttt{<}\bar{a}\texttt{>}\ t_1\ \texttt{->}\ t_2 \Rightarrow \Theta''}{\Theta \vdash \textbf{pc}\ \texttt{(<}\bar{a}\texttt{>}\ t_1\ \texttt{\~{}>}\ t_2\texttt{)} = \textbf{pc}\ \texttt{(<}\bar{b}\texttt{>}\ t_3\ \texttt{\~{}>}\ t_4\texttt{)} \Rightarrow \Theta''}\ \textsf{uni:pc}$$

Instance $\Theta \vdash \texttt{<}\bar{a}\texttt{>}\ t_1 \preceq \texttt{<}\bar{b}\texttt{>}\ t_2 \Rightarrow \Theta'$

$$\frac{\begin{array}{c}\overline{X}\ \text{fresh} \qquad \Theta \vdash t_1[\overline{X}/\bar{a}] = t_2 \Rightarrow \Theta' \\ \bar{b} \# \mathrm{FTV}(\Theta'(\texttt{<}\bar{a}\texttt{>}\ t_1)) \qquad \bar{c}\ \text{fresh} \qquad \bar{b} \# \mathrm{FTV}(\Theta'(\texttt{<}\bar{c}\texttt{>}\ t_2[\bar{c}/\bar{b}]))\end{array}}{\Theta \vdash \texttt{<}\bar{a}\texttt{>}\ t_1 \preceq \texttt{<}\bar{b}\texttt{>}\ t_2 \Rightarrow \Theta'\restriction_{\bar{b}}}\ \textsf{iinst}$$

$$\text{where } \Theta\restriction_{\bar{b}} = \{t/X \mid \forall (t/X) \in \Theta, \bar{b} \# \mathrm{FTV}(t)\}$$

Fig. 6. The unification and instance algorithms

Pointcut projection

$$\begin{aligned}
\pi(\mathbf{before}, (\texttt{<}\bar{a}\texttt{>}\ t_1 \texttt{~>} t_2)) &\triangleq \texttt{<}\bar{a}\texttt{>}\ t_1\\
\pi(\mathbf{after}, (\texttt{<}\bar{a}\texttt{>}\ t_1 \texttt{~>} t_2)) &\triangleq \texttt{<}\bar{a}\texttt{>}\ t_2\\
\pi(\mathbf{around}, (\texttt{<}\bar{a}\texttt{>}\ t_1 \texttt{~>} t_2)) &\triangleq \texttt{<}\bar{a}\texttt{>}\ t_1 \texttt{->} t_2\\
\pi(\mathbf{stk}, (\texttt{<}\bar{a}\texttt{>}\ t_1 \texttt{~>} t_2)) &\triangleq \texttt{<}\bar{a}\texttt{>}\ t_1
\end{aligned}$$

Generalization

$$gen(\Gamma, t) \triangleq \texttt{<}\bar{a}\texttt{>}\ t[\bar{a}/\overline{X}] \quad \text{where } \overline{X} = \mathrm{FUV}(t) - \mathrm{FUV}(\Gamma) \text{ and } \bar{a} \text{ fresh}$$

Context refining substitution

$$\begin{aligned}
\cdot\langle t/a\rangle &\triangleq \cdot\\
\Gamma, x :: s\langle t/a\rangle &\triangleq \Gamma\langle t/a\rangle, x :: s[t/a]\\
\Gamma, x : s\langle t/a\rangle &\triangleq \Gamma\langle t/a\rangle
\end{aligned}$$

Fig. 7. Auxiliary definitions

That is, the substitution Θ is extended to produce a new substitution Θ' so that $\Theta'(t_1) = \Theta'(t_2)$. Furthermore, Θ' is the most general unifier for these monotypes. In this and other judgments, we use the convention that the outputs of the algorithm appear to the right of the $\Rightarrow$ symbol. Note that output substitutions cannot refer to bound type variables. Our unification algorithm is given in Figure 6.

Unification is defined in a mutually recursive fashion with the *instance* algorithm. Whenever two pointcut types are compared by the unification algorithm, it verifies that they are both instances of each other. The relation $\Theta \vdash \texttt{<}\bar{a}\texttt{>}\ t_1 \preceq \texttt{<}\bar{b}\texttt{>}\ t_2 \Rightarrow \Theta'$, also defined in Figure 6, is read as:

> "Given input substitution Θ, polytype $\texttt{<}\bar{a}\texttt{>}\ t_1$ can be shown to at least as general as polytype $\texttt{<}\bar{b}\texttt{>}\ t_2$, by producing an extended substitution Θ'."

By "at least as general", we mean that there exists a substitution Θ' for some of the quantified variables $\bar{a}$ in t_1 such that $\Theta'(t_1)$ will be equal to t_2. This definition is the same as in DM type inference.[9] We restrict the substitution produced by the iinst rule, as indicated with $\Theta'\lceil_{\bar{b}}$, to enforce the invariant that variables in the substitution will not escape their scope. This does not affect the correctness of the algorithm as any unification variables with $\bar{b}$ in their range will be effectively "dead" at this point.

The rules for inference are presented in Figure 8. Figure 7 presents some useful auxiliary definitions. In the local and global type inference judgments, Θ is an input substitution, Γ the term variable context, Δ the type variable context, and Φ the set of function names currently in scope. The need for Φ is explained in Section 4.3. The local type inference judgment, $\Theta; \Delta; \Phi; \Gamma \overset{loc}{\vdash} e \Rightarrow t; \Theta'$, is read as:

> "Given the input substitution Θ and the contexts Δ,Φ, and Γ, the term e has type t, as specified by the programmer, and produces substitution Θ'."

This judgment holds when either the type of e was annotated in the source text or when e is an expression whose type is "easy" to determine, such as a variable whose

[9] Note that our notation follows the convention of viewing $\texttt{<}\bar{a}\texttt{>}\ t_1$ as a subtype of $\texttt{<}\bar{b}\texttt{>}\ t_2$.

Local rules $\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathtt{e} \Rightarrow \mathtt{t}; \Theta'$

$$\frac{\Delta \vdash \mathtt{t}_2 \qquad \Theta; \Delta; \Phi; \Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}_1; \Theta' \qquad \Theta' \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta''}{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} (\mathtt{e}\texttt{:}\mathtt{t}_2) \Rightarrow \mathtt{t}_2; \Theta''}\ \textsf{litm:cnv}$$

$$\frac{\mathtt{x} :: \mathtt{t} \in \Gamma}{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathtt{x} \Rightarrow \mathtt{t}; \Theta}\ \textsf{litm:var} \qquad \frac{}{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathtt{()} \Rightarrow \mathbf{Unit}; \Theta}\ \textsf{litm:unit}$$

$$\frac{}{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathtt{c} \Rightarrow \mathbf{String}; \Theta}\ \textsf{litm:string}$$

$$\frac{}{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathbf{any} \Rightarrow \mathbf{pc}\ \mathtt{(<ab>\ a\ \text{\textasciitilde}>\ b)}; \Theta}\ \textsf{litm:any}$$

$$\frac{\Delta, \bar{\mathtt{a}} \vdash \mathtt{t}_1 \qquad \Delta, \bar{\mathtt{a}} \vdash \mathtt{t}_2 \qquad \left(\begin{array}{l} \forall \mathtt{i}\ \ \mathtt{f}_\mathtt{i} \in \Phi \\ \qquad \Gamma(\mathtt{f}_\mathtt{i}) = \mathtt{<}\bar{\mathtt{b}}\mathtt{>}\ \mathtt{t}_{1,\mathtt{i}}\ \mathtt{->}\ \mathtt{t}_{2,\mathtt{i}} \\ \qquad \Theta_{\mathtt{i}-1} \vdash \mathtt{<}\bar{\mathtt{a}}\mathtt{>}\ \mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2 \preceq \mathtt{<}\bar{\mathtt{b}}\mathtt{>}\ \mathtt{t}_{1,\mathtt{i}}\ \mathtt{->}\ \mathtt{t}_{2,\mathtt{i}} \Rightarrow \Theta_\mathtt{i} \end{array}\right)}{\Theta_0; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathtt{\#}\bar{\mathtt{f}}\mathtt{:(<}\bar{\mathtt{a}}\mathtt{>}\ \mathtt{t}_1\ \mathtt{\text{\textasciitilde}>}\ \mathtt{t}_2\mathtt{)\#} \Rightarrow \mathbf{pc}\ \mathtt{(<}\bar{\mathtt{a}}\mathtt{>}\ \mathtt{t}_1\ \mathtt{\text{\textasciitilde}>}\ \mathtt{t}_2\mathtt{)}; \Theta_n}\ \textsf{litm:set-ann}$$

$$\frac{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{d} \Rightarrow \Theta'; \Phi'; \Gamma' \qquad \Theta'; \Delta; \Phi, \Phi'; \Gamma, \Gamma' \stackrel{\mathrm{loc}}{\vdash} \mathtt{e} \Rightarrow \mathtt{t}; \Theta''}{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathbf{let}\ \mathtt{d}\ \mathbf{in}\ \mathtt{e} \Rightarrow \mathtt{t}; \Theta''}\ \textsf{litm:let}$$

Global rules $\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}; \Theta'$

$$\frac{\Theta; \Delta; \Phi; \Gamma \stackrel{\mathrm{loc}}{\vdash} \mathtt{e} \Rightarrow \mathtt{t}; \Theta'}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}; \Theta'}\ \textsf{gitm:cnv} \qquad \frac{\Gamma(\mathtt{x}) = \mathtt{<}\bar{\mathtt{a}}\mathtt{>}\ \mathtt{t} \qquad \overline{\mathrm{X}}\ \text{fresh}}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{x} \Rightarrow \mathtt{t}[\overline{\mathrm{X}}/\bar{\mathtt{a}}]; \Theta}\ \textsf{gitm:var}$$

$$\frac{\begin{array}{c}\Theta_1; \Delta; \Phi; \Gamma \vdash \mathtt{e}_1 \Rightarrow \mathtt{t}_1; \Theta_2 \\ \Theta_2; \Delta; \Phi; \Gamma \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2; \Theta_3 \qquad \mathrm{X}\ \text{fresh} \qquad \Theta_3 \vdash \mathtt{t}_1 = \mathtt{t}_2\ \mathtt{->}\ \mathrm{X} \Rightarrow \Theta_4\end{array}}{\Theta_1; \Delta; \Phi; \Gamma \vdash \mathtt{e}_1\,\mathtt{e}_2 \Rightarrow \mathrm{X}; \Theta_4}\ \textsf{gitm:app}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{e} \Rightarrow \mathbf{Stack}; \Theta_0 \quad \Theta_0; \Delta; \Phi; \Gamma \vdash \mathtt{e}' \Rightarrow \mathtt{t}; \Theta_0'' \\ \left(\begin{array}{ll} \forall \mathtt{i}\ \ \Theta_{\mathtt{i}-1}''; \Delta; \Phi; \Gamma \vdash \mathtt{p}_\mathtt{i} \Rightarrow \Theta_\mathtt{i}; \Delta_\mathtt{i}; \Gamma_\mathtt{i} & \Theta_\mathtt{i}; \Delta, \Delta_\mathtt{i}; \Phi; \Gamma, \Gamma_\mathtt{i} \vdash \mathtt{e}_\mathtt{i} \Rightarrow \mathtt{t}_\mathtt{i}; \Theta_\mathtt{i}' \\ \quad\ \Theta_\mathtt{i}' \vdash \mathtt{t}_\mathtt{i} = \mathtt{t} \Rightarrow \Theta_\mathtt{i}'' & \mathrm{FTV}(\mathtt{t}_\mathtt{i}) \mathbin{\#} \Delta_\mathtt{i} \end{array}\right)\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \mathbf{stkcase}\ \mathtt{e}\ \mathtt{(}\overline{\mathtt{p=>e}}\ \mathtt{|_=>}\ \mathtt{e}'\mathtt{)} \Rightarrow \mathtt{t}; \Theta_n''}\ \textsf{gitm:scase}$$

$$\frac{\begin{array}{c}\mathtt{a} \in \Delta \qquad \Delta \vdash \mathtt{t} \qquad \Theta; \Delta; \Phi; \Gamma \vdash \mathtt{e} \Rightarrow \mathtt{t}'; \Theta' \qquad \Theta' \vdash \mathtt{t}' = \mathtt{t} \Rightarrow \Theta_0 \\ \left(\begin{array}{ll} \forall \mathtt{i}\ \ \Delta_\mathtt{i} = \mathrm{FTV}(\mathtt{t}_\mathtt{i}) - \Delta & \Theta_{\mathtt{i}-1}; \Delta, \Delta_\mathtt{i}; \Phi; \Gamma\langle \mathtt{t}_\mathtt{i}/\mathtt{a}\rangle \vdash \mathtt{e}_\mathtt{i}[\mathtt{t}_\mathtt{i}/\mathtt{a}] \Rightarrow \mathtt{t}_\mathtt{i}'; \Theta_\mathtt{i}' \\ \quad\ \mathtt{a} \notin \mathrm{FTV}(\mathtt{t}_\mathtt{i}) & \Theta_\mathtt{i}' \vdash \mathtt{t}_\mathtt{i}' = \mathtt{t}[\mathtt{t}_\mathtt{i}/\mathtt{a}] \Rightarrow \Theta_\mathtt{i} \end{array}\right)\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \mathbf{typecase}\mathtt{<t>}\ \mathtt{a}\ \mathtt{(}\overline{\mathtt{t=>e}}\ \mathtt{|_=>}\ \mathtt{e)} \Rightarrow \mathtt{t}; \Theta_n}\ \textsf{gitm:tcase}$$

$$\frac{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{d} \Rightarrow \Theta'; \Phi'; \Gamma' \qquad \Theta'; \Delta; \Phi, \Phi'; \Gamma, \Gamma' \vdash \mathtt{e} \Rightarrow \mathtt{t}; \Theta''}{\Theta; \Delta; \Phi; \Gamma \vdash \mathbf{let}\ \mathtt{d}\ \mathbf{in}\ \mathtt{e} \Rightarrow \mathtt{t}; \Theta''}\ \textsf{gitm:let}$$

Fig. 8. Type inference for expressions

(monomorphic) type was annotated or certain constants. To propagate the type annotation on variables, the context, Γ, contains two different assertions depending on whether types are local ($\mathsf{x :: s}$) or perhaps inferred via unification ($\mathsf{x : s}$). We use the notation $\Gamma(\mathsf{x}) = \mathsf{s}$ to refer to either $\mathsf{x : s} \in \Gamma$ or $\mathsf{x :: s} \in \Gamma$.

The global type inference judgment, $\Theta; \Delta; \Phi; \Gamma \vdash \mathsf{e} \Rightarrow \mathtt{t}; \Theta'$, is read as:

> "Given the input substitution Θ and the contexts Δ,Φ, and Γ, the term e has type $\mathtt{t}$ and produces substitution Θ', possibly requiring unification to determine $\mathtt{t}$."

Local and global inference interact via the following two rules

$$\frac{\Delta \vdash \mathtt{t}_2 \qquad \Theta; \Delta; \Phi; \Gamma \vdash \mathsf{e} \Rightarrow \mathtt{t}_1; \Theta' \qquad \Theta' \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta''}{\Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} (\mathtt{e:t}_2) \Rightarrow \mathtt{t}_2; \Theta''} \ \textsf{litm:cnv}$$

$$\frac{\Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathsf{e} \Rightarrow \mathtt{t}; \Theta'}{\Theta; \Delta; \Phi; \Gamma \vdash \mathsf{e} \Rightarrow \mathtt{t}; \Theta'} \ \textsf{gitm:cnv}$$

The Rule litm:cnv says that if an expression can be inferred to have type $\mathtt{t}_1$ via global inference, and the programmer annotates the expression with type $\mathtt{t}_2$, and $\mathtt{t}_1$ and $\mathtt{t}_2$ are unifiable, the expression has the local type $\mathtt{t}_2$. The Rule gitm:cnv says that any expression that can be inferred to have a local type can also be inferred to have a global type.

4.3 First-class polymorphic pointcuts

In Section 4.1, we examined the difficulties that pointcut types cause for type inference. Pointcut terms, in particular pointcuts sets, are another tricky part of our type system. Only functions named by a **fun** declaration may be used as part of a pointcut set. To ensure this constraint, the Φ component of the typing judgments keeps track of which function names are currently in scope. All variables in a pointcut set must be a member of Φ. As we discussed in Section 2.5, this was just a design decision, and the inference algorithm and translation described in Section 6 could be redesigned to accommodate allowing any variable of function type to be used as part of a pointcut set.

The Rule litm:set-ann, in Figure 8, requires the type of each function in the set to be at most as polymorphic as the pointcut annotation $\mathtt{(<\bar{a}>\ t_1 \sim> t_2)}$ on the set. To simplify the description of inference for idealized AspectML, a pointcut set must always be annotated, and will therefore always have a local type. In our implementation if the pointcut set is unannotated, that is, written as $\mathtt{\#\bar{f}\#}$, anti-unification may be used to infer the missing annotation. However, the types of all the functions in the set must be rigid for anti-unification to compute the least general pointcut type for the set.

In the implementation, a function may not have a rigid type if it is not closed and the type of the free variable is unconstrained. For example, in the following

fragment **g** in the unannotated set **#g#** does not have a rigid type.

```
fun f x = let
            fun g y = x
          in
            #g# (* g's type is not rigid -
                   expression will not type check *)
          end
```

Here the type of **g** is not rigid because some unification variable X was introduced for the type of **x**. However, because X occurs in the environment, it will still be present in **g**'s type after it is generalized, per the definition in Figure 7. Our anti-unification algorithm is only complete (computes the least general type) for types that are rigid; it will still produce a sound result (a more general type) in the presence of unification variables. Consequently, we make this restriction so that a user can be sure that inferred pointcuts have the least general type possible.

4.4 Runtime type analysis

There are two difficulties with combining type inference with run-time type analysis. First, the return type of a **typecase** expression is difficult to determine from the types of the branches. We solve this first problem by simply requiring an annotation for the result type.

$$\frac{\begin{array}{cccc} a \in \Delta & \Delta \vdash t & \Theta;\Delta;\Phi;\Gamma \vdash e \Rightarrow t';\Theta' & \Theta' \vdash t' = t \Rightarrow \Theta_0 \end{array} \quad \left(\begin{array}{cc} \forall i\ \ \Delta_i = \mathrm{FTV}(t_i) - \Delta & \Theta_{i-1};\Delta,\Delta_i;\Phi;\Gamma\langle t_i/a\rangle \vdash e_i[t_i/a] \Rightarrow t_i';\Theta_i' \\ a \notin \mathrm{FTV}(t_i) & \Theta_i' \vdash t_i' = t[t_i/a] \Rightarrow \Theta_i \end{array}\right)}{\Theta;\Delta;\Phi;\Gamma \vdash \texttt{typecase<t> a (}\overline{\texttt{t=>e}}\texttt{ |_=> e)} \Rightarrow t;\Theta_n}\ \text{gitm:tcase}$$

As the rule above shows, if the expression should be of type t then a branch for type t_i may be of type $t[t_i/a]$. This substitution is sound because if the branch is executed, then the type a is the same as the type t_i. When type checking each branch, the context will also change. In Figure 7 we define context refining substitution, $\Gamma\langle t_i/a\rangle$. A type t_i is substituted for the variable a *only* in local assumptions $x :: s$. Furthermore, global assumptions are eliminated from the context.

The reason for these restrictions is two-fold. Firstly, we must not allow refinement in parts of the context that contain unification variables because, even with the return type annotation on **typecase**, there are some expressions with no principal type. For example, in the following code fragment:

```
fun h <a>(x:a) = fn y => typecase<Int> a
                          of Int => x + y + 1
                           | _   => 2
```

we can assign the types **<a> a -> a -> Int** or **<a> a -> Int -> Int** to **h**, and neither is more general than the other. The problem is that it is equally valid for **y** to have type **Int** or to have a type that refines to **Int**. By requiring the user to specify the type of **y** for refinement to apply, we eliminate this confusion.

Secondly, even if we were to allow global types in the context, but did not refine them, we run into an inconsistency in the behavior of type inference.

```
fun h <a b>(x:a -> Int, y:b) = (fn z => typecase<Int> a
                                                  of b => x z
                                                   | _ => 0)
                                y
```

Here the **typecase** expression chooses **b** to be the canonical element of the equivalence class {**a**, **b**}, so inside the **typecase** branch, **x** has type **b -> Int**. The variable **z** has some unification variable X as its type, and which unifies **b** due to the application. Outside the **typecase** branch **y** has type **b** and applying it to a function of type **b -> Int** is well typed. But consider the exact same program, but with **a** swapped for **b** in the **typecase** expression.

```
fun h <a b>(x:a -> Int, y:b) = (fn z => typecase<Int> b
                                                  of a => x z
                                                   | _ => 0)
                                y
```

Here the **typecase** expression chooses **a** to be the canonical element of the equivalence class {**a**, **b**}, so inside the **typecase** branch, **x** has type **a -> Int**. The variable **z** has some unification variable X as its type, and which unifies **a** due to the application. Outside the **typecase** branch **y** has type **b** and applying it to a function of type **a -> Int** is ill-typed.

Therefore, to make it easier for a programmer to reason about the behavior of a program, we conservatively eliminate all term variables with global types from the context. If this rules out a user's program, they can simply add additional type annotations to whatever term variables they need inside of a **typecase** branch.

These issues have arisen before in type inference systems for Generalized Algebraic Datatypes (also called Guarded Recursive Datatypes or GADTs) [Peyton Jones et al. 2006; Simonet and Pottier 2005; Sulzmann et al. 2006]. In these systems, as in ours, type refinements prevent some unannotated terms from having principal types.

At present, we believe we might be able to lessen the need to annotate **typecase** expressions through the use of other type annotations the user does supply. This would involve adopting local type inference techniques, such as *bidirectional type inference* [Pierce and Turner 1998; Peyton Jones et al. 2007], *boxy types* [Vytiniotis et al. 2006], or *shape inference* [Pottier and Régis-Gianas 2006].

A final subtle point about the type inference rule for **typecase** is each type pattern t_i is allowed to contain free variables bound in the context Δ. This has a number of uses, including writing a concise "type-safe cast".

```
fun cast <a b>(x:a):Option b = typecase<Option b> a
                                 of b => Some x
                                  | _ => None
```

However, we do not allow a type variable being analyzed by **typecase** to appear in type patterns. This is so that we can ensure that the refined return type $t[t_i/a]$ will not contain the variable. Furthermore, we do not believe there is any way to use the

Declarations $\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{d} \Rightarrow \Theta'; \Phi'; \Gamma'$

$$\frac{\begin{array}{c}\Theta; \Delta, \bar{a}; \Phi; \Gamma, \mathtt{f} :: \mathtt{t}_1 \ \texttt{->}\ \mathtt{t}_2, \mathtt{x} :: \mathtt{t}_1 \vdash \mathtt{e}_1 \Rightarrow \mathtt{t}_3; \Theta' \\ \Theta' \vdash \mathtt{t}_2 = \mathtt{t}_3 \Rightarrow \Theta'' \qquad \mathtt{s} = \texttt{<}\bar{a}\texttt{>}\ \mathtt{t}_1 \to \mathtt{t}_2 \qquad \bar{a} \notin \Theta''(\Gamma)\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{fun}\ \ \mathtt{f}\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:t}_1\texttt{):t}_2\ \texttt{=}\ \mathtt{e}_1 \Rightarrow \Theta''; \cdot, \mathtt{f}; \cdot, \mathtt{f} :: \mathtt{s}} \ \text{id:fun-ann}$$

$$\frac{\begin{array}{c}\mathsf{X}, \mathsf{Y}\ \text{fresh} \qquad \Theta; \Delta; \Phi; \Gamma, \mathtt{f} : \mathsf{X}\ \texttt{->}\ \mathsf{Y}, \mathtt{x} : \mathsf{X} \vdash \mathtt{e}_1 \Rightarrow \mathtt{t}; \Theta' \\ \Theta' \vdash \mathsf{Y} = \mathtt{t} \Rightarrow \Theta'' \qquad \mathtt{s} = gen(\Theta''(\Gamma), \Theta''(\mathsf{X}\ \texttt{->}\ \mathsf{Y}))\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{fun}\ \ \mathtt{f}\ \mathtt{x}\ \texttt{=}\ \mathtt{e}_1 \Rightarrow \Theta''; \cdot, \mathtt{f}; \cdot, \mathtt{f} : \mathtt{s}} \ \text{id:fun}$$

$$\frac{\begin{array}{c}\mathtt{tm} \in \{\textbf{before}, \textbf{after}\} \qquad \Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e}_1 \Rightarrow \textbf{pc}\ \mathtt{pt}; \Theta' \qquad \pi(\mathtt{tm}, \mathtt{pt}) = \texttt{<}\bar{a}\texttt{>}\ \mathtt{t}_1 \\ \Theta'; \Delta, \bar{a}; \Phi; \Gamma, \mathtt{x} :: \mathtt{t}_1, \mathtt{y} :: \textbf{Stack}, \mathtt{z} :: \textbf{String} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2; \Theta'' \qquad \Theta'' \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{advice}\ \mathtt{tm}\ \texttt{(|e}_1\texttt{|) (x,y,z) =}\ \mathtt{e}_2 \Rightarrow \Theta'''; \cdot; \cdot} \ \text{id:advice-befaft}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e}_1 \Rightarrow \textbf{pc}\ \mathtt{pt}; \Theta' \qquad \pi(\textbf{around}, \mathtt{pt}) = \texttt{<}\bar{a}\texttt{>}\ \mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \\ \Theta'; \Delta, \bar{a}; \Phi; \Gamma, \mathtt{x} :: \mathtt{t}_1, \mathtt{y} :: \textbf{Stack}, \mathtt{z} :: \textbf{String}, \mathtt{proceed} :: \mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_3; \Theta'' \\ \Theta'' \vdash \mathtt{t}_2 = \mathtt{t}_3 \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{advice around}\ \texttt{(|e}_1\texttt{|) (x,y,z) =}\ \mathtt{e}_2 \Rightarrow \Theta'''; \cdot; \cdot} \ \text{id:advice-aro}$$

$$\frac{\begin{array}{c}\mathtt{tm} \in \{\textbf{before}, \textbf{after}\} \qquad \Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e}_1 \Rightarrow \textbf{pc}\ \mathtt{pt}; \Theta' \qquad \pi(\mathtt{tm}, \mathtt{pt}) = \texttt{<}\bar{a}\texttt{>}\ \mathtt{t}_1 \\ \Theta'; \Delta, \bar{a}; \Phi; \Gamma, \mathtt{x} :: \mathtt{t}_1, \mathtt{y} :: \textbf{Stack}, \mathtt{z} :: \textbf{String} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2; \Theta'' \\ \Theta'' \vdash \mathtt{t}_2 = \mathtt{t}_1 \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{advice}\ \mathtt{tm}\ \texttt{(|e}_1\texttt{|)}\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:t}_1\texttt{,y,z) =}\ \mathtt{e}_2 \Rightarrow \Theta'''; \cdot; \cdot} \ \text{id:advice-ann-befaft}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e}_1 \Rightarrow \textbf{pc}\ \mathtt{pt}; \Theta' \qquad \pi(\textbf{around}, \mathtt{pt}) = \texttt{<}\bar{a}\texttt{>}\ \mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \\ \Theta'; \Delta, \bar{a}; \Phi; \Gamma, \mathtt{x} :: \mathtt{t}_1, \mathtt{y} :: \textbf{Stack}, \mathtt{z} :: \textbf{String}, \mathtt{proceed} :: \mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_3; \Theta'' \\ \Theta'' \vdash \mathtt{t}_2 = \mathtt{t}_3 \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{advice around}\ \texttt{(|e}_1\texttt{|)}\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:t}_1\texttt{,y,z):t}_2\ \texttt{=}\ \mathtt{e}_2 \Rightarrow \Theta'''; \cdot; \cdot} \ \text{id:advice-ann-aro}$$

$$\frac{\begin{array}{c}\mathtt{tm} \in \{\textbf{before}, \textbf{after}\} \qquad \Delta' = \mathrm{FTV}(\mathtt{t}_1) - \Delta \qquad \Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e}_1 \Rightarrow \textbf{pc}\ \mathtt{pt}; \Theta' \\ \Theta'; \Delta, \Delta'; \Phi; \Gamma, \mathtt{x} :: \mathtt{t}_1, \mathtt{y} :: \textbf{Stack}, \mathtt{z} :: \textbf{String} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2; \Theta'' \\ \Theta'' \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{case-advice}\ \mathtt{tm}\ \texttt{(|e}_1\texttt{|) (x:t}_1\texttt{,y,z) =}\ \mathtt{e}_2 \Rightarrow \Theta'''; \cdot; \cdot} \ \text{id:cadvice-befaft}$$

$$\frac{\begin{array}{c}\Delta' = \mathrm{FTV}(\mathtt{t}_1) \cup \mathrm{FTV}(\mathtt{t}_2) - \Delta \qquad \Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e}_1 \Rightarrow \textbf{pc}\ \mathtt{pt}; \Theta' \\ \Theta'; \Delta, \Delta'; \Phi; \Gamma, \mathtt{x} :: \mathtt{t}_1, \mathtt{y} :: \textbf{Stack}, \mathtt{z} :: \textbf{String}, \mathtt{proceed} :: \mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_3; \Theta'' \\ \Theta'' \vdash \mathtt{t}_2 = \mathtt{t}_3 \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \textbf{case-advice around}\ \texttt{(|e}_1\texttt{|) (x:t}_1\texttt{,y,z):t}_2\ \texttt{=}\ \mathtt{e}_2 \Rightarrow \Theta'''; \cdot; \cdot} \ \text{id:cadvice-aro}$$

Fig. 9. Type inference for declarations

analyzed type variable in a type pattern that is not trivial or vacuous. Consider,

```
typecase<...> a
  of a       => ...  (* Fine, but trivial -- binds nothing *)
   | (a -> a) => ...  (* Impossible, because we do not have
                         equi-recursive types *)
```

4.5 Declarations

The inference judgment for declarations, $\Theta; \Delta; \Phi; \Gamma \vdash \mathsf{d} \Rightarrow \Theta'; \Phi'; \Gamma'$, defined in Figure 9, is read as:

> "Given input substitution Θ, the contexts Δ, Φ, and Γ, the declaration `d` is well-formed and produces context extensions Φ' and Γ' and a new substitution Θ'."

Note that Φ' and Γ' are only additions to be appended to Φ and Γ, and not entirely new contexts to be used, as Θ' is a new substitution to be threaded through the remainder of the derivation.

As alluded to in Section 4.1, the typing rules for advice declarations (Rules id:advice-befaft, id:advice-aro, id:advice-ann-befaft, and id:advice-ann-aro in Figure 9) state that the type of a pointcut must be determinable using the local type inference judgment. That way, the inference algorithm need not use higher-order unification to determine the type **pc** `pt`. Note that when the body of the advice is checked, the parameters are added to the context with known types, even though they need not be annotated by the user. In our rules, we use the notation $\pi(\mathtt{tm}, \mathtt{pt})$ to indicate projecting the appropriate polytype from the pointcut type. If `tm` is **before**, the first component will be projected; if it is **after**, the second will be projected. If `tm` is **around**, both components are projected. There is also a special trigger time, **stk**, used only by the type inference algorithm, that is essentially equivalent to **before**. This notation is defined in Figure 7.

The typing rules for **case-advice** (Rules id:cadvice-befaft and id:cadvice-aro in Figure 9) are similar to that for advice. The **case-advice** declaration requires a type annotation on `x`, the first parameter to the advice, for **before** and **after case-advice**, and on both `x` and the result for **around case-advice**. These annotations drive the underlying run-time type analysis. Note that, unlike regular **advice**, the programmer is not required to indicate the binding type variables – they can be determined from the type annotations.

4.6 Stack and frame patterns

The last syntactic category in idealized AspectML contains stack and frame patterns. Figure 10 describes the type inference algorithm for patterns for the `stkcase` expression. The judgment $\Theta; \Delta; \Phi; \Gamma \vdash \mathsf{p} \Rightarrow \Theta'; \Delta'; \Gamma'$ is read as:

> "Given the input substitution Θ and the contexts Δ,Φ, and Γ, the pattern `p` produces context extensions Δ' and Γ' and a new substitution Θ'."

As with advice declarations, the typing rules for pointcut patterns (Rules ipat:cons and ipat:cons-ann) require that the type of a pointcut must be determinable using the local type inference judgment. Additionally, similar to the declaration judgments, Δ' and Γ' are only meant to be appended to Δ and Γ and not to replace them.

4.7 A declarative specification

Some users of ML rely on the *declarative* nature of the HM type system, which elides the uses of unification [Milner 1978]. In Figures 11, 12, 13, and 14, we develop a similar declarative specification for our type system.

Patterns $\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{p} \Rightarrow \Theta'; \Delta'; \Gamma'$

$$\frac{}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{[]} \Rightarrow \Theta; \cdot; \cdot}\ \text{ipat:nil} \qquad \frac{}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{x} \Rightarrow \Theta; \cdot; \cdot, \mathtt{x} :: \mathbf{Stack}}\ \text{ipat:var}$$

$$\frac{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{p} \Rightarrow \Theta'; \Delta'; \Gamma'}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{_::p} \Rightarrow \Theta'; \Delta'; \Gamma'}\ \text{ipat:wild}$$

$$\frac{\Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e} \Rightarrow \mathbf{pc}\ \mathtt{pt}; \Theta' \qquad \pi(\mathbf{stk}, \mathtt{pt}) = \mathtt{<\bar{a}>\ t} \qquad \Theta'; \Delta; \Phi; \Gamma \vdash \mathtt{p} \Rightarrow \Theta''; \Delta'; \Gamma'}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{(|e|)(x,z)::p} \Rightarrow \Theta''; \Delta', \bar{\mathtt{a}}; \Gamma', \mathtt{x{:}t}, \mathtt{z}{:}\mathbf{String}}\ \text{ipat:cons}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathtt{e} \Rightarrow \mathbf{pc}\ \mathtt{pt}; \Theta' \\ \pi(\mathbf{stk}, \mathtt{pt}) = \mathtt{<\bar{a}>\ t_1} \qquad \Theta'; \Delta; \Phi; \Gamma \vdash \mathtt{p} \Rightarrow \Theta''; \Delta'; \Gamma' \qquad \Theta'' \vdash \mathtt{t_1} = \mathtt{t_2} \Rightarrow \Theta'''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \mathtt{(|e|)<\bar{a}>(x{:}t_2,z)::p} \Rightarrow \Theta'''; \Delta', \bar{\mathtt{a}}; \Gamma', \mathtt{x{:}t_2}, \mathtt{z}{:}\mathbf{String}}\ \text{ipat:cons-ann}$$

Fig. 10. Type inference for patterns

Types $\Delta \vdash \mathtt{t}$

$$\frac{\Delta, \bar{\mathtt{a}} \vdash \mathtt{t}}{\Delta \vdash \mathtt{<\bar{a}>\ t}}\ \text{wfstp:all} \qquad \frac{\mathtt{a} \in \Delta}{\Delta \vdash \mathtt{a}}\ \text{wfstp:var} \qquad \frac{}{\Delta \vdash \mathsf{X}}\ \text{wfstp:unif}$$

$$\frac{}{\Delta \vdash \mathbf{Unit}}\ \text{wfstp:unit} \qquad \frac{}{\Delta \vdash \mathbf{String}}\ \text{wfstp:string} \qquad \frac{}{\Delta \vdash \mathbf{Stack}}\ \text{wfstp:stack}$$

$$\frac{\Delta \vdash \mathtt{t_1} \qquad \Delta \vdash \mathtt{t_2}}{\Delta \vdash \mathtt{t_1\ {->}\ t_2}}\ \text{wfstp:arr} \qquad \frac{\Delta, \bar{\mathtt{a}} \vdash \mathtt{t_1} \qquad \Delta, \bar{\mathtt{a}} \vdash \mathtt{t_2}}{\Delta \vdash \mathbf{pc}\ \mathtt{(<\bar{a}>\ t_1\ {\sim>}\ t_2)}}\ \text{wfstp:pc}$$

Type equality $\Delta \vdash \mathtt{t_1} = \mathtt{t_2}$

$$\frac{\Delta \vdash \mathtt{t}}{\Delta \vdash \mathtt{t} = \mathtt{t}}\ \text{eq:refl} \qquad \frac{\Delta \vdash \mathtt{t_1} = \mathtt{t_3} \qquad \Delta \vdash \mathtt{t_2} = \mathtt{t_4}}{\Delta \vdash \mathtt{t_1\ {->}\ t_2} = \mathtt{t_3\ {->}\ t_4}}\ \text{eq:arr}$$

$$\frac{\Delta \vdash \mathtt{<\bar{a}>\ t_1 t_2} \preceq \mathtt{<\bar{b}>\ t_3 t_4} \qquad \Delta \vdash \mathtt{<\bar{b}>\ t_3 t_4} \preceq \mathtt{<\bar{a}>\ t_1 t_2}}{\Delta \vdash \mathbf{pc}\ \mathtt{(<\bar{a}>\ t_1\ {\sim>}\ t_2)} = \mathbf{pc}\ \mathtt{(<\bar{b}>\ t_3\ {\sim>}\ t_4)}}\ \text{eq:pc}$$

Instance $\Delta \vdash \mathtt{<\bar{a}>\ t_1} \preceq \mathtt{<\bar{b}>\ t_2}$

$$\frac{\Delta, \bar{\mathtt{a}} \vdash \mathtt{t_1} \qquad \Delta, \bar{\mathtt{b}} \vdash \mathtt{t_2} \qquad \Delta, \bar{\mathtt{b}} \vdash \mathtt{t_1}\,[\bar{\mathtt{t}}/\bar{\mathtt{a}}] = \mathtt{t_2}}{\Delta \vdash \mathtt{<\bar{a}>\ t_1} \preceq \mathtt{<\bar{b}>\ t_2}}\ \text{sinst}$$

Fig. 11. Declarative semantics for types

Local Expressions $\Delta; \Phi; \Gamma \overset{loc}{\vdash} \texttt{e : t}$

$$\frac{\Delta \vdash \texttt{t} \qquad \Delta; \Phi; \Gamma \vdash \texttt{e : t}}{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \texttt{(e:t) : t}}\ \text{ltm:cnv} \qquad \frac{\texttt{x :: t} \in \Gamma}{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \texttt{x : t}}\ \text{ltm:var}$$

$$\frac{}{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \texttt{() : }\mathbf{Unit}}\ \text{ltm:unit} \qquad \frac{}{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \mathbf{any : pc \ (<ab> \ a \sim> b)}}\ \text{ltm:any}$$

$$\frac{\texttt{f}_i \in \Phi \qquad \Gamma(\texttt{f}_i) = \texttt{<}\overline{\texttt{b}}\texttt{> t}_{1,i} \texttt{ -> t}_{2,i} \qquad \begin{array}{c} \Delta, \overline{\texttt{a}} \vdash \texttt{t}_1 \qquad \Delta, \overline{\texttt{a}} \vdash \texttt{t}_2 \qquad \forall i \\ \Delta \vdash \texttt{<}\overline{\texttt{a}}\texttt{> t}_1 \texttt{ -> t}_2 \preceq \texttt{<}\overline{\texttt{b}}\texttt{> t}_{1,i} \texttt{ -> t}_{2,i} \end{array}}{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \mathbf{\#}\overline{\texttt{f}}\mathbf{:(<}\overline{\texttt{a}}\mathbf{>}\ \texttt{t}_1 \mathbf{\sim>} \texttt{t}_2\mathbf{)\#} : \mathbf{pc\ (<}\overline{\texttt{a}}\mathbf{>}\ \texttt{t}_1 \mathbf{\sim>} \texttt{t}_2\mathbf{)}}\ \text{ltm:set-ann}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \texttt{d} \dashv \Phi'; \Gamma' \qquad \Delta; \Phi, \Phi'; \Gamma, \Gamma' \overset{loc}{\vdash} \texttt{e : t}}{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \mathbf{let}\ \texttt{d}\ \mathbf{in}\ \texttt{e : t}}\ \text{ltm:let}$$

Global Expressions $\Delta; \Phi; \Gamma \vdash \texttt{e : t}$

$$\frac{\Delta; \Phi; \Gamma \overset{loc}{\vdash} \texttt{e : t}}{\Delta; \Phi; \Gamma \vdash \texttt{e : t}}\ \text{gtm:cnv} \qquad \frac{\Gamma(\texttt{x}) = \texttt{<}\overline{\texttt{a}}\texttt{> t} \qquad \Delta \vdash \texttt{t}_i}{\Delta; \Phi; \Gamma \vdash \texttt{x : t}[\overline{\texttt{t}}/\overline{\texttt{a}}]}\ \text{gtm:var}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \texttt{e}_1 : \texttt{t}_1 \texttt{ -> t}_2 \qquad \Delta; \Phi; \Gamma \vdash \texttt{e}_2 : \texttt{t}_1}{\Delta; \Phi; \Gamma \vdash \texttt{e}_1\, \texttt{e}_2 : \texttt{t}_2}\ \text{gtm:app}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \texttt{e : }\mathbf{Stack} \qquad \Delta; \Phi; \Gamma \vdash \texttt{e}' : \texttt{t} \qquad \left(\begin{array}{l} \forall i \ \ \Delta; \Phi; \Gamma \vdash \texttt{p}_i \dashv \Delta_i; \Gamma_i \\ \qquad \Delta, \Delta_i; \Phi; \Gamma, \Gamma_i \vdash \texttt{e}_i : \texttt{t} \end{array}\right)}{\Delta; \Phi; \Gamma \vdash \mathbf{stkcase}\ \texttt{e}\ \mathbf{(}\overline{\texttt{p=>e}}\ \mathbf{|_=>}\ \texttt{e}'\mathbf{)} : \texttt{t}}\ \text{gtm:scase}$$

$$\frac{\texttt{a} \in \Delta \qquad \Delta; \Phi; \Gamma \vdash \texttt{e : t} \qquad \left(\begin{array}{l} \forall i \ \ \Delta_i = \mathrm{FTV}(\texttt{t}_i) - \Delta \\ \qquad \texttt{a} \notin \mathrm{FTV}(\texttt{t}_i) \\ \qquad \Delta, \Delta_i; \Phi; \Gamma\langle \texttt{t}_i/\texttt{a}\rangle \vdash \texttt{e}_i[\texttt{t}_i/\texttt{a}] : \texttt{t}[\texttt{t}_i/\texttt{a}] \end{array}\right)}{\Delta; \Phi; \Gamma \vdash \mathbf{typecase<}\texttt{t}\mathbf{>}\ \texttt{a}\ \mathbf{(}\overline{\texttt{t=>e}}\ \mathbf{|_=>}\ \texttt{e}\mathbf{)} : \texttt{t}}\ \text{gtm:tcase}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \texttt{d} \dashv \Phi'; \Gamma' \qquad \Delta; \Phi, \Phi'; \Gamma, \Gamma' \vdash \texttt{e : t}}{\Delta; \Phi; \Gamma \vdash \mathbf{let}\ \texttt{d}\ \mathbf{in}\ \texttt{e : t}}\ \text{gtm:let}$$

Fig. 12. Declarative semantics for expressions

Unfortunately, the rule for annotated pointcuts (litm:set-ann) in Figure 8 has undesirable interactions with the declarative specification of HM-style type inference. This rule uses the function $\texttt{f}_i$ without instantiation, breaking the following property: if $\Delta; \Phi; \Gamma \vdash \texttt{e : t}$ and Γ' is a more general context than Γ, then $\Delta; \Phi; \Gamma' \vdash \texttt{e : t}$. This property does not hold because a more general type for a function $\texttt{f}_i$ may require a more general pointcut type annotation when that function appears in a pointcut set. Because this property fails, our algorithm is not complete with respect to the standard specification of HM-style inference extended with our new terms. The reason is that the algorithm always uses the most general type for let-bound variables, whereas the declarative system is free to use a less general type.

Declarations $\Delta; \Phi; \Gamma \vdash \mathtt{d} \dashv \Phi'; \Gamma'$

$$\frac{\Delta, \bar{a}; \Phi; \Gamma, f :: t_1 \ \texttt{->}\ t_2, x :: t_1 \vdash e_1 : t_2}{\Delta; \Phi; \Gamma \vdash \textbf{fun}\ \ f\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:}t_1\texttt{):}t_2\ \texttt{=}\ e_1 \dashv \cdot, f; \cdot, f :: \texttt{<}\bar{a}\texttt{>}\ t_1 \ \texttt{->}\ t_2}\ \text{wfsd:fun-ann}$$

$$\frac{\Delta, \bar{a} \vdash t_1 \qquad \Delta, \bar{a} \vdash t_2 \qquad \Delta, \bar{a}; \Phi; \Gamma, f : t_1 \ \texttt{->}\ t_2, x : t_1 \vdash e_1 : t_2}{\Delta; \Phi; \Gamma \vdash \textbf{fun}\ \ f\ x\ \texttt{=}\ e_1 \dashv \cdot, f; \cdot, f : \texttt{<}\bar{a}\texttt{>}\ t_1 \ \texttt{->}\ t_2}\ \text{wfsd:fun}$$

$$\frac{\begin{array}{c} tm \in \{\textbf{before}, \textbf{after}\} \qquad \pi(tm, pt) = \texttt{<}\bar{a}\texttt{>}\ t \\ \Delta; \Phi; \Gamma \overset{loc}{\vdash} e_1 : \textbf{pc}\ pt \qquad \Delta, \bar{a}; \Phi; \Gamma, x :: t, y :: \textbf{Stack}, z :: \textbf{String} \vdash e_2 : t \end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{advice}\ tm\ \texttt{(|}e_1\texttt{|)}\ \texttt{(x,y,z)}\ \texttt{=}\ e_2 \dashv \cdot; \cdot}\ \text{wfsd:advice-befaft}$$

$$\frac{\begin{array}{c} \pi(\textbf{around}, pt) = \texttt{<}\bar{a}\texttt{>}\ t_1 \ \texttt{->}\ t_2 \qquad \Delta; \Phi; \Gamma \overset{loc}{\vdash} e_1 : \textbf{pc}\ pt \\ \Delta, \bar{a}; \Phi; \Gamma, x :: t_1, y :: \textbf{Stack}, z :: \textbf{String}, proceed :: t_1 \ \texttt{->}\ t_2 \vdash e_2 : t_2 \end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{advice around}\ \texttt{(|}e_1\texttt{|)}\ \texttt{(x,y,z)}\ \texttt{=}\ e_2 \dashv \cdot; \cdot}\ \text{wfsd:advice-aro}$$

$$\frac{\begin{array}{c} tm \in \{\textbf{before}, \textbf{after}\} \qquad \pi(tm, pt) = \texttt{<}\bar{a}\texttt{>}\ t \\ \Delta; \Phi; \Gamma \overset{loc}{\vdash} e_1 : \textbf{pc}\ pt \qquad \Delta, \bar{a}; \Phi; \Gamma, x :: t, y :: \textbf{Stack}, z :: \textbf{String} \vdash e_2 : t \end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{advice}\ tm\ \texttt{(|}e_1\texttt{|)}\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:t,y,z)}\ \texttt{=}\ e_2 \dashv \cdot; \cdot}\ \text{wfsd:advice-ann-befaft}$$

$$\frac{\begin{array}{c} \pi(\textbf{around}, pt) = \texttt{<}\bar{a}\texttt{>}\ t_1 \ \texttt{->}\ t_2 \qquad \Delta; \Phi; \Gamma \overset{loc}{\vdash} e_1 : \textbf{pc}\ pt \\ \Delta, \bar{a}; \Phi; \Gamma, x :: t_1, y :: \textbf{Stack}, z :: \textbf{String}, proceed :: t_1 \ \texttt{->}\ t_2 \vdash e_2 : t_2 \end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{advice around}\ \texttt{(|}e_1\texttt{|)}\ \texttt{<}\bar{a}\texttt{>}\ \texttt{(x:}t_1\texttt{,y,z):}t_2\ \texttt{=}\ e_2 \dashv \cdot; \cdot}\ \text{wfsd:advice-ann-aro}$$

$$\frac{\begin{array}{c} tm \in \{\textbf{before}, \textbf{after}\} \qquad \Delta' = \mathrm{FTV}(t) - \Delta \\ \Delta; \Phi; \Gamma \overset{loc}{\vdash} e_1 : \textbf{pc}\ pt \qquad \Delta, \Delta'; \Phi; \Gamma, x :: t, y :: \textbf{Stack}, z :: \textbf{String} \vdash e_2 : t \end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{case-advice}\ tm\ \texttt{(|}e_1\texttt{|)}\ \texttt{(x:t,y,z)}\ \texttt{=}\ e_2 \dashv \cdot; \cdot}\ \text{wfsd:cadvice-befaft}$$

$$\frac{\begin{array}{c} \Delta' = \mathrm{FTV}(t_1) \cup \mathrm{FTV}(t_2) - \Delta \qquad \Delta; \Phi; \Gamma \overset{loc}{\vdash} e_1 : \textbf{pc}\ pt \\ \Delta, \Delta'; \Phi; \Gamma, x :: t_1, y :: \textbf{Stack}, z :: \textbf{String}, proceed :: t_1 \ \texttt{->}\ t_2 \vdash e_2 : t_2 \end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{case-advice around}\ \texttt{(|}e_1\texttt{|)}\ \texttt{(x:}t_1\texttt{,y,z):}t_2\ \texttt{=}\ e_2 \dashv \cdot; \cdot}\ \text{wfsd:cadvice-aro}$$

Fig. 13. Declarative semantics for declarations

For example, the following term type checks according to the rules of our declarative specification, but *not* according to our algorithm. The declarative rules may assign **f** the type **String -> String**, but our algorithm will always choose the most general type, **<a> a -> a**

```
let
  fun f x = x
in
  #f : (String ~> String)#
end
```

We believe this term should not type check, as, given the definition of **f**, the user should expect that it has type **<a> a -> a** and might be used at many types. We conjecture that if the specification were required to choose the most general type for let-bound variables, it would correspond exactly with our algorithm, though we have not proved this fact. Happily, even though we are changing the specification

Patterns $\Delta; \Phi; \Gamma \vdash \mathsf{p} \dashv \Delta'; \Gamma'$

$$\frac{}{\Delta; \Phi; \Gamma \vdash \texttt{[]} \dashv \cdot; \cdot} \text{ wfspat:nil} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash \mathsf{x} \dashv \cdot; \cdot, \mathsf{x} :: \texttt{Stack}} \text{ wfspat:var}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \mathsf{p} \dashv \Delta'; \Gamma'}{\Delta; \Phi; \Gamma \vdash \texttt{_::p} \dashv \Delta'; \Gamma'} \text{ wfspat:wild}$$

$$\frac{\Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathsf{e} : \textbf{pc}\ \mathsf{pt} \quad \pi(\textbf{stk}, \mathsf{pt}) = \texttt{<}\bar{\mathsf{a}}\texttt{>}\ \mathsf{t} \quad \Delta; \Phi; \Gamma \vdash \mathsf{p} \dashv \Delta'; \Gamma'}{\Delta; \Phi; \Gamma \vdash \texttt{(|e|)(x,z)::p} \dashv \Delta', \bar{\mathsf{a}}; \Gamma', \mathsf{x}{:}\mathsf{t}, \mathsf{z}{:}\texttt{String}} \text{ wfspat:cons}$$

$$\frac{\Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathsf{e} : \textbf{pc}\ \mathsf{pt} \quad \pi(\textbf{stk}, \mathsf{pt}) = \texttt{<}\bar{\mathsf{a}}\texttt{>}\ \mathsf{t} \quad \Delta; \Phi; \Gamma \vdash \mathsf{p} \dashv \Delta'; \Gamma'}{\Delta; \Phi; \Gamma \vdash \texttt{(|e|)<}\bar{\mathsf{a}}\texttt{>(x:t,z)::p} \dashv \Delta', \bar{\mathsf{a}}; \Gamma', \mathsf{x}{:}\mathsf{t}, \mathsf{z}{:}\texttt{String}} \text{ wfspat:cons-ann}$$

Fig. 14. Declarative semantics for patterns

for pure ML terms, this change would not invalidate any ML programs. It merely reduces the number of alternate typing derivations for terms that use `let`. The derivation that uses the most general type is still available.

4.8 Soundness of type inference

Although our inference algorithm is not complete with respect to our declarative specification, we can show that our type inference rules are sound with respect to the declarative semantics.

Definition 4.1 Well-formed inference substitutions. $\Delta \vdash \Theta$ *iff for all* $\mathsf{X} \in \text{dom}(\Theta)$, $\Delta \vdash \Theta(\mathsf{X})$.

Theorem 4.2 Soundness of inference algorithm. *Given* $\Delta \vdash \Theta_1$ *then*

(1) *If* $\Theta_1 \vdash \mathsf{t}_1 = \mathsf{t}_2 \Rightarrow \Theta_2$ *then* $\Delta \vdash \Theta_2(\mathsf{t}_1) = \Theta_2(\mathsf{t}_2)$.
(2) *If* $\Theta_1 \vdash \mathsf{s}_1 \preceq \mathsf{s}_2 \Rightarrow \Theta_2$ *then* $\Delta \vdash \Theta_2(\mathsf{s}_1) \preceq \Theta_2(\mathsf{s}_2)$.
(3) *If* $\Theta_1; \Delta; \Phi; \Gamma \overset{\text{loc}}{\vdash} \mathsf{e} \Rightarrow \mathsf{t}; \Theta_2$ *then* $\Delta; \Phi; \Theta_2(\Gamma) \overset{\text{loc}}{\vdash} \mathsf{e} : \Theta_2(\mathsf{t})$.
(4) *If* $\Theta_1; \Delta; \Phi; \Gamma \vdash \mathsf{e} \Rightarrow \mathsf{t}; \Theta_2$ *then* $\Delta; \Phi; \Theta_2(\Gamma) \vdash \mathsf{e} : \Theta_2(\mathsf{t})$.
(5) *If* $\Theta_1; \Delta; \Phi; \Gamma \vdash \mathsf{d} \Rightarrow \Theta_2; \Phi'; \Gamma'$ *then* $\Delta; \Phi; \Theta_2(\Gamma) \vdash \mathsf{d} \dashv \Phi'; \Theta_2(\Gamma')$.
(6) *If* $\Theta_1; \Delta; \Phi; \Gamma \vdash \mathsf{p} \Rightarrow \Theta_2; \Delta'; \Gamma'$ *then* $\Delta; \Phi; \Theta_2(\Gamma) \vdash \mathsf{p} \dashv \Delta'; \Theta_2(\Gamma')$.

5. POLYMORPHIC CORE CALCULUS

In the previous section, we defined the syntax and static semantics for AspectML. One might choose to define the operational semantics for this language directly as a step-by-step term rewriting relation, as is often done for λ-calculi. However, the semantics of certain constructs is very complex. For example, function call, which is normally the simplest of constructs in the λ-calculus, combines the ordinary semantics of functions with execution of advice, the possibility of run-time type analysis and the extraction of metadata from the call stack.

Rather than attempt to specify all of these features directly, we specify the dynamic semantics in stages. First, we show how to compile the high-level constructs into a

core calculus, called $\mathbb{F}_A$. The translation breaks down complex high-level objects into substantially simpler, orthogonal concepts. This core calculus is also typed and the translation is type-preserving. Second, we define an operational semantics for the core calculus. Since we have proven that the $\mathbb{F}_A$ type system is sound and the translation from the source is type-preserving, AspectML is safe.

Our core calculus differs from the AspectML in that it is not oblivious – control-flow points that trigger advice must be explicitly annotated. Furthermore, it is explicitly typed – type abstraction and applications must also be explicitly marked in the program, as well as argument types for all functions. Also, we have carefully considered the orthogonality of the core calculus – for example, it does not include the combination of advice and type analysis that is found in the **`case-advice`** construct. For these reasons, one would not want to program in the core calculus. However, in exchange, the core calculus is much more expressive than the source language.

Because $\mathbb{F}_A$ is so expressive, we can easily experiment with the source language, by adding new features to scale the language up or removing features to improve reasoning power. For instance, by removing the single type analysis construct, we recover a language with parametric polymorphism. In fact, during the process of developing our AspectML, we have made numerous changes. Fortunately, for the most part, we have not had to make many changes in $\mathbb{F}_A$. Consequently, we have not needed to reprove soundness of the target language, only recheck that the translation is type-preserving, a much simpler task. Finally, in our implementation, the type checker for the $\mathbb{F}_A$ has caught many errors in the translation and helped the debugging process tremendously.

In this section, we describe the semantics of $\mathbb{F}_A$, and in Section 6, we sketch the translation from AspectML to $\mathbb{F}_A$.

5.1 The semantics of explicit join points

The core calculus $\mathbb{F}_A$ is an extension of the core calculus from WZL [2003] with polymorphic labels, polymorphic advice, and run-time type analysis. It also improves upon the semantics of context analysis.

For expository purposes, we begin with a simplified version of $\mathbb{F}_A$, and extend it in the following subsections. The initial syntax is summarized below.

$$
\begin{array}{rl}
\tau ::= & 1 \mid \mathsf{string} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \ldots \times \tau_n \mid \alpha \mid \forall \alpha.\tau \mid (\overline{\alpha}.\tau)\ \mathsf{label} \\
\mid & (\overline{\alpha}.\tau)\ \mathsf{pc} \mid \mathsf{advice} \\
e ::= & \langle\rangle \mid c \mid x \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \mathbf{fix}\ x{:}\tau.e \\
\mid & \langle\overline{e}\rangle \mid \mathbf{let}\ \langle\overline{x}\rangle = e_1\ \mathbf{in}\ e_2 \mid \mathbf{new}\ \overline{\alpha}.\tau \leq e \mid \ell \\
\mid & \{\overline{e}\} \mid e_1 \cup e_2 \mid e_1[\overline{\tau}][\![e_2]\!] \mid \{e_1[\overline{\alpha}](x{:}\tau_1, f{:}\tau_1 \to \tau_2) \rhd e_2\} \mid \Uparrow e \\
\mid & \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2)
\end{array}
$$

The basis of $\mathbb{F}_A$ is a typed λ-calculus with unit, strings and n-tuples. If $\overline{e}$ is a sequence of expressions $e_1 \ldots e_n$ for $n \geq 2$, then $\langle\overline{e}\rangle$ creates a tuple. The expression $\mathbf{let}\ \langle\overline{x}\rangle = e_1\ \mathbf{in}\ e_2$ binds the contents of a tuple to a vector of variables $\overline{x}$ in the scope of e_2. Unlike WZL, we add impredicative polymorphism to the core calculus, including type abstraction ($\Lambda\alpha.e$) and type application ($e[\tau]$). We write $\langle\rangle$ for the unit value and c for string constants.

Abstract labels, ℓ, play an essential role in the calculus. Labels are used to mark control-flow points where advice may be triggered, with the syntax $\ell[\overline{\tau}][\![e]\!]$. We

call such points in the core calculus *join points*. Unlike the labels in WZL, which are designed solely for **before** and **after** advice, labels in our calculus allow **around** advice. The value passed to the join point represents the **proceed** function that can be invoked by advice in the source language. The value returned once the join point executes is a function that executes any advice, or, if there is no advice, is the **proceed** function that was passed to the join point. For example, in the addition expression $v_1 + (\ell[\overline{\tau}][\![e_2]\!]\ v_3)$, after e_2 has been evaluated to a function v_2, evaluation of the resulting subterm $\ell[\overline{\tau}][\![v_2]\!]$ returns a function that, when applied to v_3, causes any advice associated with ℓ to be triggered.

Another difference from WZL is that the labels form a tree-shaped hierarchy. The top label in the hierarchy is $\mathcal{U}$. All other labels ℓ sit somewhere below $\mathcal{U}$. If $\ell_1 \leq \ell_2$ then ℓ_1 sits below ℓ_2 in the hierarchy. The expression $\mathbf{new}\ \overline{\alpha}.\tau \leq e$ evaluates e, obtaining a label ℓ_2, and generates a new label ℓ_1 defined such that $\ell_1 \leq \ell_2$. This label structure closely resembles the label hierarchy defined by Bruns et al. for their (untyped) μABC calculus [2004].

First class labels may be grouped into collections using the label set expression, $\{\overline{e}\}$. Label-sets can then be combined using the union operation, $e_1 \cup e_2$. Label-sets form the basis for specifying when a piece of advice applies.

All advice in $\mathbb{F}_A$ is around advice and exchanges data with a particular join point, making it similar to a function. Note that advice (written $\{e_1[\overline{\alpha}](x{:}\tau_1, f{:}\tau_1 \to \tau_2) \rhd e_2\}$) is first-class. The type variables $\overline{\alpha}$ and term variables x and f are bound in the body of the advice e_2. The variable f is bound to a "proceed" function for the around advice, and the variable x is the value that the join point's resulting function will be called upon. The expression e_1 is a label set that describes when the advice is triggered. For example, the advice $\{\{\overline{\ell}\}[](x{:}\mathsf{int}, f{:}\mathsf{int} \to \mathsf{int}) \rhd e\}$ is triggered when control-flow reaches a join point marked with ℓ_1, provided ℓ_1 is a descendant of a label in the set $\{\overline{\ell}\}$. If this advice has been installed in the program's dynamic environment, $v_1 + (\ell_1[][\![v_2]\!]\ v_3)$ evaluates to $v_1 + (e[v_2/f][v_3/x])$.

When labels are polymorphic, both types and values are exchanged between labeled control-flow points and advice. For instance, if ℓ_1 is a polymorphic label capable of marking a control-flow point with any type, we might write $v_1 + (\ell_1[\mathsf{int} \to \mathsf{int}][\![v_2]\!]\ v_3)$. In this case, if the advice $\{\{\ell_1\}[\alpha](x{:}\alpha, f{:}\alpha \to \alpha) \rhd e\}$ has been installed, then the previous expression evaluates to $v_1 + (e[\mathsf{int}/\alpha][v_2/f][v_3/x])$.

Advice is installed into the run-time environment with the expression $\Uparrow e$. Multiple pieces of advice may apply to the same control-flow point, so the order advice is installed in the run-time environment is important. WZL included mechanisms for installing advice both before or after currently installed advice, for simplicity $\mathbb{F}_A$ only allows advice to be installed after.

5.2 Operational semantics

The operational semantics must keep track of both the labels that have been generated and the advice that has been installed. The main operational judgment has the form $\Sigma; A; e \mapsto \Sigma'; A'; e'$. An allocation-style semantics keeps track of the set Σ of labels allocated so far (and their associated types) and A, an ordered list of installed advice. The label hierarchy is determined from the label set Σ by the relation $\Sigma \vdash \ell_1 \leq \ell_2$ in Figure 15.

Labels $\Sigma \vdash \ell_1 \leq \ell_2$

$$\frac{\ell{:}_1\, \overline{\alpha}.\tau \leq \ell_2 \in \Sigma}{\Sigma \vdash \ell_1 \leq \ell_1}\ \textsf{labsb:refl} \qquad \frac{\Sigma \vdash \ell_1 \leq \ell_2 \qquad \Sigma \vdash \ell_2 \leq \ell_3}{\Sigma \vdash \ell_1 \leq \ell_3}\ \textsf{labsb:trans}$$

$$\frac{\ell_1{:}\overline{\alpha}.\tau \leq \ell_2 \in \Sigma}{\Sigma \vdash \ell_1 \leq \ell_2}\ \textsf{labsb:def}$$

Fig. 15. Label subsumption in $\mathbb{F}_A$

The main rule of the operational semantics, ev:beta in Figure 16, decomposes an expression into an evaluation context and primitive reduct. The rules in Figure 16 with the form $\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'$ give the primitive β-reductions for expressions in the calculus.

We use the following syntax for values v and evaluation contexts E:

$$\begin{array}{rl} v ::= & \langle\rangle \mid \lambda x{:}\tau.e \mid \langle\overline{v}\rangle \mid \Lambda\alpha.e \mid \ell \mid \{\overline{v}\} \mid \{v[\overline{\alpha}](x{:}\tau_1, f{:}\tau_1 \to \tau_2) \rhd e\} \\ E ::= & [\,] \mid E\ e \mid v\ E \mid E[\tau] \mid \langle E, \ldots, e\rangle \mid \langle v, \ldots, E\rangle \mid E \cup e \mid v \cup E \\ \mid & \{E, \ldots, e\} \mid \{v, \ldots, E\} \mid \textbf{let}\ \langle\overline{x}\rangle = E\ \textbf{in}\ e \mid E[\overline{\tau}][\![e]\!] \mid v[\overline{\tau}][\![E]\!] \mid\ \Uparrow E \\ \mid & \{E[\overline{\alpha}](x{:}\tau, f{:}\tau_1 \to \tau_2) \rhd e\} \mid \textbf{new}\ \overline{\alpha}.\tau \leq E \end{array}$$

This definition of evaluation contexts gives the core aspect calculus a call-by-value, left-to-right evaluation order, but that choice is orthogonal to the design of the language.

A third judgment form $\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow v$ in Figure 17 describes, given a particular label ℓ marking a control-flow point, and type $\tau_1 \to \tau_2$ for the object at that point, how to pick out and compose the advice in context A that should execute at the control-flow point. (Note that the type of the object is not used to select the advice, it merely determines type annotations and instantiations in the result.) The result of this advice composition process is a function v that may be applied to a value with type $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$. The argument of the function v is the proceed function f with type $(\tau_1 \to \tau_2)$. The result of applying v to the proceed function is a function of type $\tau_1 \to \tau_2$ whose argument (of type τ_1) is passed as the variable x to the advice.

The advice composition judgment is described by three rules. The first composition rule represents when no advice is available, and, when passed a proceed function f and a value x, applies the proceed function f to x. The other rules examine the advice at the head of the advice heap. If the label ℓ is descended from one of the labels in the label set, then that advice is triggered. The head advice is composed with the function produced from examining the rest of the advice in the list. Not only does advice composition determine if ℓ is lower in the hierarchy than some label in the label set, but it also determines the substitution for the abstract types $\overline{\alpha}$ in the body of the advice. The typing rules ensure that if the advice is triggered, this substitution will always exist, so the execution of this rule does not require run-time type information.

5.2.1 *Type system.* The judgments for well-formed types are straightforward and are described in Figure 18. In addition to the standard unit, string, etc. types,

β-reduction $\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'$

$$\frac{}{\Sigma; A; (\lambda x{:}\tau.e)v \mapsto_\beta \Sigma; A; e[v/x]}\ \text{evb:app} \qquad \frac{}{\Sigma; A; (\Lambda\alpha.e)[\tau] \mapsto_\beta \Sigma; A; e[\tau/\alpha]}\ \text{evb:tapp}$$

$$\frac{}{\Sigma; A; \textbf{fix}\ x{:}\tau.e \mapsto_\beta \Sigma; A; e[\textbf{fix}\ x{:}\tau.e/x]}\ \text{evb:fix}$$

$$\frac{}{\Sigma; A; \textbf{let}\ \langle\overline{x}\rangle = \langle\overline{v}\rangle\ \textbf{in}\ e \mapsto_\beta \Sigma; A; e[\overline{v}/\overline{x}]}\ \text{evb:let}$$

$$\frac{}{\Sigma; A; \{\overline{\ell_1}\} \cup \{\overline{\ell_2}\} \mapsto_\beta \Sigma; A; \{\overline{\ell_1\,\ell_2}\}}\ \text{evb:union}$$

$$\frac{\ell' \notin \mathrm{dom}(\Sigma)}{\Sigma; A; \textbf{new}\ \overline{\alpha}.\tau \leq \ell \mapsto_\beta \Sigma, \ell'{:}\overline{\alpha}.\tau \leq \ell; A; \ell'}\ \text{evb:new}$$

$$\frac{}{\Sigma; A; \Uparrow v \mapsto_\beta \Sigma; v, A; \langle\rangle}\ \text{evb:adv-comp}$$

$$\frac{\exists\Theta.\Theta = \mathrm{MGU}(\tau_2, \tau_3)}{\Sigma; A; \textbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) \mapsto_\beta \Sigma; A; \Theta(e_1)}\ \text{evb:tcase1}$$

$$\frac{\neg\exists\Theta.\Theta = \mathrm{MGU}(\tau_2, \tau_3)}{\Sigma; A; \textbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) \mapsto_\beta \Sigma; A; e_2[\tau_2/\alpha]}\ \text{evb:tcase2}$$

$$\frac{\ell{:}\overline{\alpha}.\tau_1 \to \tau_2 \leq \ell' \in \Sigma \qquad \Sigma; A; \ell; (\tau_1 \to \tau_2)[\overline{\tau}/\overline{\alpha}] \Rightarrow v'}{\Sigma; A; \ell[\overline{\tau}][\![v]\!] \mapsto_\beta \Sigma; A; v'\,v}\ \text{evb:cut}$$

Reduction $\Sigma; A; e \mapsto \Sigma'; A'; e'$

$$\frac{\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'}{\Sigma; A; E[e] \mapsto \Sigma'; A'; E[e']}\ \text{ev:beta}$$

Fig. 16. Operational semantics for $\mathbb{F}_A$

Advice composition $\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow e$

$$\frac{}{\Sigma; \cdot; \ell; \tau_1 \to \tau_2 \Rightarrow \lambda f{:}\tau_1 \to \tau_2.f}\ \text{adv:empty}$$

$$\frac{\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow v \qquad \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \tau_1 \to \tau_2 = \tau_1' \to \tau_2'[\overline{\tau}/\overline{\alpha}]}{\Sigma; A, \{\{\overline{\ell}\}[\overline{\alpha}](x{:}\tau_1', f{:}\tau_1' \to \tau_2') \rhd e\}; \ell; \tau_1 \to \tau_2 \Rightarrow \lambda f{:}\tau_1 \to \tau_2.v\ (\lambda x{:}\tau_1.(e[\overline{\tau}/\overline{\alpha}]))}\ \text{adv:cons1}$$

$$\frac{\Sigma; A; \ell; \tau_1 \to \tau_2 \Rightarrow v \qquad \Sigma \vdash \ell \not\leq \ell_i}{\Sigma; A, \{\{\overline{\ell}\}[\overline{\alpha}](x{:}\tau_1', f{:}\tau_1' \to \tau_2') \rhd e\}; \ell; \tau_1 \to \tau_2 \Rightarrow v}\ \text{adv:cons2}$$

Fig. 17. Advice composition operational semantics for $\mathbb{F}_A$

Types $\Delta \vdash \tau$

$$\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ \mathsf{wftp{:}var} \qquad \frac{}{\Delta \vdash 1}\ \mathsf{wftp{:}unit} \qquad \frac{}{\Delta \vdash \mathsf{string}}\ \mathsf{wftp{:}str}$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \to \tau_2}\ \mathsf{wftp{:}arr} \qquad \frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall \alpha.\tau}\ \mathsf{wftp{:}all} \qquad \frac{\forall i \qquad \Delta \vdash \tau_i}{\Delta \vdash \tau_1 \times \ldots \times \tau_n}\ \mathsf{wftp{:}prod}$$

$$\frac{\Delta, \overline{\alpha} \vdash \tau}{\Delta \vdash (\overline{\alpha}.\tau)\ \mathsf{label}}\ \mathsf{wftp{:}lab} \qquad \frac{\Delta, \overline{\alpha} \vdash \tau}{\Delta \vdash (\overline{\alpha}.\tau)\ \mathsf{pc}}\ \mathsf{wftp{:}pc} \qquad \frac{}{\Delta \vdash \mathsf{advice}}\ \mathsf{wftp{:}advice}$$

$$\frac{}{\Delta \vdash \mathsf{stack}}\ \mathsf{wftp{:}stk}$$

Instance $\Delta \vdash \overline{\alpha}.\tau_1 \preceq \overline{\beta}.\tau_2$

$$\frac{\Delta, \overline{\alpha} \vdash \tau' \qquad \Delta, \overline{\beta} \vdash \tau'' \qquad (\exists \overline{\tau} \qquad \Delta, \overline{\beta} \vdash \tau_i \qquad \tau'[\overline{\tau}/\overline{\alpha}] = \tau'')}{\Delta \vdash \overline{\alpha}.\tau' \preceq \overline{\beta}.\tau''}\ \mathsf{inst}$$

Fig. 18. Well-formed types in $\mathbb{F}_A$

there are additional types for labels, pointcuts, advice, and stacks. The same figure also contains the $\mathbb{F}_A$ instance relation, which is similar to the AspectML instance relation contained in Figure 7 and described in Section 4.3.

The primary judgment of the $\mathbb{F}_A$ type system, $\Delta; \Gamma \vdash e : \tau$, indicates that the term e can be given the type τ, where free type variables appear in Δ and the types of term variables and labels appear in Γ. The typing rules for this judgment appear in Figure 19.

The novel aspect of the $\mathbb{F}_A$ type system is how it maintains the proper typing relationship between labels, label sets and advice. Because data is exchanged between labeled control-flow points and advice, these two entities must agree about the type of data that will be exchanged. To guarantee agreement, we must be careful with the types of labels (Rule wft:lab), which have the form $\overline{\alpha}.\tau$ label. To mark a control-flow point (Rule wft:cut), the label's type τ must be a function type. A label of type $\overline{\alpha}.\tau_1 \to \tau_2$ label may mark control-flow points containing a proceed function of type $\tau_1 \to \tau_2$ and values of any type τ_1, where free type variables $\overline{\alpha}$ are replaced by other types $\overline{\tau}$. For example, a label ℓ with the type $\alpha, \beta.\alpha \to \beta$ label may mark any control flow point, as α and β may be instantiated with any type. For example, below is a well-typed tuple in which ℓ marks two different control flow points, one of type $\gamma \to \gamma$ and the other of type bool $\to$ int:

$$\langle \Lambda\gamma.\lambda x{:}\gamma.(\ell[\gamma,\gamma][\![\lambda y{:}\gamma.y]\!]\ x), (\ell[\mathsf{bool},\mathsf{int}][\![\lambda y{:}\mathsf{bool}.\mathbf{if}\ y\ \mathbf{then}\ 1\ \mathbf{else}\ 0]\!]\ \mathbf{true})\rangle$$

Notice that marking control flow points that occur inside polymorphic functions is no different from marking other control flow points even though ℓ's abstract type variables α and β may be instantiated with different types each time the polymorphic function is called.

Labeling control-flow points correctly is one side of the equation. Constructing sets of labels and using them in advice safely is the other. Typing label set construction

Well-formed terms $\Delta;\Gamma \vdash e : \tau$

$$\frac{x{:}\tau \in \Gamma}{\Delta;\Gamma \vdash x : \tau}\ \textsf{wft:var} \qquad \frac{}{\Delta;\Gamma \vdash c : \textsf{string}}\ \textsf{wft:str} \qquad \frac{}{\Delta;\Gamma \vdash \langle\rangle : \textsf{1}}\ \textsf{wft:unit}$$

$$\frac{\Delta;\Gamma, x{:}\tau \vdash e : \tau \qquad \Delta \vdash \tau}{\Delta;\Gamma \vdash \textbf{fix}\ x{:}\tau.e : \tau}\ \textsf{wft:fix} \qquad \frac{\Delta;\Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta;\Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \textsf{wft:abs}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1\, e_2 : \tau_2}\ \textsf{wft:app} \qquad \frac{\Delta,\alpha;\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \textsf{wft:tabs}$$

$$\frac{\Delta;\Gamma \vdash e : \forall\alpha.\tau \qquad \Delta \vdash \tau'}{\Delta;\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}\ \textsf{wft:tapp} \qquad \frac{\Delta;\Gamma \vdash e_i : \tau_i}{\Delta;\Gamma \vdash \langle\overline{e}\rangle : \tau_1 \times \ldots \times \tau_n}\ \textsf{wft:tuple}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \times \ldots \times \tau_n \qquad \Delta;\Gamma,\overline{x{:}\tau} \vdash e_2 : \tau}{\Delta;\Gamma \vdash \textbf{let}\ \langle\overline{x}\rangle = e_1\ \textbf{in}\ e_2 : \tau}\ \textsf{wft:let} \qquad \frac{\ell{:}\overline{\alpha}.\tau \in \Gamma}{\Delta;\Gamma \vdash \ell : (\overline{\alpha}.\tau)\ \textsf{label}}\ \textsf{wft:lab}$$

$$\frac{\Delta;\Gamma \vdash e_i : (\overline{\alpha_i}.\tau_i)\ \textsf{label} \qquad \Delta \vdash \overline{\beta}.\tau \preceq \overline{\alpha_i}.\tau_i}{\Delta;\Gamma \vdash \{\overline{e}\} : (\overline{\beta}.\tau)\ \textsf{pc}}\ \textsf{wft:pc}$$

$$\frac{\Delta;\Gamma \vdash e_i : (\overline{\alpha}.\tau_i)\ \textsf{pc} \qquad \Delta \vdash \overline{\beta}.\tau \preceq \overline{\alpha}.\tau_i}{\Delta;\Gamma \vdash e_1 \cup e_2 : (\overline{\beta}.\tau)\ \textsf{pc}}\ \textsf{wft:union}$$

$$\frac{\Delta;\Gamma \vdash e : (\overline{\beta}.\tau_2)\ \textsf{label} \qquad \Delta \vdash \overline{\beta}.\tau_2 \preceq \overline{\alpha}.\tau_1}{\Delta;\Gamma \vdash \textbf{new}\ (\overline{\alpha}.\tau_1) \leq e : (\overline{\alpha}.\tau_1)\ \textsf{label}}\ \textsf{wft:new}$$

$$\frac{\Delta;\Gamma \vdash e_1 : (\overline{\alpha}.\tau' \to \tau'')\ \textsf{label} \qquad \Delta \vdash \tau_i \qquad \Delta;\Gamma \vdash e_2 : (\tau' \to \tau'')[\overline{\tau}/\overline{\alpha}]}{\Delta;\Gamma \vdash e_1\,[\overline{\tau}][\![e_2]\!] : (\tau' \to \tau'')[\overline{\tau}/\overline{\alpha}]}\ \textsf{wft:cut}$$

$$\frac{\Delta;\Gamma \vdash e : \textsf{advice}}{\Delta;\Gamma \vdash \Uparrow e : \textsf{1}}\ \textsf{wft:adv-inst}$$

$$\frac{\Delta;\Gamma \vdash e_1 : (\overline{\alpha}.\tau_1 \to \tau_2)\ \textsf{pc} \qquad \Delta,\overline{\alpha};\Gamma, x{:}\tau_1, f{:}\tau_1 \to \tau_2 \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \{e_1\,[\overline{\alpha}](x{:}\tau_1, f{:}\tau_1 \to \tau_2) \rhd e_2\} : \textsf{advice}}\ \textsf{wft:advice}$$

$$\frac{\begin{array}{c}\Delta,\alpha \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \Delta' = \mathrm{FTV}(\tau_3) - \Delta \\ (\Theta = \textsc{mgu}(\tau_2,\tau_3)\ \text{implies}\ \Delta,\Delta';\Theta(\Gamma) \vdash \Theta(e_1) : \Theta(\tau_1\,[\tau_3/\alpha])) \qquad \Delta,\alpha;\Gamma \vdash e_2 : \tau_1\end{array}}{\Delta;\Gamma \vdash \textbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) : \tau_1\,[\tau_2/\alpha]}\ \textsf{wft:tcase}$$

Fig. 19. Term typing rules for $\mathbb{F}_A$

in the core calculus is quite similar to typing pointcuts in the source. Each label in the set must be a generic instance of the type of the set. For example, given labels ℓ_1 of type $1 \to 1$ label and ℓ_2 of type $(1 \to \textsf{bool})$ label, a label set containing them can be given the type $(\alpha.1 \to \alpha)$ pc because $\alpha.1 \to \alpha$ can be instantiated to either $1 \to 1$ or $1 \to \textsf{bool}$. The rules for label sets and label set union (wft:pc and wft:union) ensure these invariants.

When typing advice in the core calculus (Rule wft:advice), the advice body must not make unwarranted assumptions about the types and values it is passed from

labeled control flow points. Consequently, if the label set e_1 has type $\overline{\alpha}.\tau_1 \to \tau_2$ pc then advice $\{e_1[\overline{\alpha}](f{:}\tau_1' \to \tau_2', x{:}\tau_1') \rhd e_2\}$ type checks only when $\tau_1' \to \tau_2'$ is $\tau_1 \to \tau_2$. The type $\tau_1' \to \tau_2'$ cannot be more specific than $\tau_1 \to \tau_2$. If advice needs to refine the type of $\tau_1 \to \tau_2$, it must do so explicitly with type analysis. In this respect the core calculus obeys the principle of *orthogonality*: advice is completely independent of type analysis.

The label hierarchy may be dynamically extended with $\mathbf{new}\ \overline{\alpha}.\tau \leq e$ (Rule wft:new). The argument e becomes the parent of the new label. For soundness, there must be a connection between the types of the child and parent labels: the child label must have a more specific type than its parent (written $\Delta \vdash \tau_1 \preceq \tau_2$ if τ_2 is more specific than τ_1). To see how label creation, labeled control flow points and advice are all used together in the core calculus, consider the following example. It creates a new label, installs advice for this label (that calls the proceed function f on its argument x – essentially an identity function) and then uses this label to mark a join point inside a polymorphic function.

$$\begin{array}{l}\mathbf{let}\ l = \mathbf{new}\ \alpha.\alpha \to \alpha \leq \mathcal{U}\ \mathbf{in} \\ \quad \mathbf{let}\ _ = \Uparrow \{l[\beta](x{:}\beta, f{:}\beta \to \beta) \rhd f\ x\}\ \mathbf{in} \\ \qquad \Lambda\gamma.l[\gamma][\![\lambda z{:}\gamma.z]\!]\end{array}$$

The $\mathbf{typecase}$ expression is slightly more general in the core calculus than in the source language. To support the preservation theorem, we must allow arbitrary types, not just type variables, to be the object of scrutiny. In each branch of $\mathbf{typecase}$, we know that the scrutinee is the same as the pattern. In the source language, we substituted the pattern for the scrutinized type variable when typechecking the branches. In the core calculus, however, we must compute the appropriate substitution, using the most general unifier (MGU). If no unifier exists, the branch can never be executed. In that case, the branch need not be checked.

The typing rules for the other constructs in the language including strings, unit, functions and tuples are fairly standard.

5.3 Stacks and stack analysis

Languages such as AspectJ include pointcut operators such as CFlow to enable advice to be triggered in a context-sensitive fashion. In $\mathbb{F}_A$, we not only provide the ability to reify and pattern match against stacks, as in AspectML, but also allow manual construction of stack frames. In fact, managing the structure of the stack is entirely up to the program itself. Stacks are just one possible extension enabled by $\mathbb{F}_A$'s orthogonality.

WZL's monomorphic core calculus also contained the ability to query the stack, but the stack was not first-class and queries had to be formulated as regular expressions. Our pattern matching facilities are simpler and more general. Moreover, they fit perfectly within the functional programming idiom. Aside from the polymorphic patterns, they are quite similar to the stack patterns used by Dantas and Walker [2006].

Below are the necessary new additions to the syntax of $\mathbb{F}_A$ for storing type and value information on the stack, capturing and representing the current stack as a data structure, and analyzing a reified stack. The operational rules for execution of

Stack reification $\mathit{data}(E)$

$$
\begin{aligned}
\mathit{data}([\,]) &= \bullet \\
\mathit{data}(\mathbf{store}\ \ell[\overline{\tau}][\![v]\!]\ \mathbf{in}\ E) &= \mathit{data}(E) :: \ell[\overline{\tau}][\![v]\!] \\
\mathit{data}(E[E']) &= \mathit{data}(E') \text{ otherwise}
\end{aligned}
$$

β-reduction $\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'$

$$\frac{}{\Sigma; A; \mathbf{store}\ \ell[\overline{\tau}][\![v_1]\!]\ \mathbf{in}\ v_2 \mapsto_\beta \Sigma; A; v_2}\ \textsf{evb:store}$$

$$\frac{\Sigma \vdash v \simeq \varphi \rhd \Theta}{\Sigma; A; \mathbf{stkcase}\ v\ (\varphi \Rightarrow e_1, x \Rightarrow e_2) \mapsto_\beta \Sigma; A; \Theta(e_1)}\ \textsf{evb:scase1}$$

$$\frac{\Sigma \vdash v \not\simeq \varphi \rhd \Theta}{\Sigma; A; \mathbf{stkcase}\ v\ (\varphi \Rightarrow e_1, x \Rightarrow e_2) \mapsto_\beta \Sigma; A; e_2\,[v/x]}\ \textsf{evb:scase2}$$

Reduction $\Sigma; A; e \mapsto \Sigma'; A'; e'$

$$\frac{\mathit{data}(E) = v}{\Sigma; A; E[\mathbf{stack}] \mapsto \Sigma; A; E[v]}\ \textsf{ev:stk} \qquad \frac{\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'}{\Sigma; A; E[e] \mapsto \Sigma'; A'; E[e']}\ \textsf{ev:beta}$$

Stack-matching $\Sigma \vdash v \simeq \varphi \rhd \Theta$

$$\frac{}{\Sigma \vdash \bullet \simeq \bullet \rhd \cdot}\ \textsf{sm:nil}$$

$$\frac{\begin{array}{c}\Sigma \vdash v_2 \simeq \varphi \rhd \Theta \\ \ell{:}\overline{\beta}.\tau_2 \leq \ell' \in \Sigma \qquad \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \exists\overline{\sigma}.\tau_2\,[\overline{\tau}/\overline{\beta}] = \tau_1\,[\overline{\sigma}/\overline{\alpha}]\end{array}}{\Sigma \vdash \ell[\overline{\tau}][\![v_1]\!]{::}v_2 \simeq \{\overline{\ell}\}[\overline{\alpha}][\![x]\!]{:}\tau_1 :: \varphi \rhd \Theta, \overline{\sigma}/\overline{\alpha}, v_1/x}\ \textsf{sm:cons}$$

$$\frac{\Sigma \vdash v' \simeq \varphi \rhd \Theta}{\Sigma \vdash \ell[\overline{\tau}][\![v]\!]{::}v' \simeq _{::}\varphi \rhd \Theta}\ \textsf{sm:wild} \qquad \frac{}{\Sigma \vdash v \simeq x \rhd \cdot, v/x}\ \textsf{sm:var}$$

Fig. 20. Stack operational semantics

stack commands may be found in Figure 20 and the typing rules in Figure 21.

$$
\begin{aligned}
\tau &::= \ldots \mid \textsf{stack} \\
e &::= \ldots \mid \mathbf{stack} \mid \bullet \mid \ell[\overline{\tau}][\![v_1]\!]{::}v_2 \mid \mathbf{store}\ e_1[\overline{\tau}][\![e_2]\!]\ \mathbf{in}\ e_3 \\
&\mid \mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3) \\
E &::= \ldots \mid \mathbf{store}\ v[\overline{\tau}][\![E]\!]\ \mathbf{in}\ e \mid \mathbf{store}\ v_1[\overline{\tau}][\![v_2]\!]\ \mathbf{in}\ E \\
&\mid \mathbf{stkcase}\ E\ (\rho \Rightarrow e_1, x \Rightarrow e_2) \\
&\mid \mathbf{stkcase}\ v\ (P \Rightarrow e_1, x \Rightarrow e_2) \\
\rho &::= \bullet \mid e[\overline{\alpha}][\![y]\!]{:}\tau{::}\rho \mid x \mid _{::}\rho \\
\varphi &::= \bullet \mid v[\overline{\alpha}][\![y]\!]{::}\varphi \mid x \mid _{::}\varphi \\
P &::= E[\overline{\alpha}][\![y]\!]{::}\varphi \mid e[\overline{\alpha}][\![y]\!]{::}P \mid _{::}P
\end{aligned}
$$

The operation $\mathbf{store}\ e_1[\overline{\tau}][\![e_2]\!]\ \mathbf{in}\ e_3$ allows the programmer to store data e_2 marked by the label e_1 in the evaluation context of the expression e_3. Because this label may

Well-formed terms $\Delta;\Gamma \vdash e : \tau$

$$\frac{\Delta;\Gamma \vdash e_1 : (\overline{\alpha}.\tau)\ \mathsf{label} \quad \Delta \vdash \tau_i \quad \Delta;\Gamma \vdash e_2 : \tau[\overline{\tau}/\overline{\alpha}] \quad \Delta;\Gamma \vdash e_3 : \tau'}{\Delta;\Gamma \vdash \mathbf{store}\ e_1\,[\overline{\tau}][\![e_2]\!]\ \mathbf{in}\ e_3 : \tau'}\ \mathsf{wft{:}store}$$

$$\frac{}{\Delta;\Gamma \vdash \mathbf{stack} : \mathsf{stack}}\ \mathsf{wft{:}stk} \qquad \frac{}{\Delta;\Gamma \vdash \bullet : \mathsf{stack}}\ \mathsf{wft{:}stk\text{-}nil}$$

$$\frac{\ell{:}\overline{\alpha}.\tau \in \Gamma \quad \Delta \vdash \tau_i \quad \Delta;\Gamma \vdash v_1 : \tau[\overline{\tau}/\overline{\alpha}] \quad \Delta;\Gamma \vdash v_2 : \mathsf{stack}}{\Delta;\Gamma \vdash \ell[\overline{\tau}][\![v_1]\!]{::}v_2 : \mathsf{stack}}\ \mathsf{wft{:}stk\text{-}cons}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \mathsf{stack} \quad \Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma' \quad \Delta,\Delta';\Gamma,\Gamma' \vdash e_2 : \tau \quad \Delta;\Gamma,x{:}\mathsf{stack} \vdash e_3 : \tau}{\Delta;\Gamma \vdash \mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3) : \tau}\ \mathsf{wft{:}scase}$$

Well-formed patterns $\Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma'$

$$\frac{}{\Delta;\Gamma \vdash \bullet \dashv \cdot;\cdot}\ \mathsf{wfpt{:}nil} \qquad \frac{}{\Delta;\Gamma \vdash x \dashv \cdot;\cdot,x{:}\mathsf{stack}}\ \mathsf{wfpt{:}var}$$

$$\frac{\Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma'}{\Delta;\Gamma \vdash _{::}\rho \dashv \Delta';\Gamma'}\ \mathsf{wfpt{:}wild} \qquad \frac{\Delta;\Gamma \vdash e : (\overline{\alpha}.\tau)\ \mathsf{pc} \quad \Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma'}{\Delta;\Gamma \vdash e[\overline{\alpha}][\![x]\!]{:}\tau{::}\rho \dashv \Delta',\overline{\alpha};\Gamma',x:\tau}\ \mathsf{wfpt{:}store}$$

Fig. 21. Stack typing

be polymorphic, it must be instantiated with type arguments $\overline{\tau}$. The term $\mathbf{stack}$ captures the data stored in its execution context E as a first-class data structure. This context is converted into a data structure, using the auxiliary function $\mathsf{data}(E)$. We represent a stack using the list with terms $\bullet$ for the empty list and :: (cons) to prefix an element onto the front of the list. A list of stored stack information may be analyzed with the pattern matching term $\mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3)$. This term attempts to match the pattern ρ against e_1, a reified stack. Note that stack patterns, ρ, include first-class pointcuts so they must be evaluated to pattern values, φ, to resolve these pointcuts before matching.

If, after evaluation, the pattern value successfully matches the stack, then the expression e_2 evaluates, with its pattern variables replaced with the corresponding part of the stack. Otherwise execution continues with e_3. These rules rely on the stack matching relation $\Sigma \vdash v \simeq \varphi \rhd \Theta$ that compares a stack pattern value φ with a reified stack v to produce a substitution Θ.

5.4 Type Safety

We have shown that $\mathbb{F}_A$ is type sound through the usual Progress and Preservation theorems. Figure 22 describes the typing rules for the term variable and label context Γ, the label heap Σ, and the advice heap A. It should be noted that Γ and Σ always contain the top label $\mathcal{U}$. We use the judgment $\vdash (\Sigma; A; e)$ ok in Rule (wfcfg) to denote a well-formed abstract machine state.

THEOREM 5.1 PROGRESS. *If* $\vdash (\Sigma; A; e)$ ok *then either the configuration is finished, that is* e *is a value, or there exists another configuration* $\Sigma'; A'; e'$ *such that* $\Sigma; A; e \mapsto \Sigma'; A'; e'$.

Term Variable and Label Context $\Delta \vdash \Gamma$

$$\frac{}{\Delta \vdash \mathcal{U}{:}\alpha.\alpha}\ \text{wfc:base} \qquad \frac{\Delta \vdash \tau \qquad \Delta \vdash \Gamma}{\Delta \vdash \Gamma, x{:}\tau}\ \text{wfc:var} \qquad \frac{\Delta, \overline{\alpha} \vdash \tau \qquad \Delta \vdash \Gamma}{\Delta \vdash \Gamma, \ell{:}\overline{\alpha}.\tau}\ \text{wfc:lab}$$

Label Heaps $\vdash \Sigma : \Gamma$

$$\frac{}{\vdash (\mathcal{U}{:}\alpha.\alpha \leq \mathcal{U}) : (\mathcal{U}{:}\alpha.\alpha)}\ \text{wflh:base}$$

$$\frac{\ell_2{:}\overline{\beta}.\tau_2 \leq \ell_3 \in \Sigma \qquad \cdot \vdash \overline{\beta}.\tau_2 \preceq \overline{\alpha}.\tau_1 \qquad \vdash \Sigma : \Gamma}{\vdash (\Sigma, \ell_1{:}\overline{\alpha}.\tau_1 \leq \ell_2) : (\Gamma, \ell_1{:}\overline{\alpha}.\tau_1)}\ \text{wflh:cons}$$

Advice Heaps $\Gamma \vdash A$ ok

$$\frac{}{\Gamma \vdash \cdot \text{ ok}}\ \text{wfah:base} \qquad \frac{\cdot; \Gamma \vdash \nu : \text{advice} \qquad \Gamma \vdash A \text{ ok}}{\Gamma \vdash A, \nu \text{ ok}}\ \text{wfah:cons}$$

Machine Configurations $\vdash (\Sigma; A; e)$ ok

$$\frac{\vdash \Sigma : \Gamma \qquad \Gamma \vdash A \text{ ok} \qquad \cdot; \Gamma \vdash e : \tau}{\vdash (\Sigma; A; e) \text{ ok}}\ \text{wfcfg}$$

Fig. 22. Machine configuration typing rules in $\mathbb{F}_A$

THEOREM 5.2 PRESERVATION. *If* $\vdash (\Sigma; A; e)$ ok *and* $\Sigma; A; e \mapsto \Sigma'; A'; e'$, *then* Σ' *and* A' *extend* Σ *and* A *such that* $\vdash (\Sigma'; A'; e')$ ok.

6. TRANSLATION

We give an operational semantics to well-typed AspectML programs by defining a type-preserving translation into the $\mathbb{F}_A$ language. This translation is defined by the following mutually recursive judgments for over terms, types, patterns, declarations and pointcut designators. The translation was significantly inspired by those in found in WZL [2003] and Dantas and Walker [2006].

Throughout the translation we assume that there exists an implicit injection from AspectML type and term variables ($a, b, \ldots$ and $x, y, \ldots$) and $\mathbb{F}_A$ type and term variables ($\alpha, \beta, \ldots$ and $x, y, \ldots$).

We begin by defining several translation abbreviations in Figure 23. These allow us to specify function abstraction and application translation rules, type abstraction and application rules, **stkcase** and **typecase** rules, and other rules more succinctly.

The essence of the translation is that join points must be made explicit in $\mathbb{F}_A$. Therefore, we translate functions so that that they include an explicitly labeled join point surrounding the function body and another that stores information on the stack as the function executes. More specifically, for each function we create two labels $f_{\textbf{around}}$, and $f_{\textbf{stk}}$ for these join points. So that AspectML programs can refer to the pointcut designators of any function using the **any** keyword , all labels $f_{\textbf{around}}$ are derived from a distinguished label $\mathcal{U}_{\textbf{around}}$. Likewise, $\mathcal{U}_{\textbf{stk}}$ is the parent of all $f_{\textbf{stk}}$. These constructions can be seen in Figure 25, where in Rule tctx:empty, the

Simple abbreviations

$$\begin{aligned}
\textbf{let}\ x:\tau = e_1\ \textbf{in}\ e_2 &\triangleq (\lambda x{:}\tau.e_2)e_1\\
\forall\overline{a}.\tau &\triangleq \forall\alpha_1\ldots\forall\alpha_n.\tau\\
\Lambda\overline{a}.e &\triangleq \Lambda\alpha_1\ldots\forall\alpha_n.e\\
e[\overline{\tau}] &\triangleq e[\tau_1]\ldots[\tau_n]\\
_ &\triangleq x
\end{aligned}$$

(where x fresh)

Multi-arm **stkcase** abbreviation $\textbf{stkcase}\ e_1'\ (\overline{\rho}\Rightarrow\overline{e}, _\Rightarrow e_2')$

$$\begin{aligned}
&\textbf{stkcase}\ e_1\ (\overline{\rho}\Rightarrow\overline{e}, _\Rightarrow e_2) \triangleq\\
&\quad \textbf{let}\ y:\mathsf{stack} = e_1\ \textbf{in}\ \textbf{stkcase}'\ y\ (\overline{\rho}\Rightarrow\overline{e}, _\Rightarrow e_2)\\
&\textbf{stkcase}'\ e_1\ (\rho\Rightarrow e_2, x\Rightarrow e_3) \triangleq \textbf{stkcase}\ e_1\ (\rho\Rightarrow e_2, _\Rightarrow e_3)\\
&\textbf{stkcase}'\ e_1\ (\rho\Rightarrow e_2, \overline{\rho}\Rightarrow\overline{e}, x\Rightarrow e_2) \triangleq\\
&\quad \textbf{stkcase}\ e_1\ (\rho\Rightarrow e_2, _\Rightarrow \textbf{stkcase}'\ e_1\ (\overline{\rho}\Rightarrow\overline{e}, _\Rightarrow e_2))
\end{aligned}$$

(where y fresh)

Multi-arm **typecase** abbreviation $\textbf{typecase}[\alpha.\tau_1']\ \alpha\ (\overline{\tau}\Rightarrow\overline{e}, \alpha\Rightarrow e')$

$$\begin{aligned}
&\textbf{typecase}[\alpha.\tau']\ \alpha\ (\tau\Rightarrow e_1, \alpha\Rightarrow e_2) \triangleq \textbf{typecase}[\alpha.\tau']\ \alpha\ (\tau\Rightarrow e_1, \alpha\Rightarrow e_2)\\
&\textbf{typecase}[\alpha.\tau']\ \alpha\ (\tau\Rightarrow e_1, \overline{\tau}\Rightarrow\overline{e}, \alpha\Rightarrow e_2) \triangleq\\
&\quad \textbf{typecase}[\alpha.\tau']\ \alpha\ (\tau\Rightarrow e_1, \alpha\Rightarrow \textbf{typecase}[\alpha.\tau']\ \alpha\ (\overline{\tau}\Rightarrow\overline{e}, \alpha\Rightarrow e_2))
\end{aligned}$$

Pointcut splitting helper $\pi(tm, e)$

$$\begin{aligned}
\pi(\textbf{around}, e) &= \textbf{let}\ \langle x, _\rangle = e\ \textbf{in}\ x\\
\pi(\textbf{stk}, e) &= \textbf{let}\ \langle _, x\rangle = e\ \textbf{in}\ x
\end{aligned}$$

(where x fresh)

Fig. 23. Translation abbreviations

$\mathcal{U}_{\textbf{around}}$ and $\mathcal{U}_{\textbf{stk}}$ labels are created in the $\mathbb{F}_A$ context. Similarly, in Rules tctx:lc-fun and tctx:gc-fun, $f_{\textbf{around}}$ and $f_{\textbf{stk}}$ labels are created in the $\mathbb{F}_A$ term variable context for each f in the AspectML context. Finally, the actual $\mathcal{U}_{\textbf{around}}$ and $\mathcal{U}_{\textbf{stk}}$ labels are created in Rule tprog directly underneath $\mathcal{U}$ in the $\mathbb{F}_A$ label hierarchy.

Pointcuts are translated into a tuple of two $\mathbb{F}_A$ pointcuts in Figure 26. The pointcut **any** becomes a tuple containing the $\mathcal{U}_{\textbf{around}}$ and $\mathcal{U}_{\textbf{stk}}$ pointcuts, which, as explained previously, contain the parents of all **around** and **stk** labels respectively. Sets of functions are translated into tuples of pointcuts containing their associated **before**, **after**, and **stk** labels. We then use a pointcut splitting helper function to pick either the first or second element of the pointcut tuple depending on whether we attempting to use the pointcut in advice or in a stack pattern.

The translation of functions (Rule tds:fun-ann) in Figure 27 begins by creating the labels, $f_{\textbf{around}}$ and $f_{\textbf{stk}}$ for the function's join points. Inside the body of the translated function, a **store** statement marks the function's stack frame with the $f_{\textbf{stk}}$ label. The function's body is η-expanded and passed to the join point to be

Type variable context translation $\Delta \Longrightarrow \Delta'$

$\Delta \Longrightarrow \Delta'$ iff for all $a \in \Delta$, $\alpha \in \Delta'$.

Polytype translation $\Delta \vdash s \xrightarrow{\text{type}} \tau$

$$\frac{\Delta, \overline{a} \vdash t \xrightarrow{\text{type}} \tau'}{\Delta \vdash \texttt{<}\overline{a}\texttt{>}\ t \xrightarrow{\text{type}} \forall \overline{\alpha}.\tau'}\ \text{tpy:all}$$

Monotype translation $\Delta \vdash t \xrightarrow{\text{type}} \tau$

$$\frac{a \in \Delta}{\Delta \vdash a \xrightarrow{\text{type}} \alpha}\ \text{ttp:var} \qquad \frac{}{\Delta \vdash X \xrightarrow{\text{type}} 1}\ \text{ttp:unif} \qquad \frac{}{\Delta \vdash \texttt{Unit} \xrightarrow{\text{type}} 1}\ \text{ttp:unit}$$

$$\frac{}{\Delta \vdash \texttt{String} \xrightarrow{\text{type}} \mathsf{string}}\ \text{ttp:str} \qquad \frac{}{\Delta \vdash \texttt{Stack} \xrightarrow{\text{type}} \mathsf{stack}}\ \text{ttp:stk}$$

$$\frac{\Delta \vdash t_1 \xrightarrow{\text{type}} \tau_1' \qquad \Delta \vdash t_2 \xrightarrow{\text{type}} \tau_1'}{\Delta \vdash t_1\ \texttt{->}\ t_2 \xrightarrow{\text{type}} \tau_1' \to \tau_2'}\ \text{ttp:fun}$$

$$\frac{\Delta, \overline{a} \vdash t_1 \xrightarrow{\text{type}} \tau_1' \qquad \Delta, \overline{a} \vdash t_2 \xrightarrow{\text{type}} \tau_2'}{\begin{array}{l}\Delta \vdash \texttt{pc (<}\overline{a}\texttt{>}\ t_1\ \texttt{\~{}>}\ t_2\texttt{)} \xrightarrow{\text{type}} \\ \quad (\overline{\alpha}.(\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{pc} \times (\overline{\alpha}.\tau_1' \times \mathsf{string})\ \mathsf{pc}\end{array}}\ \text{ttp:pc}$$

Term variable context translation $\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma'$

$$\frac{}{\begin{array}{l}\Delta; \Phi \vdash \cdot \Longrightarrow\ \mathcal{U}_{\texttt{around}}{:}(\alpha\beta.(\alpha \times \mathsf{stack} \times \mathsf{string}) \to (\beta \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label}, \\ \qquad\qquad\quad \mathcal{U}_{\texttt{stk}}{:}(\alpha.\alpha \times \mathsf{string})\ \mathsf{label}\end{array}}\ \text{tctx:empty}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad \Delta \vdash s \xrightarrow{\text{type}} \tau}{\Delta; \Phi \vdash \Gamma, x :: s \Longrightarrow \Gamma', x{:}\tau}\ \text{tctx:lc} \qquad \frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad \Delta \vdash s \xrightarrow{\text{type}} \tau}{\Delta; \Phi \vdash \Gamma, x : s \Longrightarrow \Gamma', x{:}\tau}\ \text{tctx:gc}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad f \in \Phi \qquad \Delta \vdash s \xrightarrow{\text{type}} \forall\overline{\alpha}.\tau_1 \to \tau_2}{\begin{array}{l}\Delta; \Phi \vdash \Gamma, f :: s \Longrightarrow \Gamma',\ f_{\texttt{around}}{:}(\overline{\alpha}.(\tau_1 \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2 \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label}, \\ \qquad\qquad\qquad\qquad\quad f_{\texttt{stk}}{:}(\overline{\alpha}.\tau_1 \times \mathsf{string})\ \mathsf{label}, \\ \qquad\qquad\qquad\qquad\quad f{:}\forall\overline{\alpha}.\tau_1 \to \tau_2\end{array}}\ \text{tctx:lc-fun}$$

$$\frac{\Delta; \Phi \vdash \Gamma \Longrightarrow \Gamma' \qquad f \in \Phi \qquad \Delta \vdash s \xrightarrow{\text{type}} \forall\overline{\alpha}.\tau_1 \to \tau_2}{\begin{array}{l}\Delta; \Phi \vdash \Gamma, f : s \Longrightarrow \Gamma',\ f_{\texttt{around}}{:}(\overline{\alpha}.(\tau_1 \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2 \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label}, \\ \qquad\qquad\qquad\qquad\quad f_{\texttt{stk}}{:}(\overline{\alpha}.\tau_1 \times \mathsf{string})\ \mathsf{label}, \\ \qquad\qquad\qquad\qquad\quad f{:}\forall\overline{\alpha}.\tau_1 \to \tau_2\end{array}}\ \text{tctx:gc-fun}$$

Fig. 24. Context and type translation

Programs $\mathsf{e : t} \overset{\mathsf{prog}}{\Longrightarrow} e'$

$$\frac{\cdot;\cdot;\cdot \vdash \mathsf{e : t} \overset{\mathsf{term}}{\Longrightarrow} e'}{\mathsf{e : t} \overset{\mathsf{prog}}{\Longrightarrow}} \text{tprog}$$

$$\begin{aligned}&\textbf{\textit{let}}\ \mathcal{U}_{\mathsf{around}} : (\alpha\beta.(\alpha \times \mathsf{stack} \times \mathsf{string}) \to (\beta \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label} = \\ &\quad \textbf{\textit{new}}\ (\alpha\beta.(\alpha \times \mathsf{stack} \times \mathsf{string}) \to (\beta \times \mathsf{stack} \times \mathsf{string})) \leq \mathcal{U}\ \textbf{\textit{in}} \\ &\textbf{\textit{let}}\ \mathcal{U}_{\mathsf{stk}} : (\alpha.\alpha \times \mathsf{string})\ \mathsf{label} = \\ &\quad \textbf{\textit{new}}\ (\alpha.\alpha \times \mathsf{string}) \leq \mathcal{U}\ \textbf{\textit{in}}\ e'\end{aligned}$$

Fig. 25. Program translation

Local term translation $\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathsf{e : t} \overset{\mathsf{term}}{\Longrightarrow} e'$

$$\frac{\Delta \vdash \mathsf{t} \qquad \Delta; \Phi; \Gamma \vdash \mathsf{e : t} \overset{\mathsf{term}}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathsf{e{:}t : t} \overset{\mathsf{term}}{\Longrightarrow} e'} \text{lttm:cnv} \qquad \frac{\mathsf{x :: t} \in \Gamma}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathsf{x : t} \overset{\mathsf{term}}{\Longrightarrow} x} \text{lttm:var}$$

$$\frac{}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathsf{c} : \mathbf{String} \overset{\mathsf{term}}{\Longrightarrow} c} \text{lttm:string} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathbf{()} : \mathbf{Unit} \overset{\mathsf{term}}{\Longrightarrow} \langle\rangle} \text{lttm:unit}$$

$$\frac{\forall i \qquad \mathsf{f}_i \in \Phi \qquad \Gamma(\mathsf{f}_i) = \texttt{<}\bar{a}\texttt{>}\ \mathsf{t}_{1,i}\ \texttt{->}\ \mathsf{t}_{2,i} \qquad \Delta \vdash \texttt{<}\bar{b}\texttt{>}\ \mathsf{t}_1\ \texttt{->}\ \mathsf{t}_2 \preceq \texttt{<}\bar{a}\texttt{>}\ \mathsf{t}_{1,i}\ \texttt{->}\ \mathsf{t}_{2,i}}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \texttt{\#}\bar{\mathsf{f}}\texttt{:(<}\bar{b}\texttt{>}\ \mathsf{t}_1\ \texttt{\~{}>}\ \mathsf{t}_2\texttt{)\#} : \mathbf{pc}\ \texttt{(<}\bar{b}\texttt{>}\ \mathsf{t}_1\ \texttt{\~{}>}\ \mathsf{t}_2\texttt{)} \overset{\mathsf{term}}{\Longrightarrow} \langle\{\overline{f_{\mathsf{around}}}\}, \{\overline{f_{\mathsf{stk}}}\}\rangle} \text{lttm:set-ann}$$

$$\frac{}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathbf{any} : \mathbf{pc}\ \texttt{(<ab> a \~{}> b)} \overset{\mathsf{term}}{\Longrightarrow} \langle\{\mathcal{U}_{\mathsf{around}}\}, \{\mathcal{U}_{\mathsf{stk}}\}\rangle} \text{lttm:any}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \mathsf{d}\,;\mathsf{e : t} \overset{\mathsf{decs}}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \mathbf{let}\ \mathsf{d}\ \mathbf{in}\ \mathsf{e : t} \overset{\mathsf{term}}{\Longrightarrow} e'} \text{lttm:let}$$

Fig. 26. Local expression translation

used as the proceed function f by any advice triggered by the f_{around} label. Because AspectML advice expects the current stack and a string of the function name, we also insert **stack**s and string constants into the join points.

Advice translation is defined in Figures 27 and 28. The most significant difference between advice in AspectML and $\mathbb{F}_A$ is that $\mathbb{F}_A$ has no notion of a trigger time. Because the pointcut argument of the advice will translate into a tuple of two $\mathbb{F}_A$ pointcuts, `tm` is used to determine which component is used. The translation also splits the input of the advice into the two arguments that AspectML expects and immediately installs the advice.

To simplify the translation, only the rules for **around** advice (tds:adv-ann and tds:cadv-aro) are directly defined – **before** advice (Rules tds:adv-bef and tds:cadv-bef) becomes **around** advice where the **before** advice is executed, then **proceed** is called on the result. Similarly, **after** advice (Rules tds:adv-aft and tds:cadv-aft) becomes **around** advice where the **proceed** function executes the function body, and then the **after** advice is run on the result. Finally, the **around case-advice** declaration (Rule tds:cadv-aro) uses a ***typecase*** expression to perform the necessary type analysis.

Declarations $\Delta; \Phi; \Gamma \vdash \mathbf{d}\,; \mathtt{e} : \mathtt{t} \stackrel{\mathsf{decs}}{\Longrightarrow} e'$

$$\frac{\Delta, \bar{a} \vdash \mathtt{t_1 \text{ -> } t_2} \stackrel{\mathsf{type}}{\Longrightarrow} \tau_1' \to \tau_2' \qquad \Delta, \bar{a}; \Phi; \Gamma, \mathtt{f} :: \mathtt{t_1 \text{ -> } t_2}, \mathtt{x} :: \mathtt{t_1} \vdash \mathtt{e_1} : \mathtt{t_2} \stackrel{\mathsf{term}}{\Longrightarrow} e_1' \quad \Delta; \Phi, \mathtt{f}; \Gamma, \mathtt{f} :: \mathtt{<\bar{a}>\ t_1 \text{ -> } t_2} \vdash \mathtt{e_2} : \mathtt{t} \stackrel{\mathsf{term}}{\Longrightarrow} e_2'}{\begin{array}{l} \Delta; \Phi; \Gamma \vdash \mathbf{fun}\ \mathtt{f\ <\bar{a}>\ (x{:}t_1){:}t_2 = e_1; e_2} : \mathtt{t} \stackrel{\mathsf{decs}}{\Longrightarrow} \\ \quad \mathbf{let}\ f_{\mathsf{around}} : (\overline{\alpha}.(\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string}))\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.(\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to (\tau_2' \times \mathsf{stack} \times \mathsf{string})) \leq \mathcal{U}_{\mathsf{around}}\ \mathbf{in} \\ \quad \mathbf{let}\ f_{\mathsf{stk}} : (\overline{\alpha}.\tau_1' \times \mathsf{string})\ \mathsf{label} = \mathbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{string}) \leq \mathcal{U}_{\mathsf{stk}}\ \mathbf{in} \\ \quad \mathbf{let}\ f : \forall\overline{\alpha}.\tau_1' \to \tau_2' = \Lambda\overline{\alpha}.\mathbf{fix}\ f : \tau_1' \to \tau_2'.\lambda x{:}\tau_1'.\mathbf{store}\ f_{stk}[\overline{\alpha}][\![\langle x, \text{“f”}\rangle]\!]\ \mathbf{in} \\ \qquad (\mathbf{let}\ \langle x'', _, _\rangle = f_{\mathsf{around}}[\overline{\alpha}][\![\lambda w : (\tau_1' \times \mathsf{stack} \times \mathsf{string}). \\ \qquad\quad \mathbf{let}\ \langle x, y, z\rangle = w\ \mathbf{in}\ \langle e_1', y, z\rangle]\!]\ (x', \mathbf{stack}, \text{“f”})\ \mathbf{in}\ x'') \\ \quad e_2' \end{array}} \text{ tds:fun-ann}$$

$$\frac{\Delta, \bar{a} \vdash \mathtt{t_1} \qquad \Delta, \bar{a} \vdash \mathtt{t_2} \qquad \Delta; \Phi; \Gamma \vdash \mathbf{fun}\ \mathtt{f\ <\bar{a}>\ (x{:}t_1){:}t_2 = e_1\,; e_2} : \mathtt{t} \stackrel{\mathsf{decs}}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \mathbf{fun}\ \mathtt{f\ x = e_1\,; e_2} : \mathtt{t} \stackrel{\mathsf{decs}}{\Longrightarrow} e'} \text{ tds:fun}$$

$$\frac{\mathtt{tm} \in \{\mathbf{before}, \mathbf{after}\} \quad \Delta, \bar{a} \vdash \mathtt{t_1} \qquad \Delta; \Phi; \Gamma \vdash \mathbf{advice}\ \mathtt{tm\ (|e_1|)\ <\bar{a}>\ (x{:}t_1, y, z) = e_2\,; e_3} : \mathtt{t_2} \stackrel{\mathsf{decs}}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \mathbf{advice}\ \mathtt{tm\ (|e_1|)\ (x, y, z) = e_2\,; e_3} : \mathtt{t_2} \stackrel{\mathsf{decs}}{\Longrightarrow} e'} \text{ tds:adv}$$

$$\frac{\Delta, \bar{a} \vdash \mathtt{t_2} \quad \Delta; \Phi; \Gamma \vdash \mathbf{advice\ around}\ \mathtt{(|e_1|)\ <\bar{a}>\ (x{:}t_1, y, z){:}t_2 = proceed\ e_2\,; e_3} : \mathtt{t_3} \stackrel{\mathsf{decs}}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \mathbf{advice\ before}\ \mathtt{(|e_1|)\ <\bar{a}>\ (x{:}t_1, y, z) = e_2\,; e_3} : \mathtt{t_3} \stackrel{\mathsf{decs}}{\Longrightarrow} e'} \text{ tds:adv-bef}$$

$$\frac{\Delta, \bar{a} \vdash \mathtt{t_2} \qquad \begin{array}{l}\Delta; \Phi; \Gamma \vdash \mathbf{advice\ around}\ \mathtt{(|e_1|)\ <\bar{a}>\ (x{:}t_1, y, z){:}t_2 =} \\ \quad (\mathbf{let}\ x : \mathtt{t_2} = (\mathtt{proceed}\ x)\ \mathbf{in}\ \mathtt{e_2})\,; \mathtt{e_3} : \mathtt{t_3} \stackrel{\mathsf{decs}}{\Longrightarrow} e'\end{array}}{\Delta; \Phi; \Gamma \vdash \mathbf{advice\ after}\ \mathtt{(|e_1|)\ <\bar{a}>\ (x{:}t_2, y, z) = e_2\,; e_3} : \mathtt{t_3} \stackrel{\mathsf{decs}}{\Longrightarrow} e'} \text{ tds:adv-aft}$$

$$\frac{\Delta, \bar{a} \vdash \mathtt{t_1} \quad \Delta, \bar{a} \vdash \mathtt{t_2} \qquad \Delta; \Phi; \Gamma \vdash \mathbf{advice\ around}\ \mathtt{(|e_1|)\ <\bar{a}>\ (x{:}t_1, y, z){:}t_2 = e_2\,; e_3} : \mathtt{t_3} \stackrel{\mathsf{decs}}{\Longrightarrow} e'}{\Delta; \Phi; \Gamma \vdash \mathbf{advice\ around}\ \mathtt{(|e_1|)\ (x, y, z) = e_2\,; e_3} : \mathtt{t_3} \stackrel{\mathsf{decs}}{\Longrightarrow} e'} \text{ tds:adv-aro}$$

Fig. 27. Declaration translation (Part 1)

Finally, we include the global expression translation rules (Figure 29) and the pattern translation rules (Figure 30). In Rule tpat:cons-ann, the second, stack element of the pointcut tuple is selected by the π function as the pointcut to be used by the **stkcase** expression.

It is straightforward to show that programs that are well-typed with respect to our algorithm will produce a translation.

Theorem 6.1 Translation defined on well-typed programs. *If* $\cdot; \cdot; \cdot; \cdot \vdash \mathtt{e} \Rightarrow \mathtt{t}; \Theta$ *then* $\Theta(\mathtt{e}) : \Theta(\mathtt{t}) \stackrel{\mathsf{prog}}{\Longrightarrow} e'$

We have proved that the translation always produces well-formed $\mathbb{F}_A$ programs.

Theorem 6.2 Translation type soundness. *If* $\mathtt{e} : \mathtt{t} \stackrel{\mathsf{prog}}{\Longrightarrow} e'$ *then* $\cdot; \cdot \vdash e' : \tau'$ *where* $\cdot \vdash \mathtt{t} \stackrel{\mathsf{type}}{\Longrightarrow} \tau'$.

Declarations $\Delta; \Phi; \Gamma \vdash d ; e : t \xRightarrow{\text{decs}} e'$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash^{loc} e_1 : \textbf{pc}\ pt \xRightarrow{\text{term}} e_1' \qquad \pi(\textbf{around}, pt) = \texttt{<}\bar{a}\texttt{>}\ t_1 \to t_2 \\ \pi(\textbf{around}, e_1') = e_1'' \qquad \Delta, \bar{a} \vdash t_1 \to t_2 \xRightarrow{\text{type}} \tau_1' \to \tau_2' \\ \Delta, \bar{a}; \Phi; \Gamma, x{:}t_1, y{:}\textbf{Stack}, z{:}\textbf{String}, proceed{:}t_1 \to t_2 \vdash e_2 : t_2 \xRightarrow{\text{term}} e_2' \\ \Delta; \Phi; \Gamma \vdash e_3 : t_3 \xRightarrow{\text{term}} e_3'\end{array}}{\begin{array}{l}\Delta; \Phi; \Gamma \vdash \textbf{advice around (|}e_1\textbf{|)}\ \texttt{<}\bar{a}\texttt{>}\ \textbf{(}x{:}t_1, y, z\textbf{)}{:}t_2 = e_2; e_3 : t_3 \xRightarrow{\text{decs}} \\ \quad \mathbf{let}\ _ : 1 = \Uparrow \{e_1''.\bar{\alpha}(x{:}(\tau_1' \times \text{stack} \times \text{string}), \\ \qquad f{:}(\tau_1' \times \text{stack} \times \text{string}) \to (\tau_2' \times \text{stack} \times \text{string})) \to \\ \qquad\quad \mathbf{let}\ \langle x, y, z\rangle = x\ \mathbf{in} \\ \qquad\quad \langle e_2'[\lambda x : \tau_1'.\mathbf{let}\ \langle x, _, _\rangle = f\ \langle x, y, z\rangle\ \mathbf{in}\ x/\text{proceed}], y, z\rangle\} \\ \quad \mathbf{in}\ e_3'\end{array}}\ \text{tds:adv-ann}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \textbf{case-advice around (|}e_1\textbf{|) (}x{:}t_1, y, z\textbf{)}{:}a = \text{proceed}\ e_2 ; e_3 : t_2 \xRightarrow{\text{decs}} e'}{\Delta; \Phi; \Gamma \vdash \textbf{case-advice before (|}e_1\textbf{|) (}x{:}t_1, y, z\textbf{)} = e_2 ; e_3 : t_2 \xRightarrow{\text{decs}} e'}\ \text{tds:cadv-bef}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash \textbf{case-advice around (|}e_1\textbf{|) (}x{:}a, y, z\textbf{)}{:}t_1 = \\ (\mathbf{let}\ x : t_1 = (\text{proceed}\ x)\ \mathbf{in}\ e_2) ; e_3 : t_2 \xRightarrow{\text{decs}} e'\end{array}}{\Delta; \Phi; \Gamma \vdash \textbf{case-advice after (|}e_1\textbf{|) (}x{:}t_1, y, z\textbf{)} = e_2 ; e_3 : t_2 \xRightarrow{\text{decs}} e'}\ \text{tds:cadv-aft}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash^{loc} e_1 : \textbf{pc}\ pt \xRightarrow{\text{term}} e_1' \qquad \pi(\textbf{around}, pt) = \texttt{<}\bar{a}\texttt{>}\ t_1 \to t_2 \\ \pi(\textbf{around}, e_1') = e_1'' \qquad \bar{b} = \mathrm{FTV}(t_3) \cup \mathrm{FTV}(t_4) - \Delta \\ \Delta, \bar{a} \vdash t_1 \to t_2 \xRightarrow{\text{type}} \tau_1' \to \tau_2' \qquad \Delta, \bar{b} \vdash t_3 \to t_4 \xRightarrow{\text{type}} \tau_3' \to \tau_4' \\ \Delta, \bar{b}; \Phi; \Gamma, x{:}t_3, y{:}\textbf{Stack}, z{:}\textbf{String}, proceed{:}t_3 \to t_4 \vdash e_2 : t_4 \xRightarrow{\text{term}} e_2' \\ \Delta; \Phi; \Gamma \vdash e_3 : t \xRightarrow{\text{term}} e_3'\end{array}}{\begin{array}{l}\Delta; \Phi; \Gamma \vdash \textbf{case-advice around (|}e_1\textbf{|) (}x{:}t_3, y, z\textbf{)}{:}t_4 = e_2; e_3 : t \xRightarrow{\text{decs}} \\ \quad \mathbf{let}\ _ : 1 = \Uparrow \{e_1''.\bar{\alpha}x{:}(\tau_1' \times \text{stack} \times \text{string}), \\ \qquad f{:}(\tau_1' \times \text{stack} \times \text{string}) \to (\tau_2' \times \text{stack} \times \text{string}) \to \\ \qquad\quad \mathbf{let}\ \langle x, y, z\rangle = x\ \mathbf{in} \\ \qquad\quad \langle \mathbf{typecase}[\alpha.\alpha]\ \tau_1' \to \tau_2' \\ \qquad\qquad (\tau_3' \to \tau_4' \Rightarrow e_2'[\lambda x : \tau_1'.\mathbf{let}\ \langle x, _, _\rangle = f\ \langle x, y, z\rangle\ \mathbf{in}\ x/\text{proceed}], \\ \qquad\qquad \alpha \Rightarrow x), y, z\rangle\} \\ \quad \mathbf{in}\ e_3'\end{array}}\ \text{tds:cadv-aro}$$

Fig. 28. Declaration translation (Part 2)

Furthermore, because we know that $\mathbb{F}_A$ is a type safe language, AspectML inherits safety as a consequence.

THEOREM 6.3 ASPECTML SAFETY. *Suppose* $e : t \xRightarrow{\text{prog}} e'$ *then either* e' *fails to terminate or there exists a sequence of reductions* $\cdot; \cdot; e' \mapsto^* \Sigma; A; e''$ *to a finished configuration.*

7. RELATED WORK

Over the last several years, researchers have begun to build semantic foundations for aspect-oriented programming paradigms [Wand et al. 2003; Douence et al. 2001; Clifton and Leavens 2002; Jagadeesan et al. 2003a; 2003b; Masuhara et al. 2002; Walker et al. 2003; Douence et al. 2004; Bruns et al. 2004]. As mentioned earlier, our work builds upon the framework proposed by Walker, Zdancewic, and Ligatti [2003],

Pattern splitting helper $\mathrm{split}(\Xi, e)$

$$\mathrm{split}(\cdot, e) = e$$
$$\mathrm{split}(\Xi, x \mapsto (y, z), e) = \mathrm{split}(\Xi, \mathbf{let}\ \langle y, z\rangle = x\ \mathbf{in}\ e)$$

Global term translation $\Delta; \Phi; \Gamma \vdash \mathtt{e} : \mathtt{t} \xrightarrow{\mathsf{term}} e'$

$$\frac{\Delta; \Phi; \Gamma \vdash^{loc} \mathtt{e} : \mathtt{t} \xrightarrow{\mathsf{term}} e'}{\Delta; \Phi; \Gamma \vdash \mathtt{e} : \mathtt{t} \xrightarrow{\mathsf{term}} e'}\ \text{gttm:cnv} \qquad \frac{\Gamma(x) = \mathtt{<}\overline{\mathtt{a}}\mathtt{>}\ \mathtt{t} \qquad \Delta \vdash \mathtt{t}_i \xrightarrow{\mathsf{type}} \tau'_i}{\Delta; \Phi; \Gamma \vdash x : \mathtt{t}[\overline{\mathtt{t}}/\overline{\mathtt{a}}] \xrightarrow{\mathsf{term}} x[\overline{\tau'}]}\ \text{gttm:var}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \mathtt{e}_1 : \mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2 \xrightarrow{\mathsf{term}} e'_1 \qquad \Delta; \Phi; \Gamma \vdash \mathtt{e}_2 : \mathtt{t}_1 \xrightarrow{\mathsf{term}} e'_2}{\Delta; \Phi; \Gamma \vdash \mathtt{e}_1\,\mathtt{e}_2 : \mathtt{t}_2 \xrightarrow{\mathsf{term}} e'_1\, e'_2}\ \text{gttm:app}$$

$$\frac{\begin{array}{c}\Delta; \Phi; \Gamma \vdash \mathtt{e} : \mathbf{Stack} \xrightarrow{\mathsf{term}} e_t \\ \left(\begin{array}{l}\forall i\ \ \Delta; \Phi; \Gamma \vdash \mathtt{p}_i \xrightarrow{\mathsf{pat}} \rho'_i \dashv \Delta_i; \Gamma_i; \Xi_i \\ \quad\ \ \Delta, \Delta_i; \Phi; \Gamma, \Gamma_i \vdash \mathtt{e}_i : \mathtt{t} \xrightarrow{\mathsf{term}} e'_i \\ \quad\ \ \Delta; \Phi; \Gamma \vdash \mathtt{e}' : \mathtt{t} \xrightarrow{\mathsf{term}} e'_t\end{array}\right)\end{array}}{\begin{array}{c}\Delta; \Phi; \Gamma \vdash \mathbf{stkcase}\ \mathtt{e}\ \mathtt{(}\overline{\mathtt{p=>e}}\ \mathtt{|_=>}\ \mathtt{e}'\mathtt{)} : \mathtt{t} \xrightarrow{\mathsf{term}} \\ \mathbf{stkcase}\ e_t\ (\overline{\rho' \Rightarrow \mathrm{split}(\Xi, e')}, x \Rightarrow e'_t)\end{array}}\ \text{gttm:scase}$$

$$\frac{\begin{array}{c}\Delta, \mathtt{a} \vdash \mathtt{t} \xrightarrow{\mathsf{type}} \tau' \qquad \Delta; \Phi; \Gamma \vdash \mathtt{e} : \mathtt{t} \xrightarrow{\mathsf{term}} e' \\ \left(\begin{array}{ll}\forall i\ \ \Delta_i = \mathrm{FTV}(\mathtt{t}_i) - \Delta & \Delta, \Delta_i \vdash \mathtt{t}_i \xrightarrow{\mathsf{type}} \tau'_i \\ \quad\ \ \mathtt{a} \notin \mathrm{FTV}(\mathtt{t}_i) & \Delta, \Delta_i; \Phi; \Gamma\langle \mathtt{t}_i/\mathtt{a}\rangle \vdash \mathtt{e}_i[\mathtt{t}_i/\mathtt{a}] : \mathtt{t}[\mathtt{t}_i/\mathtt{a}] \xrightarrow{\mathsf{term}} e'_i\end{array}\right)\end{array}}{\begin{array}{c}\Delta; \Phi; \Gamma \vdash \mathbf{typecase}\mathtt{<t>}\ \mathtt{a}\ \mathtt{(}\overline{\mathtt{t=>e}}\ \mathtt{|_=>}\ \mathtt{e)} : \mathtt{t} \xrightarrow{\mathsf{term}} \\ \mathbf{typecase}[\alpha.\tau']\ \alpha\ (\overline{\tau' \Rightarrow e'}, \alpha \Rightarrow e')\end{array}}\ \text{gttm:tcase}$$

$$\frac{\Delta; \Phi; \Gamma \vdash \mathtt{d}\,;\mathtt{e} : \mathtt{t} \xrightarrow{\mathsf{decs}} e'}{\Delta; \Phi; \Gamma \vdash \mathbf{let}\ \mathtt{d}\ \mathbf{in}\ \mathtt{e} : \mathtt{t} \xrightarrow{\mathsf{term}} e'}\ \text{gttm:let}$$

Fig. 29. Global expression translation

but extends it with polymorphic versions of functions, labels, label sets, stacks, pattern matching, advice and the auxiliary mechanisms to define the meaning of each of these constructs. We also define "around" advice and a novel type inference algorithm that is conservative over Hindley-Milner inference, which were missing from WZL's work.

Our core calculus also has interesting connections to Bruns et al.'s μABC calculus in that the structure of labels in the two systems are similar. However, the connection is not so deep, as μABC is untyped. It would be interesting to explore whether the type structure of our calculus can be used to define a type system for μABC.

Concurrently with our research,[10] Masuhara, Tatsuzawa, and Yonezawa [2005] have implemented an aspect-oriented version of core O'Caml they call Aspectual Caml. Their implementation effort is impressive and deals with several features

[10]We made a preliminary report describing our type system available on the Web in October 2004, and a technical report with more details in December 2004. As far as we are aware, Masuhara et al.'s work first appeared in March 2005.

Splitting context translation $\Delta; \Gamma \vdash \Xi \Longrightarrow \Gamma'$

$$\frac{}{\Delta; \cdot \vdash \cdot \Longrightarrow \cdot}\ \text{tsctx:empty} \qquad \frac{\Delta; \Gamma \vdash \Xi \Longrightarrow \Gamma'}{\Delta; \Gamma, x{:}\mathbf{Stack} \vdash \Xi \Longrightarrow \Gamma', x{:}\mathsf{stack}}\ \text{tsctx:cons1}$$

$$\frac{\Delta; \Gamma \vdash \cdot \Longrightarrow \Gamma'}{\Delta; \Gamma, x{:}t \vdash \cdot \Longrightarrow \Gamma'}\ \text{tsctx:cons2}$$

$$\frac{\Delta; \Gamma \vdash \Xi \Longrightarrow \Gamma' \qquad \Delta \vdash t \overset{\mathsf{type}}{\Longrightarrow} \tau}{\Delta; \Gamma, y{:}t, z{:}\mathbf{String} \vdash \Xi, x \mapsto (y, z) \Longrightarrow \Gamma', x{:}\tau \times \mathsf{string},}\ \text{tsctx:cons3}$$

Patterns $\Delta; \Phi; \Gamma \vdash p \overset{\mathsf{pat}}{\Longrightarrow} \rho \dashv \Delta'; \Gamma'; \Xi$

$$\frac{}{\Delta; \Phi; \Gamma \vdash \mathtt{[]} \overset{\mathsf{pat}}{\Longrightarrow} \bullet \dashv \cdot; \cdot; \cdot}\ \text{tpat:nil} \qquad \frac{}{\Delta; \Phi; \Gamma \vdash x \overset{\mathsf{pat}}{\Longrightarrow} x \dashv \cdot; \cdot, x{:}\mathbf{Stack}; \cdot}\ \text{tpat:var}$$

$$\frac{\Delta; \Phi; \Gamma \vdash p \overset{\mathsf{pat}}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi}{\Delta; \Phi; \Gamma \vdash \mathtt{_::}p \overset{\mathsf{pat}}{\Longrightarrow} _{::}\rho' \dashv \Delta'; \Gamma'; \Xi}\ \text{tpat:wild}$$

$$\frac{\Delta, \overline{a} \vdash t \qquad \Delta; \Phi; \Gamma \vdash \mathtt{(|e|)<\overline{a}>(x{:}t,z)::}p \overset{\mathsf{pat}}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi}{\Delta; \Phi; \Gamma \vdash \mathtt{(|e|)(x,z)::}p \overset{\mathsf{pat}}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi}\ \text{tpat:cons}$$

$$\frac{\Delta; \Phi; \Gamma \overset{\mathrm{loc}}{\vdash} e : \mathbf{pc}\ \mathtt{(<\overline{a}>}\ t_1\ \mathtt{\sim>}\ t_2\mathtt{)} \overset{\mathsf{term}}{\Longrightarrow} e' \qquad \pi(\mathbf{stk}, e') = e'' \qquad \Delta; \Phi; \Gamma \vdash p \overset{\mathsf{pat}}{\Longrightarrow} \rho' \dashv \Delta'; \Gamma'; \Xi \qquad y \text{ fresh}}{\Delta; \Phi; \Gamma \vdash \mathtt{(|e|)<\overline{a}>(x{:}t_1,z)::}p \overset{\mathsf{pat}}{\Longrightarrow} e''[\overline{\alpha}][\![y]\!]{::}\rho' \dashv \Delta', \overline{a}; \Gamma', x{:}t_1, z{:}\mathbf{String}; \Xi, y \mapsto (x, z)}\ \text{tpat:cons-ann}$$

Fig. 30. Pattern translation

we have not considered here including curried functions and datatypes. Although there are similarities between AspectML and Aspectual Caml, there are also many differences:

—Pointcut designators in AspectML can only reference names that are in scope. AspectML names are indivisible and α-vary as usual. In Aspectual Caml, programmers use regular expressions to refer to all names that match the regular expression in any scope. For instance, **`get*`** references all objects with a name beginning with **`get`** in all scopes.

—Aspectual Caml does not check pointcut designators for well-formedness. When a programmer writes the pointcut designator **`call f (x:int)`**, the variable **`f`** is assumed to be a function and the argument **`x`** is assumed to have type **`int`**. There is some run-time checking to ensure safety, but it is not clear what happens in the presence of polymorphism or type definitions. Aspectual Caml does not appear to have run-time type analysis.

—Aspectual Caml pointcuts are second-class citizens. It is not possible to write down the type of a pointcut in Aspectual Caml, or pass a pointcut to a function, store it in a tuple, etc.

—The previous two limitations have made it possible to develop a two-phase type inference algorithm for Aspectual Caml (ordinary O'Caml type inference occurs first and inference for pointcuts and advice occurs second), which bears little resemblance to the type inference algorithm described in this paper.

—There is no formal description of the Aspectual Caml type system, type inference algorithm or operational semantics. We have a formal description of both the static semantics and the dynamic semantics of AspectML. AspectML's type system has been proven sound with respect to its operational semantics.

Since we first published work on AspectML, Meng Wang, Kung Chen, and Siau-Cheng Khoo have also begun to examining language design problems in combining aspects with a polymorphic functional language [2006]. Their design makes fundamentally different assumptions about aspects that lead to significant differences in expressiveness:

—Their advice is scoped such that it is not possible to install advice that will affect functions that have already been defined. We feel that this has both positive and negative consequences for the language. It is positive, because they use a type-directed weaving algorithm (not unlike the way type classes are compiled to dictionary passing in Haskell) to completely eliminate the need to dynamically calculate advice composition, as our operational semantics does. However, we feel that this design decision does not adequately take into account the needs of separate compilation: A programmer could not compile a program separately from its advice. Furthermore, some of our most interesting uses of advice so far have involved advising an already defined function.

—Their advice is named. Not only is this useful as mnemonic for the programmer, but it allows them to advise advice. We do not presently name our advice, but there is no fundamental reason that we cannot, and likewise support advice advisement.

—Like Aspectual Caml, their pointcuts are second-class. We believe that first-class pointcuts are an important step toward allowing programmers to develop reusable libraries of advice.

—Their design does not provide a mechanism for examining the call-stack or obtaining information about the specific function being advised. But we do not see any technical challenges that would prevent them from adding such features.

To our knowledge, the only other previous study of the interaction between polymorphism and aspect-oriented programming features has occurred in the context of Lieberherr, Lorenz and Ovlinger's Aspectual Collaborations [2003]. They extend a variant of AspectJ with a form of module that allows programmers to choose the join points (i.e., control-flow points) that are exposed to external aspects. Aspectual Collaborations has parameterized aspects that resemble the parameterized classes of Generic Java. When a parameterized aspect is linked into a module, concrete class names replace the parameters. Since types are merely names, the sort of

polymorphism necessary is much simpler (at least in certain ways) than required by a functional programming language. For instance, there is no need to develop a generalization relation and type analysis may be replaced by conventional object-oriented down-casts. Overall, the differences between functional and object-oriented language structure have caused our two groups to find quite different solutions to the problem of constructing generic advice.

Closely related to Aspectual Collaborations is Aldrich's notion of Open Modules [2005]. The central novelty of this proposal is a special module sealing operator that hides internal control-flow points from external advice. Aldrich used logical relations to show that sealed modules have a powerful implementation-independence property. In earlier work [Dantas and Walker 2003], we suggested augmenting these proposals with access-control specifications in the module interfaces that allow programmers to specify whether or not data at join points may be read or written. Neither of these proposals consider polymorphic types or modules that can hide type definitions.

Dantas and Walker [2006] have developed "harmless advice". With harmless advice, aspects may only observe computations performed by the program, they may not alter them. Specifically Dantas and Walker prove a noninterference-style result showing that a program using mutable references may be composed with arbitrary advice and not impact the behavior of original program.

Additionally, building on concurrent work by Washburn and Weirich [2005], we are working on extending AspectML with protection mechanisms that ensure data abstractions are respected even in the presence of type-analyzing advice. This extended language, called InforML, will feature a simple module system of dependent sums in addition to its information-flow type and kind system. We expect that the same policies that can be used to track and restrict the flow of type information can also provide guarantees similar to harmless advice.

Tucker and Krishnamurthi [2003] developed a variant of Scheme with aspect-oriented features. They demonstrate the pleasures of programming with point-cuts and advice as first-class objects. Of course, Scheme is dynamically typed. Understanding the type structure of statically-typed polymorphic functional languages with advice is the main contribution of this paper. In particular, we develop a type inference algorithm and reconcile the typing of advice with polymorphic functions.

8. CONCLUSION

This paper defines AspectML, a new functional and aspect-oriented programming language. In particular, we focus on the synergy between polymorphism and aspect-oriented programming – the combination is clearly more expressive than the sum of its parts. At the simplest level, our language allows programmers to reference control-flow points that appear in polymorphic code. However, we have also shown that polymorphic pointcuts are necessary even when the underlying code base is completely monomorphic. Otherwise, there is no way to assemble a collection of join points that appear in code with different types. In addition, run-time type analysis allows programmers to define polymorphic advice that behaves differently depending upon the type of its argument.

From a technical standpoint, we have defined a type inference algorithm for AspectML that handles first-class polymorphic pointcuts in a simple but effective way, allowing programmers to write convenient security, profiling or debugging libraries. We give AspectML a semantics by compiling it into a typed intermediate calculus. We have proven the intermediate calculus is type-safe. The reason for giving AspectML a semantics this way is to first decompose complex source-level syntax into a series of simple and orthogonal constructs. Giving a semantics to the simple constructs of the intermediate calculus and proving the intermediate calculus sound is quite straightforward.

The definition of the intermediate calculus is also an important contribution of this work. The most interesting part is the definition of our label hierarchy, which allows us to form groups of related control flow points. Here, polymorphism is again essential: it is not possible to define these groups in a monomorphic language. The second interesting element of our calculus is our support for reification of the current call stack. In addition to being polymorphic, our treatment of static analysis is more flexible, simpler semantically and easier for programmers to use than the initial proposition by WZL. Moreover, it is a perfect fit with standard data-driven functional programming idioms.

ACKNOWLEDGMENT

We appreciate the insightful comments by anonymous reviewers on earlier revisions of this work.

REFERENCES

Abadi, M. and Fournet, C. 2003. Access control based on execution history. In *Proceedings of the 10th Symposium on Network and Distributed System Security* (San Diego, CA). Internet Society, Reston, VA, 107–121.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA.

Aldrich, J. 2005. Open modules: Modular reasoning about advice. In *Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK). 144–168.

Avgustinov, P., Hajiyev, E., Ongkingco, N., de Moor, O., Sereni, D., Tibble, J., and Verbaere, M. 2007. Semantics of static pointcuts in AspectJ. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France). ACM Press, New York, NY, USA, 11–23.

Barendregt, H. 1985. *The lambda calculus: its syntax and semantics*. Number 103 in Studies in Logic. North Holland.

Bauer, L., Ligatti, J., and Walker, D. 2005. Composing security policies in polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL). ACM Press, New York, NY, USA, 305–314.

Bruns, G., Jagadeesan, R., Jeffrey, A., and Riely, J. 2004. muABC: A minimal aspect calculus. In *Proceedings of the 15th International Conference on Concurrency Theory* (London, UK), P. Gardner and N. Yoshida, Eds. Lecture Notes in Computer Science, vol. 3170. Springer, Berlin, Germany, 209–224.

Clifton, C. and Leavens, G. T. 2002. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Proceedings of the 2002 Workshop on Foundations of Aspect-Oriented Languages* (Enschede, The Netherlands). 33–44.

Colcombet, T. and Fradet, P. 2000. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Boston, MA). ACM Press, New York, NY, USA, 54–66.

Colyer, A. and Clement, A. 2004. Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*. ACM Press, New York, NY, USA, 56–65.

Damas, L. and Milner, R. 1982. Principal type schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, NM). ACM Press, New York, NY, USA, 207–212.

Dantas, D. S. and Walker, D. 2003. Aspects, information hiding and modularity. Tech. Rep. TR-696-04, Princeton University. Nov.

Dantas, D. S. and Walker, D. 2006. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, SC). ACM Press, New York, NY, USA, 383–396.

Dantas, D. S., Walker, D., Washburn, G., and Weirich, S. 2005a. PolyAML: A polymorphic aspect-oriented functional programming language. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia). ACM Press, New York, NY, 306–319.

Dantas, D. S., Walker, D., Washburn, G., and Weirich, S. 2005b. PolyAML: A polymorphic aspect-oriented functional programmming language (extended version). Tech. Rep. MS-CIS-05-07, University of Pennsylvania. May.

Douence, R., Fradet, P., and Südholt, M. 2004. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development* (Lancaster, UK). ACM Press, New York, NY, USA, 141–150.

Douence, R., Motelet, O., and Südholt, M. 2001. A formal definition of crosscuts. In *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (Kyoto, Japan), A. Yonezawa and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 2192. Springer-Verlag, Berlin, Germany, 170–186.

Erlingsson, Úlfar. and Schneider, F. B. 1999. SASI enforcement of security policies: A retrospective. In *Proceedings of the Workshop on New Security Paradigms* (Caledon Hills, Canada). ACM Press, New York, NY, USA, 87–95.

Erlingsson, Úlfar. and Schneider, F. B. 2000. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Oakland, California). IEEE Computer Society, Washington, DC, 246–255.

Evans, D. and Twyman, A. 1999. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (Oakland, CA). IEEE Computer Society, Washington, DC, 32–45.

Filman, R. E. and Friedman, D. P. 2005. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, MA, Chapter Aspect-Oriented Programming is Quantification and Obliviousness, 21–35.

Fiuczynski, M., Cody, Y., Grimm, R., and Walker, D. 2005. Patch(1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems* (Santa Fe, NM). USENIX, 91–96.

Gordon, A. and Fournet, C. 2003. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems 25,* 3 (May), 360–399.

Harper, R. and Stone, C. 1998. A type-theoretic interpretation of Standard ML. In *Proof, Language and Interaction: Essays in Honour of Robin Milner*. The MIT Press.

Harper, R. W. 2005. *Programming Languages: Theory and Practice*. In preparation, a draft can be obtained from `http://www.cs.cmu.edu/~rwh/plbook/`.

Hinze, R., Löh, A., and Oliveira, B. C. 2006. "Scrap your boilerplate" reloaded. In *Proceedings of the 8th International Symposium on Functional and Logic Programming* (Fuji Susono, Japan), P. Waldler and M. Hagiya, Eds. 24–26.

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

Jagadeesan, R., Jeffrey, A., and Riely, J. 2003b. A calculus of untyped aspect-oriented programs. In *Proceedings of the 17th European Conference on Object-Oriented Programming* (Darmstadt, Germany). Springer-Verlag, Berlin, Germany, 415–427.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming* (Budapest, Hungary). Springer-Verlag, 327–353.

Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O. 1999. Formally specified monitoring of temporal properties. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems* (York, UK). 114–121.

Läufer, K. and Odersky, M. 1992. An extension of ML with first-class abstract types. In *Proceedings of the SIGPLAN Workshop on ML and its Applications* (San Fransisco, California). 78–91.

Lee, D. K., Crary, K., and Harper, R. 2007. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France). ACM Press, New York, NY, USA, 173–184.

Lee, I., Kannan, S., Kim, M., Sokolsky, O., and Viswanathan, M. 1999. Run-time assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (Las Vegas, NV).

Lieberherr, K. J., Lorenz, D., and Ovlinger, J. 2003. Aspectual collaborations – combining modules and aspects. *The Computer Journal 46,* 5 (September), 542–565.

Masuhara, H., Kiczales, G., and Dutchyn, C. 2002. Compilation semantics of aspect-oriented programs. In *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages* (Lancaster, UK), G. T. Leavens and R. Cytron, Eds. 17–25.

Masuhara, H., Tatsuzawa, H., and Yonezawa, A. 2005. Aspectual caml: an aspect-oriented functional language. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming* (Tallinn, Estonia). ACM Press, New York, NY, USA, 320–330.

Matthews, D. 2005. Poly/ML. http://www.polyml.org/.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17,* 3, 348–375.

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

Peyton Jones, S., Vytiniotis, D., Weirich, S., and Washburn, G. 2006. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming* (Portland, OR). ACM Press, New York, NY, USA, 50–61.

Peyton Jones, S. L., Vytiniotis, D., Weirich, S., and Shields, M. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming 17,* 1 (Jan.), 1–82.

Pierce, B. C. and Turner, D. N. 1998. Local type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, CA). ACM Press, New York, NY, USA, 252–265.

Plotkin, G. D. 1970. A note on inductive generalization. In *Machine Intelligence*. Vol. 5. Edinburgh University Press, 153–163.

Plotkin, G. D. 1971. A further note on inductive generalization. In *Machine Intelligence*. Vol. 6. Edinburgh University Press, 101–124.

Pottier, F. and Régis-Gianas, Y. 2006. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, SC). ACM Press, New York, NY, 232–244.

Schneider, F. B. 2000. Enforceable security policies. *ACM Transactions on Information and Systems Security 3,* 1 (Feb.), 30–50.

Sheard, T. 2005. Putting Curry-Howard to work. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell* (Tallinn, Estonia). ACM, 74–85.

Shields, M. and Peyton Jones, S. 2002. Lexically scoped type variables. Microsoft Research. Available at http://research.microsoft.com/Users/simonpj/papers/scoped-tyvars.

Simonet, V. and Pottier, F. 2005. Constraint-based type inference for guarded algebraic data types. Tech. Rep. Research Report 5462, INRIA. Jan.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. 2005. Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th*

Conference on European Software Engineering held jointly with the 13th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (Lisbon, Portugal). ACM Press, New York, NY, USA, 166–175.

Sulzmann, M., Schrijvers, T., and Stuckey, P. J. 2006. Type inference for gadts via herbrand constraint abduction.

Tucker, D. B. and Krishnamurthi, S. 2003. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development* (Boston, MA). ACM Press, New York, NY, USA, 158–167.

Vytiniotis, D., Weirich, S., and Peyton Jones, S. 2006. Boxy types: inference for higher-rank types and impredicativity. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming* (Portland, OR). ACM Press, New York, NY, USA, 251–262.

Walker, D., Zdancewic, S., and Ligatti, J. 2003. A theory of aspects. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming* (Uppsala, Sweden). ACM Press, New York, NY, USA, 127–139.

Wand, M., Kiczales, G., and Dutchyn, C. 2003. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems 26,* 5, 890–910.

Wang, M., Chen, K., and Khoo, S.-C. 2006. On the pursuit of staticness and coherence. In *Proceedings of the 5th Workshop on Foundations of Aspect-Oriented Languages* (Bonn, Germany).

Washburn, G. and Weirich, S. 2005. Generalizing parametricity using information flow. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science* (Chicago, IL, USA). IEEE Computer Society Press, 62–71.