# PolyAML: A Polymorphic Aspect-oriented Functional Programming Language

Daniel S. Dantas David Walker
Department of Computer Science
Princeton University
{ddantas,dpw}@cs.princeton.edu

Geoffrey Washburn Stephanie Weirich
Department of Computer and Information Science
University of Pennsylvania
{geoffw,sweirich}@cis.upenn.edu

## Abstract

This paper defines PolyAML, a typed functional, aspect-oriented programming language. The main contribution of PolyAML is the seamless integration of polymorphism, run-time type analysis and aspect-oriented programming language features. In particular, PolyAML allows programmers to define type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points. PolyAML 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. These labels are organized in a tree structure such that a parent in the tree serves as a representative for all of its children. Type safety requires that the type of each child is less polymorphic than its parent type. Similarly, 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, Theory

***Keywords*** aspects-oriented programming, functional programming, ad-hoc polymorphism, type systems, type inference

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*ICFP'05* September 26–28, 2005, Tallinn, Estonia.
Copyright © 2005 ACM 1-59593-064-7/05/0009...$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 [21] 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 [7] 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 amongst many modules and it is 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 [16, 22, 23, 6, 14, 15, 3].
- Fourth, in some situations, the source code is unavailable or does not have the right to modify it and consequently manual insertion of function calls is out of the question. In these cases, aspects can be used as a robust form of external software patch [18].

To date there have 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 of developing such a language comes in constructing a typing discipline that is safe, yet 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 function being advised. However, many aspect-oriented programming tasks, including the profiling task mentioned above, are best handled by a single piece of advice that executes before (or after) 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, the otherwise generic profiling advice might be specialized so that on any call to a function with an integer argument, it tracks the distribution of calls with particular arguments. 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.

Another central consideration when designing a typed functional programming language is support for type inference. Here, both polymorphic pointcuts and run-time type analysis pose serious challenges to language designers. Polymorphic pointcuts prove difficult because they include quantified types. To use pointcuts as first-class objects, an important feature for building effective aspect-oriented libraries, it is necessary to weaken beyond ML's restriction on prenex polymorphism. Likewise, run-time type analysis is challenging because it refines types in the typing context and because each branch of a typecase statement may have a different type. Nevertheless, any extension of an ML-like like language with these features should be *conservative*. In other words, type inference should work as usual for ordinary ML programs; only when aspect-oriented features are involved should programmers be required to add typing annotations.

In this paper, we develop a typed functional programming language with polymorphic pointcuts, run-time type analysis and a conservative extension of ML's Hindley-Milner type inference algorithm. The language we define contains before and after advice and is *oblivious* [17]. In other words, programmers can add functionality to a program "after-the-fact" in the typical aspect-oriented style. 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 intermediate language with its own operational semantics. This strategy follows previous work by Walker, Zdancewic and Ligatti (WZL) [39], who define the semantics of a monomorphic language in this way. This translation helps to modularize the semantics for the source by unraveling complex source-language objects into simple, orthogonal intermediate language 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.

Our core language, though it builds directly on WZL, is itself an important contribution of our work. One of the novelties of the core language 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 whereas 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 the 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 (*i.e.*, child of an internal tree node).

The main contributions of this paper are as follows.

- We formally define a surface language 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 our language. 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 semantics of PolyAML by a translation into a typed core language, $\mathbb{F}_{\mathrm{A}}$. This core language defines primitive new notions of polymorphic labeled control flow points and polymorphic advice. We prove the core language is type safe, that the translation is type-preserving and therefore that the surface language is also safe.
- We have a complete prototype implementation that uses our type inference algorithm to infer types, translates to our intermediate language, and implements its operational semantics as an interpreter. This prototype is implemented in Standard ML of New Jersey and currently stands at approximately 5200 lines of code [1].

One of the limitations of this paper is that we do not consider *around* advice, one of the staples of AspectJ. We have two reasons for omitting around advice at this time. First, in a companion paper [10], we have defined an extended type system that prevents advice from interfering with the functional behavior of mainline code and thereby facilitates reasoning about aspect-oriented programs. This system of *harmless advice* is incompatible with around advice and we plan to merge it with the polymorphic programming constructs defined here. Second, around advice does not seem important for the security applications that we are most interested in. For now, around advice is beyond the scope of our work.

In the remaining sections of this paper, we define and analyze our new polymorphic, functional and aspect-oriented programming language PolyAML. Section 2 introduces the PolyAML syntax and informally describes the semantics through a series of examples. Section 3 describes the formal semantics of the PolyAML type system and type inference algorithm. Section 4 introduces the

[1] Available at `http://www.cs.princeton.edu/~ddantas/aspectml/`

```
(polytypes)       s ::= all ā.t
(pointcut type)  pt ::= (s1,s2)
(monotypes)       t ::= a | unit | string | stack |
                      | t1 -> t2 | pc pt
(trigger time)   tm ::= before | after
(terms)           e ::= x | () | c | e1e2 | let d in e
                      | stkcase e1 (p̄=>ē |_=> e2)
                      | typecase[t] a (t̄=>ē |_=> e)
                      | {f̄}:pt | any | e:t
(stack patterns)  p ::= x | nil | f::p
(frame patterns)  f ::= _ | e(x,y) | e(x:t,y)
(declarations)    d ::= rec f x = e
                      | rec f (x:t1):t2 = e
                      | advice tm e1 (x,y,z) = e2
                      | advice tm e1 (x:t,y,z) = e2
                      | case-advice tm e1 (x:t,y,z) = e2
```

**Figure 1.** Syntax of PolyAML

semantics of our polymorphic core calculus, $\mathbb{F}_A$. Section 5 shows have to give a semantics to PolyAML in terms of $\mathbb{F}_A$. Finally, Sections 6 and 7 describe related work and conclusions.

## 2. Programming with aspects

PolyAML is a polymorphic functional, aspect-oriented language based on the ML family of languages. Figure 1 presents its syntax. Here and elsewhere, we use over-bars to denote lists of syntactic objects: $\bar{x}$ refers to a sequence $x_1 \ldots x_n$, and $x_i$ stands for an arbitrary member of this sequence. Bold-faced text is used to indicate actual syntax, as opposed to meta-variables. We assume the usual conventions for variable binding and $\alpha$-equivalence of types and terms.

As in ML, the type structure of PolyAML is divided into *polytypes* and *monotypes*. The polytypes are normally written `all` $\bar{a}$`.t` where `t` is a monotype. However, when the list of binding type variables $\bar{a}$ is empty, we may abbreviate `all .t` as just `t`.

Here, and unlike in ML, the word "monotype" is a slight misnomer for the syntactic category `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 includes a pair of polytypes. We explain pointcut types in more detail later.

PolyAML expressions include variables, `x`, constants like unit, **`()`**, and strings, `c`, function application and let declarations. New functions may be declared in a let declaration. These functions may be polymorphic and they may or may not be annotated with their argument and result types. When annotations are omitted, PolyAML will infer these types. We assume it is easy to extend the language with other simple features such as integers, arithmetic and I/O, and we will make use of such things in our examples. Note that PolyAML does not include anonymous functions, a point we will address later.

The most interesting features of our language are pointcuts and advice. Advice in PolyAML 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 PolyAML, a pointcut designator has two parts, a *trigger time*, which may either be **`before`** or **`after`**, and a *pointcut* proper, which is a set of function names. The set of function names may be written out verbatim as **`{`**$\bar{f}$**`}`**, or, to indicate all functions, a programmer may use the set **`any`**.

Anonymous functions are nameless so it would be impossible to write explict advice for them. It would be reasonable to make **`any`** advice apply to anonymous functions. However, it might also be useful to write advice that applies just to anonymous functions using a distinguished pointcut. Finally, it could be argued that advice simplely should not apply to anonymous functions. Because these design choices do not present any technical difficulities for our framework, we have chosen to not address anonymous functions until we have more experience with programming in PolyAML.

In a larger language we would add a greater variety of pointcuts, including ones that corresponded to different actions in a module such as reading or writing reference cells and raising or catching exceptions, or different domains of interest, such as all function points in a particular module. We would also add a small language for specifying sets of function names, exceptions, etc., perhaps built on regular expressions.

Informally, the pointcut type, **`(`**$s_1$**`,`**$s_2$**`)`**, describes the I/O behaviour of a pointcut. In PolyAML, pointcuts are used to describe sets of functions, and as such $s_1$ and $s_2$ are conservative estimates of what the domains and ranges of those functions have in common. For example, if there are functions **`f`** and **`g`** with types **`string -> string`** and **`string -> unit`** respectively, we could give the pointcut **`{f,g}`** the pointcut type **`(string,all a.a)`**. This is because their domains are equal, so the least general polytype that describes them both is just **`string`**. However, they have different ranges, so the least general polytype that can be used to describe them both is **`all a.a`**. As we mentioned, pointcut types are conservative, so it would have also been fine to annotate the pointcut **`{f,g}`** with the pointcut type **`(all a.a,all a.a)`**. In the examples that follow, because the polytype **`all a.a`** is commonly used, we abbreviate it to **`T`**. The semantics of pointcut types is given precisely in Section 3.

The pointcut designator **`before {f}:`**`pt` represents the point in time immediately before executing a call to the function **`f`**. Likewise **`after {g,h}:`**`pt` represents the point in time immediately after executing either **`g`** or **`h`**. In both cases, the set is annotated with type information `pt` to aid type checking. First-class pointcuts, such as **`{g,h}`**, require that both their domain and range types be annotated. To make this easier when they appear in a pointcut designator, we introduce the syntactic sugar **`dom`** `s` and **`rng`** `s` for the pointcut types **`(`**`s`**`,T)`** and **`(T,`**`s`**`)`** respectively.

The most basic kind of advice has the form

**`advice`** `tm` $e_1$ **`(`**`x,y,z`**`) =`** $e_2$

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`** advice) or the result of function execution (in the case of **`after`** advice). The variable `x` may optionally be annotated with its type. 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. For our purposes, we will assume the metadata is a string corresponding to the function name as written in the source text. In other situations, it might include security information, such as the name of the code signer. Since our advice exchanges data with the designated control flow point, it must return a value with the same type as the first argument `x`.

A contrived example of using advice is the following code fragment for an implementation of factorial.

```
(* code *)
let rec fact x = if (x = 1) then 1
                 else x * fact (x-1) in
(* advice *)
let advice before {fact} : dom int
                    (arg, stk, name) =
  if (arg = 0) then 1 else arg
```

Here advice is used to correct the implementation of factorial, which did not correctly handle the case for $0! \triangleq 1$. We do not expect

that advice would be used like this in practice except when more significant patching is necessary or the source code is unavailable.

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. The type annotation **rng int** on the set **{f,g}** means that as an argument to a **before** pointcut designator it must be able to accept any type of data and as an argument to an **after** pointcut designator it may only accept data of type **int**.

```
(* code *)                        (* trace *)
let f x = x + 1 in                 entering g
let g x = if x then f 1            entering f
              else f 0 in          leaving f => 2
let h x = false in                 leaving g => 2
                                   entering h

(* advice *)
let advice before any (arg, stk, name) =
     print "entering "; println name; arg in
let advice after {f,g}: rng int
                 (arg, stk, name) =
     print ("leaving " ^ name ^ " => ");
     printint arg; println ""; arg
in
  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; quite the opposite, it is an *anti-unification* problem. Our type inference algorithm does not currently does not solve anti-unification problems, so we must require a typing annotation on pointcuts formed from sets of functions.

### 2.1 Run-time type analysis

We might also want the tracing routine to print not only the name of the function that is called, but also its argument. To do this, we need to analyze the type of the argument to the function. PolyAML makes this 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 first piece of advice is always triggered, the second piece of advice is only triggered when the function argument is an integer, and the third piece of advice is only triggered when the function argument is a boolean. (All advice that is applicable to a program point is triggered in the order in which the advice was declared.)

```
let advice before any (arg, stk, name) =
     print "entering "; println name;
     arg

in let case-advice
      before any (arg:int, stk, name) =
     print " with arg "; println (itos arg);
     arg
in let case-advice
      before any (arg:bool, stk, name) =
     print " with arg ";
     println (if arg then "true"
                     else "false");
     arg
in ...
```

This ability to conditionally trigger advice based on the type of the argument means that polymorphism is not parametric in PolyAML—programmers can analyze the types of values at runtime. However, without this ability we cannot implement this tracing aspect and other similar examples. For further flexibility, PolyAML also includes a typecase construct to analyze type variables directly. Below, to aid type checking, **[unit]** annotates the return type of the typecase expression.

```
let advice before any (arg:a, stk, name) =
  print "entering"; print name;
  print " with arg ";
  (typecase[unit] a of
     int  => println (itos arg)
   | bool => println (if arg then "true"
                             else "false")
   | _    => println " <unprintable>");
  arg
in ...
```

### 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 PolyAML 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**.

```
let
 advice before {f}: dom T (farg,fstk,fname) =
  (stkcase fstk of
   _::({g}: dom bool (garg, gname))::rest =>
     if garg then
       print "entering f from g(true)"
     else ()
  | other => ()); farg
in ...
```

A stack is a list of frames describing the execution context. The head of the stack contains information about the function that triggered the advice (e.g. **f** in the example above). Each frame on the stack describes a function in the context and can be matched by a frame pattern: either a wild-card _ or the pattern `e(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. The variable `x` is the argument of that function, and `y` is a string containing the name of the function.

A more sophisticated example of context analysis is to use an aspect to implement a stack-inspection-like security monitor for the program. 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:string -> string -> bool** determines whether the first argument (a function name) provides the capability for the second argument (another function name) to execute. We also assume **abort()** terminates the program.

```
let advice before any (arg1, stk, name1) =
   let rec walk y =
      stkcase y of
        nil => abort()
      | any (arg2, name2) :: rest =>
          if enables name2 name1 then ()
          else walk rest
   in walk stk; arg1
```

However, a subtle point that we caught only we tested this example with our implementation, is that the **any** pointcut is very difficult to use. In particular, the above program will always diverge, because 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, that will disable all advice while e is evaluated. The advice could then be rewritten as

```
let advice before any (arg1, stk, name1) =
   let rec walk y =
      stkcase y of
        nil => abort()
      | any (arg2, name2) :: rest =>
          if enables name2 name1 then ()
          else walk rest
   in disable (walk stk); arg1
```

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

```
let rec walk name1 y =
  stkcase y of
    nil => abort()
  | any (arg2, name2) :: rest =>
      if enables name2 name1 then ()
      else walk rest in
let advice before
      (any except {walk,enables} : dom T)
      (arg1, stk, name1) =
   in walk name1 stk; arg1
```

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 4.

### 2.3 First-class pointcuts

The last interesting feature of our language is the ability to use pointcuts as first-class objects. 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. To give one simple example, consider the "f within g" pattern presented in one of the previous examples. This is a very common idiom; in fact, AspectJ has a special pointcut designator for specifying it. In PolyAML, assuming tuples for the moment, 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.

```
let rec within
        ((fpc,gpc,body) : pc (T, T) *
                          dom bool *
                          (bool -> a)) =
  let advice before fpc (farg,fstk,fname) =
       (stkcase fstk of
          _ :: gpc (garg, gname) :: rest =>
             body garg
        | _ => ()); farg
  in ()
in let rec entering x =
  if x then (println "entering f from g"; x)
  else x
in
  within ({f}:(T,T), {g}: dom bool, entering)
```

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 is that pointcut types include polytypes; polytypes cannot be determined via unification. In the next section, we formally describe how to reconcile the Hindley-Milner type system with first-class pointcuts using type annotations.

## 3. Type inference

The type system of PolyAML is carefully designed to permit efficient type inference with an algorithm that is an extension of Damas and Milner's Algorithm $\mathcal{W}$ [8]. Because the algorithm behaves exactly the same as ML for ML terms, all terms that do not include aspects or type analysis will type check without annotation, as they do in ML.

Type inference for PolyAML is specified by the judgments and rules that appear in Figure 4. The difficult part in the design of PolyAML's type system is reconciling type inference with first-class pointcuts, polymorphic pointcuts, and run-time type analysis. In general, we have tried to balance simplicity and the number of required user annotations. It should be easy for the user to predict whether an annotation will be necessary. As we gain more experience with our implementation, we will be able to better gauge how much of a burden the annotations are. In Section 3.4, we discuss extensions of the type system that could reduce the number of required annotations.

### 3.1 First-class polymorphic pointcuts

First-class polymorphic pointcuts are problematic for type inference because they inject polytypes in the syntax of monotypes, with the type **pc (**$s_1$**,** $s_2$**)**. Higher-order unification, which is known to be undecidable, would be necessary to guess the appropriate polytypes. Instead, whenever two pointcut types are compared by the unification algorithm, it requires that the polytypes abstract exactly the same type variables (up to $\alpha$-conversion) [27].

Figure 2 describes our unification algorithm and Figure 3 presents some useful auxiliary definitions. Unification variables are notated by $X, Y, Z, \ldots$ and are only introduced by the type inference algorithm. Unification variables are distinct from (rigid) programmer-supplied type variables a. Our term annotation rule behaves like that of Standard ML [29] rather than Objective Caml [24]: Type-variables occurring in annotations are assumed to be bound by their enclosing scope, rather than acting like unification variables. This design choice is investigated in more detail by Shields and Peyton-Jones [33].

We use $\Theta$ to refer to an idempotent, ever-growing substitution of monotypes for unification-variables. Our unification judgment $\Theta \vdash t_1 = t_2 \Rightarrow \Theta'$ is read as

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

$$\frac{}{\Theta \vdash t = t \Rightarrow \Theta} \qquad \frac{X \in \mathrm{dom}(\Theta) \qquad \Theta \vdash \Theta(X) = t \Rightarrow \Theta'}{\Theta \vdash X = t \Rightarrow \Theta'}$$

$$\frac{X \notin \mathrm{dom}(\Theta) \qquad X \notin \mathrm{FTV}(t)}{\Theta \vdash X = t \Rightarrow \Theta, t/X} \qquad \frac{\Theta \vdash X = t \Rightarrow \Theta'}{\Theta \vdash t = X \Rightarrow \Theta'}$$

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

$$\frac{\Theta \vdash t_1 = t_3 \Rightarrow \Theta' \qquad \Theta' \vdash t_2 = t_4 \Rightarrow \Theta''}{\Theta \vdash \texttt{pc (all } \overline{a}\texttt{.}t_1\texttt{, all } \overline{b}\texttt{.}t_2\texttt{)} = \texttt{pc (all } \overline{a}\texttt{.}t_3\texttt{, all } \overline{b}\texttt{.}t_4\texttt{)} \Rightarrow \Theta''}$$

**Figure 2.** Unification

$$\begin{array}{rcl} \Gamma & ::= & \cdot \mid \Gamma, x :: t \mid \Gamma, x : t \\ \Phi & ::= & \cdot \mid \Phi, x \\ \Delta & ::= & \cdot \mid \Delta, a \end{array}$$

$$\begin{array}{rcl} \pi(\texttt{before}, (s_1, s_2)) & \triangleq & s_1 \\ \pi(\texttt{stk}, (s_1, s_2)) & \triangleq & s_1 \\ \pi(\texttt{after}, (s_1, s_2)) & \triangleq & s_2 \end{array}$$

$$\frac{\overline{X} \text{ fresh} \qquad \Theta \vdash t_1[\overline{X}/\overline{a}] = t_2 \Rightarrow \Theta'}{\Theta \vdash \texttt{all } \overline{a}\texttt{.}t_1 \preceq \texttt{all } \overline{b}\texttt{.}t_2 \Rightarrow \Theta'}$$

$$gen(\Gamma, t) \triangleq \texttt{all } \overline{a}\texttt{.}t[\overline{a}/\overline{X}] \quad \text{where } \overline{X} = \mathrm{FTV}(t) - \mathrm{FTV}(\Gamma) \text{ and } \overline{a} \text{ fresh}$$

**Figure 3.** Auxiliary definitions

> "With input substitution $\Theta$, types $t_1$ and $t_2$ unify producing the extended substitution $\Theta'$."

That is, the substitution $\Theta$ is extended to produce a new substitution $\Theta'$ so that $\Theta'(t_1) = \Theta'(t_2)$. Furthermore, $\Theta'$ is the most general unifier for these monotypes. In this and in other judgments, we use the convention that the outputs of the algorithm appear to the right of $\Rightarrow$ symbol.

To provide flexibility with user annotations, there are two different forms of typing judgment for expressions (see Figure 4). In these judgements, $\Theta$ is an input substitution, $\Gamma$ the term variable context, $\Delta$ the type variable context, and $\Phi$ the set of function names currently in scope. The first form is the standard judgment, $\Theta; \Delta; \Phi; \Gamma \vdash e \Rightarrow t; \Theta'$, and is read as

> "Given the input substitution $\Theta$ and the contexts $\Delta$,$\Phi$, and $\Gamma$, the term $e$ has type $t$ and produces substitution $\Theta'$, possibly requiring unification to determine $t$."

The second judgment is a simple form of local type inference, $\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} e \Rightarrow t; \Theta'$, and is read as

> "Given the input substitution $\Theta$ and the contexts $\Delta$,$\Phi$, and $\Gamma$, the term $e$ has type $t$, as specified by the programmer, and produces substitution $\Theta'$."

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 (monomorphic) type was annotated or certain constants. To propagate the type annotation on variables, the context, $\Gamma$ contains two different assertions depending

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

$$\frac{\Delta \vdash t_2 \qquad \Theta; \Delta; \Phi; \Gamma \vdash e \Rightarrow t_1; \Theta' \qquad \Theta' \vdash t_1 = t_2 \Rightarrow \Theta''}{\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} e\texttt{:}t_2 \Rightarrow t_2; \Theta''}$$

$$\frac{x :: t \in \Gamma}{\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} x \Rightarrow t; \Theta} \qquad \frac{}{\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \texttt{()} \Rightarrow \texttt{unit}; \Theta}$$

$$\frac{}{\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} c \Rightarrow \texttt{string}; \Theta}$$

$$\frac{}{\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \texttt{any} \Rightarrow \texttt{pc (all } a\texttt{.}a\texttt{, all } a\texttt{.}a\texttt{)}; \Theta}$$

$$\frac{\begin{array}{c}\Delta \vdash s_1 \\ \Delta \vdash s_2 \qquad \forall i \qquad f_i \in \Phi \qquad \Gamma(f_i) = \texttt{all } \overline{a}\texttt{.}t_{1,i} \texttt{ -> } t_{2,i} \\ \Theta_{i-1} \vdash s_1 \preceq \texttt{all } \overline{a}\texttt{.}t_{1,i} \Rightarrow \Theta_i' \qquad \Theta_i' \vdash s_2 \preceq \texttt{all } \overline{a}\texttt{.}t_{2,i} \Rightarrow \Theta_i\end{array}}{\Theta_0; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} \texttt{\{}\overline{f}\texttt{\}:(}s_1\texttt{,}s_2\texttt{)} \Rightarrow \texttt{pc (}s_1\texttt{,}s_2\texttt{)}; \Theta_n}$$

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

$$\frac{\Theta; \Delta; \Phi; \Gamma \vdash^{\mathrm{loc}} e \Rightarrow t; \Theta'}{\Theta; \Delta; \Phi; \Gamma \vdash e \Rightarrow t; \Theta'} \qquad \frac{\Gamma(x) = \texttt{all } \overline{a}\texttt{.}t \qquad \overline{X} \text{ fresh}}{\Theta; \Delta; \Phi; \Gamma \vdash x \Rightarrow t[\overline{X}/\overline{a}]; \Theta}$$

$$\frac{\begin{array}{c}\Theta_1; \Delta; \Phi; \Gamma \vdash e_1 \Rightarrow t_1; \Theta_2 \qquad \Theta_2; \Delta; \Phi; \Gamma \vdash e_2 \Rightarrow t_2; \Theta_3 \\ X \text{ fresh} \qquad \Theta_3 \vdash t_1 = t_2 \texttt{ -> } X \Rightarrow \Theta_4\end{array}}{\Theta_1; \Delta; \Phi; \Gamma \vdash e_1 e_2 \Rightarrow X; \Theta_4}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Phi; \Gamma \vdash e \Rightarrow \texttt{stack}; \Theta_0 \qquad \Theta_0; \Delta; \Phi; \Gamma \vdash e' \Rightarrow t; \Theta_0'' \\ \forall i \qquad \Theta_{i-1}''; \Delta; \Phi; \Gamma \vdash p_i \Rightarrow \Theta_i; \Delta_i; \Gamma_i \\ \Theta_i; \Delta, \Delta_i; \Phi; \Gamma, \Gamma_i \vdash e_i \Rightarrow t_i; \Theta_i' \qquad \Theta_i' \vdash t_i = t \Rightarrow \Theta_i''\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \texttt{stkcase } e \texttt{ (}\overline{p\texttt{=>}e}\texttt{ |\_=> } e'\texttt{)} \Rightarrow t; \Theta_n''}$$

$$\frac{\begin{array}{c}a \in \Delta \\ \Delta \vdash t \qquad \Theta; \Delta; \Phi; \Gamma \vdash e \Rightarrow t; \Theta_0 \qquad \forall i \qquad \Delta_i = \mathrm{FTV}(t_i) - \Delta \\ a \notin \mathrm{FTV}(t_i) \qquad \Theta_{i-1}; \Delta, \Delta_i; \Phi; \Gamma\langle t_i/a\rangle \vdash e_i[t_i/a] \Rightarrow t_i'; \Theta_i' \\ \Theta_i' \vdash t_i' = t[t_i/a] \Rightarrow \Theta_i\end{array}}{\Theta; \Delta; \Phi; \Gamma \vdash \texttt{typecase[}t\texttt{] } a \texttt{ (}\overline{t\texttt{=>}e}\texttt{ |\_=> } e\texttt{)} \Rightarrow t; \Theta_n}$$

$$\frac{\Theta; \Delta; \Phi; \Gamma \vdash d \Rightarrow \Theta'; \Phi'; \Gamma' \qquad \Theta'; \Delta; \Phi, \Phi'; \Gamma, \Gamma' \vdash e \Rightarrow t; \Theta''}{\Theta; \Delta; \Phi; \Gamma \vdash \texttt{let } d \texttt{ in } e \Rightarrow t; \Theta''}$$

**Figure 4.** Type inference for expressions

on whether types are inferred via unification ($x : s$) or known ($x :: s$). We use the notation $\Gamma(x) = s$ to refer to either $x : s \in \Gamma$ or $x :: s \in \Gamma$.

The typing rule for advice declarations (in Figure 5) states that the type of a pointcut must be determinable using the local type judgment. That way, the inference algorithm need not use 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. Below we use the notation $\pi(tm, pt)$ 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. There is also special trigger time, **stk**, used only by the type inference algorithm that is essentially equivalent to **before**. This notation is defined in Figure 3.

The typing rule for **case-advice** is similar to that for advice. Note that **case-advice** requires a typing annotation on $x$, the first parameter to the advice. The user employs the annotation to drive the underlying run-time type analysis.

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

$$\frac{\begin{array}{c}\overline{\mathtt{a}} = (\mathrm{FTV}(\mathtt{t}_1) \cup \mathrm{FTV}(\mathtt{t}_2)) - \Delta \\ \Theta;\Delta,\overline{\mathtt{a}};\Phi,\mathtt{f};\Gamma,\mathtt{f} :: \mathtt{t}_1 \texttt{ -> } \mathtt{t}_2, \mathtt{x} :: \mathtt{t}_1 \vdash \mathtt{e}_1 \Rightarrow \mathtt{t}_3;\Theta' \\ \Theta' \vdash \mathtt{t}_2 = \mathtt{t}_3 \Rightarrow \Theta'' \qquad \mathtt{s} = \texttt{all } \overline{\mathtt{a}}\texttt{.}\mathtt{t}_1 \rightarrow \mathtt{t}_2\end{array}}{\Theta;\Delta;\Phi;\Gamma \vdash \texttt{rec f (x:}\mathtt{t}_1\texttt{):}\mathtt{t}_2\texttt{ = }\mathtt{e}_1 \Rightarrow \Theta'';\cdot,\mathtt{f};\cdot,\mathtt{f} :: \mathtt{s}}$$

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

$$\frac{\begin{array}{c}\Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \mathtt{e}_1 \Rightarrow \texttt{pc } \mathtt{pt};\Theta' \qquad \pi(\mathtt{tm},\mathtt{pt}) = \texttt{all } \overline{\mathtt{a}}\texttt{.}\mathtt{t}_1 \\ \Theta';\Delta,\overline{\mathtt{a}};\Phi;\Gamma,\mathtt{x} :: \mathtt{t}_1,\mathtt{y} :: \texttt{stack},\mathtt{z} :: \texttt{string} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2;\Theta'' \\ \Theta'' \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta'''\end{array}}{\Theta;\Delta;\Phi;\Gamma \vdash \texttt{advice tm }\mathtt{e}_1\texttt{ (x,y,z) = }\mathtt{e}_2 \Rightarrow \Theta''';\cdot;\cdot}$$

$$\frac{\begin{array}{c}\Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \mathtt{e}_1 \Rightarrow \texttt{pc } \mathtt{pt};\Theta' \\ \pi(\mathtt{tm},\mathtt{pt}) = \texttt{all } \overline{\mathtt{a}}\texttt{.}\mathtt{t}_3 \qquad \overline{\mathtt{a}} = \mathrm{FTV}(\mathtt{t}_3) - \Delta \\ \Theta';\Delta,\overline{\mathtt{a}};\Phi;\Gamma,\mathtt{x} :: \mathtt{t}_3,\mathtt{y} :: \texttt{stack},\mathtt{z} :: \texttt{string} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2;\Theta'' \\ \Theta'' \vdash \mathtt{t}_3 = \mathtt{t}_2 \Rightarrow \Theta'''\end{array}}{\Theta;\Delta;\Phi;\Gamma \vdash \texttt{advice tm }\mathtt{e}_1\texttt{ (x:}\mathtt{t}_3\texttt{,y,z) = }\mathtt{e}_2 \Rightarrow \Theta''';\cdot;\cdot}$$

$$\frac{\begin{array}{c}\Delta' = \mathrm{FTV}(\mathtt{t}_1) - \Delta \qquad \Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \mathtt{e}_1 \Rightarrow \texttt{pc } \mathtt{pt};\Theta' \\ \Theta';\Delta,\Delta';\Phi;\Gamma,\mathtt{x} :: \mathtt{t}_1,\mathtt{y} :: \texttt{stack},\mathtt{z} :: \texttt{string} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t}_2;\Theta'' \\ \Theta'' \vdash \mathtt{t}_1 = \mathtt{t}_2 \Rightarrow \Theta'''\end{array}}{\Theta;\Delta;\Phi;\Gamma \vdash \texttt{case-advice tm }\mathtt{e}_1\texttt{ (x:}\mathtt{t}_1\texttt{,y,z) = }\mathtt{e}_2 \Rightarrow \Theta''';\cdot;\cdot}$$

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

$$\frac{}{\Theta;\Delta;\Phi;\Gamma \vdash \texttt{nil} \Rightarrow \Theta;\cdot;\cdot}$$

$$\frac{}{\Theta;\Delta;\Phi;\Gamma \vdash \mathtt{x} \Rightarrow \Theta;\cdot;\mathtt{x} :: \texttt{stack}}$$

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

$$\frac{\begin{array}{c}\Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \mathtt{e} \Rightarrow \texttt{pc } \mathtt{pt};\Theta' \\ \pi(\texttt{stk},\mathtt{pt}) = \texttt{all } \overline{\mathtt{a}}\texttt{.}\mathtt{t} \qquad \Theta';\Delta;\Phi;\Gamma \vdash \mathtt{p} \Rightarrow \Theta'';\Delta';\Gamma'\end{array}}{\Theta;\Delta;\Phi;\Gamma \vdash \mathtt{e}\texttt{(x,z)::}\mathtt{p} \Rightarrow \Theta'';\Delta',\overline{\mathtt{a}};\Gamma',\mathtt{x}:\mathtt{t}_2,\mathtt{z}:\texttt{string}}$$

$$\frac{\begin{array}{c}\Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \mathtt{e} \Rightarrow \texttt{pc } \mathtt{pt};\Theta' \qquad \pi(\texttt{stk},\mathtt{pt}) = \texttt{all } \overline{\mathtt{a}}\texttt{.}\mathtt{t} \\ \overline{\mathtt{a}} = \mathrm{FTV}(\mathtt{t}) - \Delta \qquad \Theta';\Delta;\Phi;\Gamma \vdash \mathtt{p} \Rightarrow \Theta'';\Delta';\Gamma'\end{array}}{\Theta;\Delta;\Phi;\Gamma \vdash \mathtt{e}\texttt{(x:t,z)::}\mathtt{p} \Rightarrow \Theta'';\Delta',\overline{\mathtt{a}};\Gamma',\mathtt{x}:\mathtt{t},\mathtt{z}:\texttt{string}}$$

**Figure 5.** Type inference for declarations and patterns

### 3.2 Polymorphic pointcuts

Another tricky part of the type system is the formation of pointcuts from sets of function names. Only let-bound function names may be part of a pointcut. To ensure this constraint, the $\Phi$ component of the typing judgments is a set of function names that are currently in scope. When a pointcut is formed from a set of functions, each of those functions must be a member of $\Phi$.

Now consider the rule for pointcuts constructed from sets of functions in Figure 4. The domain type of each function in the set must be at most as polymorphic as the first polytype in the pointcut type. Similarly, the range type of each function in the set must be at most as polymorphic as the second polytype in the pointcut type. The relation $\Theta \vdash \mathtt{s}_1 \preceq \mathtt{s}_2 \Rightarrow \Theta'$ (defined in Figure 3) and is read as

> "Given input substitution $\Theta$, polytype $\mathtt{s}_1$ can be shown to be more general than polytype $\mathtt{s}_2$, by producing an extended substitution $\Theta'$."

By more general, we mean that there exists an instantiation for some of the quantified variables in $\Theta'(\mathtt{s}_1)$ that will make it equal to $\Theta'(\mathtt{s}_2)$. This is the same definition as in ML. To simplify inference, the polytypes `(`$\mathtt{s}_1$`,`$\mathtt{s}_2$`)` must be annotated on the set by the user. Because of this annotation, the expression always has a local type.

### 3.3 Run-time 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. As the rule in Figure 4 shows, if the expression should be of type $\mathtt{t}$ then a branch for type $\mathtt{t}_i$ may be of type $\mathtt{t}[\mathtt{t}_i/\mathtt{a}]$. This substitution is sound because if the branch is executed, then the type $\mathtt{a}$ is the same as the type $\mathtt{t}_i$. When type checking each branch, types in the context may also change. Above, the notation $\Gamma\langle\mathtt{t}_i/\mathtt{a}\rangle$ means that type $\mathtt{t}_i$ is substituted for the variable $\mathtt{a}$ *only* in local assumptions $\mathtt{x} :: \mathtt{s}$. Other types remain the same.

Note that we must not allow refinement in inferred parts of the context (assumptions of the form $\mathtt{a} : \mathtt{s}$) because, even with the return type annotation on **`typecase`**, there are some expressions with no principal type. For example, in the following code fragment,

```
let rec h (x:a) =
  let rec g (y) = typecase[int] a of
                    int => y + 1
                  | _   => 2
  in g
in ...
```

we can assign the types **`all a. a -> a -> int`** or **`all 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. This issue has appeared before in type inference systems for Generalized Algebraic Datatypes (also called Guarded Recursive Datatypes) [31, 34, 35].

### 3.4 Extensions to PolyAML

One property of our type system is simplicity. It is easy for the user to understand where annotations are required. However, practice may show that this simplicity comes at a price: those annotations may be burdensome to users. Therefore, we plan to use our implementation to explore a number of potential extensions and modifications of our type system. However, none of the following extensions are currently part of our implementation.

First, a few specialized rules may eliminate a number of user annotations. For example, if all of the functions in a pointcut have the same type, no annotation would be necessary.

$$\frac{\forall \mathtt{i} \qquad \mathtt{f}_i \in \Phi \qquad \Gamma(\mathtt{f}_i) = \texttt{all } \overline{\mathtt{a}}\texttt{.}\mathtt{t}_1 \texttt{ -> } \mathtt{t}_2}{\Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \texttt{\{}\overline{\mathtt{f}}\texttt{\}} \Rightarrow \texttt{pc (all }\overline{\mathtt{a}}\texttt{.}\mathtt{t}_1\texttt{,all }\overline{\mathtt{a}}\texttt{.}\mathtt{t}_2\texttt{)};\Theta}$$

Also, we could always try the type **`pc (all a.a,all a.a)`**, if no type has been supplied by the user.

$$\frac{\forall \mathtt{i} \qquad \mathtt{f}_i \in \Phi}{\Theta;\Delta;\Phi;\Gamma \vdash^{\mathrm{loc}} \texttt{\{}\overline{\mathtt{f}}\texttt{\}} \Rightarrow \texttt{pc (all a.a,all a.a)};\Theta}$$

Looking at advice declarations, if local inference fails, we could allow unification for the determination of pointcut types by requiring them to be monomorphic.

$$\frac{\begin{array}{c}\Theta_0;\Delta;\Phi;\Gamma \vdash \mathtt{e}_1 \Rightarrow \mathtt{t}_0;\Theta_1 \\ \Theta_1 \vdash \mathtt{t}_0 = \mathbf{pc}\ (\mathtt{t}_1,\mathtt{t}_2) \Rightarrow \Theta_2 \qquad \pi(\mathtt{tm},(\mathtt{t}_1,\mathtt{t}_2)) = \mathtt{t} \\ \Theta_2;\Delta;\Phi;\Gamma,\mathtt{x}::\mathtt{t},\mathtt{y}::\mathbf{stack},\mathtt{z}::\mathbf{string} \vdash \mathtt{e}_2 \Rightarrow \mathtt{t};\Theta_3\end{array}}{\Theta_0;\Delta;\Phi;\Gamma \vdash \mathbf{advice}\ \mathtt{tm}\ \mathtt{e}_1\ (\mathtt{x},\mathtt{y},\mathtt{z}) = \mathtt{e}_2 \Rightarrow \Theta_3;\cdot;\cdot}$$

Besides these minor tweaks, we also plan to explore more significant modifications. First, we may get more mileage out of our annotations by using a more sophisticated form of local type inference, such as *bidirectional type inference* [32, 30] or *boxy types* [38].

More ambitiously, if we can reconcile anti-unification constraints with unification, a number of annotations may be eliminated. Not only could we drop the annotation on the formation of pointcuts from sets of function names, but might also be able to drop the annotation on the return type of **typecase**. As long as there are multiple branches, we could use anti-unification to determine the return type of **typecase** unambiguously. For example, in the following code fragment

```
let rec f (x : a) = typecase a of int => 3
```

It is impossible to determine whether **f** should be type **all a.a -> int** or **all a.a -> a**. However, for the following code fragment

```
let rec g (x : a) = typecase a of int => 3
                                | _ => 4
```

We can unambiguously give **g** the type **all a.a -> int**.

### 3.5 Future work: A declarative specification

Some users of ML rely on the *declarative* nature of the HM type system, which elides the uses of unification [28]. We are working to develop a similar declarative specification for our type system.

Unfortunately, the rule for pointcuts has undesirable interactions with the declarative specification of HM-style type inference. This rule uses the function $\mathtt{f_i}$ without instantiation, breaking the following property: if $\Delta;\Phi;\Gamma \vdash \mathtt{e} : \mathtt{t}$ and $\Gamma'$ is a more general context than $\Gamma$, then $\Delta;\Phi;\Gamma' \vdash \mathtt{e} : \mathtt{t}$. This property does not hold because a more general type for a function $\mathtt{f_i}$ 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.

For example, the following term type checks according to the rules of such a 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, **all a.a -> a**

```
let rec f x = x in {f}:(string,string)
```

We believe this term should not type check, as, given the definition of **f**, the user should expect that it has type **all a.a -> a** and might be used at many types. Our type inference algorithm concurs. 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, but we have not proved this fact. Happily, even though we are changing the specification for pure ML terms, this change would not invalidate any ML programs. It merely cuts down the number of alternate typing derivations for terms that use let. The derivation that uses the most general type is still available.

## 4. Polymorphic core calculus

In the previous section, we defined the syntax and static semantics for PolyAML. 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 possibly of run-time type analysis and extraction of metadata from the call stack. Rather than attempt to specify all of these features directly, creating a horrendous mess, we specify the operational semantics in stages. First, we show how to compile the high-level constructs into a core language, called $\mathbb{F}_A$. The translation breaks down complex high-level objects into substantially simpler, orthogonal core-level objects. This core language is also typed and the translation is type-preserving. Second, we define an operational semantics for the core language. Since we have proven that the $\mathbb{F}_A$ type system is sound and the translation from the source is type-preserving, the PolyAML is safe.

Our core language differs from the PolyAML 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 language—for example, not including 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 language. However, in exchange, the core language is much more expressive than the source language.

Because $\mathbb{F}_A$ is so expressive, we can easily experiment with the source language, 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 PolyAML, we have made numerous changes. Fortunately, for the most part, we have not had to make corresponding few 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.

The core language $\mathbb{F}_A$ is an extension of the core language from WZL with polymorphic labels, polymorphic advice, and run-time type analysis. It also improves upon the semantics of context analysis. In this section, we sketch the semantics of $\mathbb{F}_A$, but due to lack of space, the complete semantics appears in the companion technical report [11]. In Section 5, we sketch the translation from PolyAML to $\mathbb{F}_A$.

### 4.1 The semantics of explicit join points

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.

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

The basis of $\mathbb{F}_A$ is the λ-calculus with unit, strings and n-tuples. If $\overline{e}$ is a sequence of expressions $e_1 \ldots e_n$ for $n \ge 2$, 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 language, 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, $\ell$, 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 language *join points*. For example, in the addition expression $v_1 + \ell[\overline{\tau}][\![e_2]\!]$, after $e_2$ has been evaluated to a value $v_2$, evaluation of the resulting subterm $\ell[\overline{\tau}][\![v_2]\!]$ causes any advice associated with $\ell$ to be triggered.

Here, unlike in WZL, the labels form a tree-shaped hierarchy. The top label in the hierarchy is $\mathcal{U}$. All other labels $\ell$ sit somewhere below $\mathcal{U}$. If $\ell_1 \leq \ell_2$ then $\ell_1$ sits below $\ell_2$ in the hierarchy. The expression $\mathbf{new}\ \overline{\alpha}.\tau \leq e$ evaluates $e$, obtaining a label $\ell_2$, and generates a new label $\ell_1$ such that $\ell_1 \leq \ell_2$. This label structure closely resembles the label hierarchy defined by Bruns et al. for their (untyped) $\mu$ABC calculus [4].

Our first class labels can then 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.

Advice is a computation that exchanges data with a particular join point, making it similar to a function. Note that advice in $\mathbb{F}_A$ (written $\{e_1.\overline{\alpha}x{:}\tau \rightarrow e_2\}$) is first-class. The type variables $\overline{\alpha}$ and term variable $x$ are bound in the body of the advice $e_2$, and the expression $e_1$ is a label-set that describes when the advice is triggered. For example, the advice $\{\{\overline{\ell}\}.x{:}\mathsf{int} \rightarrow e\}$ is triggered when control-flow reaches a join point marked with $\ell_1$, provided $\ell_1$ is a descendent of a label in the set $\{\overline{\ell}\}$. If this advice has been installed in the program's dynamic environment, $v_1 + \ell_1[][\![v_2]\!]$ evaluates to $v_1 + e[v_2/x]$.

When labels are polymorphic, both types and values are exchanged between labeled control-flow points and advice. For instance, if $\ell_1$ is a polymorphic label capable of marking a control-flow point with any type, we might write $v_1 + \ell_1[\mathsf{int}][\![v_2]\!]$. In this case, if the advice $\{\{\ell_1\}.\alpha x{:}\alpha \rightarrow e\}$ has been installed, then the previous expression evaluates to $v_1 + e[\mathsf{int}/\alpha][v_2/x]$. Since $\mathcal{U}$ sits at the top of the label hierarchy, once installed, advice with the form $\{\{\mathcal{U}\}.\alpha x{:}\alpha \rightarrow e\}$ is executed at every labeled control-flow point.

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.

***Operational semantics*** The operational semantics must keep track of both the labels that have been generated and the advice that has been installed. An allocation-style semantics keeps track of the set $\Sigma$ of labels allocated so far (and their associated types) and $A$, an ordered list of installed advice. The main operational judgment has the form $\Sigma; A; e \mapsto \Sigma'; A'; e'$. To describe the operational semantics, we use the following syntax for values $v$ and evaluation contexts $E$:

$$
\begin{array}{l}
v ::= \langle\rangle \mid \lambda x{:}\tau.e \mid \langle \overline{v} \rangle \mid \Lambda\alpha.e \mid \ell \mid \{v.x{:}\tau \rightarrow e\} \\
E ::= [\,] \mid E\ e \mid v\ E \mid E[\tau] \mid \langle E, \ldots, e \rangle \mid \langle v, \ldots, E \rangle \\
\quad \mid \mathbf{let}\ \langle \overline{x} \rangle = E\ \mathbf{in}\ e \mid E[\overline{\tau}][\![e]\!] \mid v[\overline{\tau}][\![E]\!] \mid \Uparrow E \mid \{E.\overline{\alpha}x{:}\tau \rightarrow e\} \\
\quad \mid \mathbf{new}\ \overline{\alpha}.\tau \leq E
\end{array}
$$

Evaluation contexts give the core aspect calculus a call-by-value, left-to-right evaluation order, but that choice is orthogonal to the design of the language. Auxiliary rules with the form $\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'$ give the primitive $\beta$-reductions for expressions in the language. The main points of interest have been described informally through examples in the previous section and are included in the excerpted rules in Figure 6.

A third judgment form $\Sigma; A; \ell; \tau \Rightarrow v$ describes, given a particular label $\ell$ marking a control-flow point, and type $\tau$ 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. The result of this advice composition process is a function $v$ that may be applied to a value with type $\tau$. This judgment (advice composition) is described by three rules shown in Figure 6. The first composition rule returns the identity function when no advice is available. The other rules examine the advice at the head of the advice heap. If the label $\ell$ 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 $\ell$ 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.

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

$$\frac{}{\Sigma; A; \{\overline{\ell_1}\} \cup \{\overline{\ell_2}\} \mapsto_\beta \Sigma; A; \{\overline{\ell_1\ell_2}\}}$$

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

$$\frac{}{\Sigma; A; \Uparrow v \mapsto_\beta \Sigma; v, A; \langle\rangle}$$

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

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

$$\frac{\ell{:}\overline{\alpha}.\tau \leq \ell' \in \Sigma \qquad \Sigma; A; \ell; \tau[\overline{\tau}/\overline{\alpha}] \Rightarrow v'}{\Sigma; A; \ell[\overline{\tau}][\![v]\!] \mapsto_\beta \Sigma; A; v'\,v}$$

**Advice composition $\Sigma; A; \ell; \tau \Rightarrow e$**

$$\frac{}{\Sigma; \cdot; \ell; \tau \Rightarrow \lambda x{:}\tau.x}$$

$$\frac{\Sigma; A; \ell; \tau_2 \Rightarrow v_2 \qquad \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \exists\overline{\tau}.\tau_2 = \tau_1[\overline{\tau}/\overline{\alpha}]}{\Sigma; A, \{\{\overline{\ell}\}.\overline{\alpha}x{:}\tau_1 \rightarrow e\}; \ell; \tau_2 \Rightarrow \lambda x{:}\tau_2.v_2(e[\overline{\tau}/\overline{\alpha}])}$$

$$\frac{\Sigma; A; \ell; \tau_2 \Rightarrow v_2 \qquad \Sigma \vdash \ell \not\leq \ell_i}{\Sigma; A, \{\{\overline{\ell}\}.\overline{\alpha}x{:}\tau_1 \rightarrow e\}; \ell; \tau_2 \Rightarrow v_2}$$

**Figure 6.** Operational semantics excerpt for $\mathbb{F}_A$

***Type system*** 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 $\tau$, where free type variables appear in $\Delta$ and the types of term variables and labels appear in $\Gamma$. The typing rules for this judgment appear in Figure 7.

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, which have the form $\overline{\alpha}.\tau\ \mathsf{label}$. Such labels may mark control-flow points containing values of any type $\tau$, where free variables $\overline{\alpha}$ are replaced by other types $\overline{\tau}$. For example, a label $\ell$ with the type $\alpha.\alpha\ \mathsf{label}$ may mark any control flow point as $\alpha$ may be instantiated with any type (See Figure 7 for the formal typing rule.). Here is a well-typed triple in which $\ell$ marks three different control flow points, each with different types:

$$\langle \Lambda\beta.\lambda x{:}\beta.\ell[\beta][\![x]\!], \ell[\mathsf{int}][\![3]\!], \ell[\mathsf{bool}][\![\mathbf{true}]\!] \rangle$$

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

$$\frac{\ell{:}\overline{\alpha}.\tau \in \Gamma}{\Delta; \Gamma \vdash \ell : (\overline{\alpha}.\tau)\ \mathsf{label}}$$

$$\frac{\Delta; \Gamma \vdash e_i : (\overline{\alpha}_i.\tau_i)\ \mathsf{label} \qquad \Delta \vdash \overline{\beta}.\tau \preceq \overline{\alpha}_i.\tau_i}{\Delta; \Gamma \vdash \{\overline{e}\} : (\overline{\beta}.\tau)\ \mathsf{pc}}$$

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

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

$$\frac{\Delta; \Gamma \vdash e_1 : (\overline{\alpha}.\tau)\ \mathsf{label} \qquad \Delta \vdash \tau_i \qquad \Delta; \Gamma \vdash e_2 : \tau[\overline{\tau}/\overline{\alpha}]}{\Delta; \Gamma \vdash e_1[\overline{\tau}][\![e_2]\!] : \tau[\overline{\tau}/\overline{\alpha}]}$$

$$\frac{\Delta; \Gamma \vdash e : \mathsf{advice}}{\Delta; \Gamma \vdash \Uparrow e : 1}$$

$$\frac{\Delta; \Gamma \vdash e_1 : (\overline{\alpha}.\tau)\ \mathsf{pc} \qquad \Delta, \overline{\alpha}; \Gamma, x{:}\tau \vdash e_2 : \tau}{\Delta; \Gamma \vdash \{e_1.\overline{\alpha}x{:}\tau \to e_2\} : \mathsf{advice}}$$

$$\frac{\begin{array}{c}\Delta, \alpha \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \Delta' = \mathrm{FTV}(\tau_3) - \Delta \\ (\Theta = \mathrm{MGU}(\tau_2, \tau_3)\ \text{implies}\ \Delta, \Delta'; \Theta(\Gamma) \vdash \Theta(e_1) : \Theta(\tau_1[t_3/\alpha])) \\ \Delta, \Delta' \vdash \mathrm{cod}(\Theta) \qquad \Delta, \alpha; \Gamma \vdash e_2 : \tau_1\end{array}}{\Delta; \Gamma \vdash \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) : \tau_1[\tau_2/\alpha]}$$

**Figure 7.** Typing rules excerpt for $\mathbb{F}_A$

Notice that marking control flow points that occur inside polymorphic functions is no different from marking other control flow points even though $\ell$'s abstract type variable $\alpha$ may be instantiated in a different way 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 in the core calculus is quite similar to typing point cuts in the source. Each label in the set must be a generic instance of the type of the set. For example, given labels $\ell_1$ of type $(1 \times 1)$ label and $\ell_2$ of type $(1 \times \mathsf{bool})$ label, a label set containing them can be given the type $(\alpha.1 \times \alpha)$ pc because $\alpha.1 \times \alpha$ can be instantiated to either $1 \times 1$ or $1 \times \mathsf{bool}$. The rules for label sets and label set union ensure these invariants.

When typing advice in the core calculus, 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 $\overline{\alpha}.\tau$ label then advice $\{e_1.\overline{\alpha}x{:}\tau' \to e_2\}$ type checks only when $\tau'$ is $\tau$. The type $\tau'$ cannot be more specific than $\tau$. If advice needs to refine the type of $\tau$, 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 is extended with $\mathbf{new}\ \overline{\alpha}.\tau \leq e$. 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 $\tau_2$ is more specific than $\tau_1$). 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 is an identity function) and then uses this label to mark a join point inside a polymorphic function.

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

The $\mathbf{typecase}$ expression is slightly more general in the core language 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 language, 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.

### 4.2 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 PolyAML, 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 language 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 [10].

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 commands may be found in Figure 8 and the typing rules in Figure 9.

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

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 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 $\mathtt{data}(E)$. 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 $\rho$ against $e_1$, a reified stack. Note that stack patterns, $\rho$, include first-class point cuts so they must be evaluated to pattern values, $\varphi$, to resolve these point cuts before matching.

If, after evaluation, the pattern value successfully matches the stack, then the expression $e_2$ evaluates, with its pattern variables

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

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

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

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

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

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

$$\frac{\mathit{data}(E) = v}{\Sigma; A; E[\mathbf{stack}] \mapsto \Sigma; A; E[v]}$$

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

$$\frac{}{\Sigma \vdash \bullet \simeq \bullet \rhd \cdot}$$

$$\frac{\Sigma \vdash v_2 \simeq \varphi \rhd \Theta \qquad \ell{:}\overline{\beta}.\tau_2 \le \ell' \in \Sigma \quad \Sigma \vdash \ell \le \ell_i \text{ for some } i \qquad \exists \overline{\sigma}.\tau_2[\overline{\tau}/\overline{\beta}] = \tau_1[\overline{\sigma}/\overline{\alpha}]}{\Sigma \vdash \ell[\overline{\tau}][\![v_1]\!]{::}v_2 \simeq \{\overline{\ell}\}[\overline{\alpha}][\![x]\!]{:}\tau_1{::}\varphi \rhd \Theta, \overline{\sigma}/\overline{\alpha}, v_1/x}$$

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

**Figure 8.** Stack operational semantics

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 $\varphi$ with a reified stack $v$ to produce a substitution $\Theta$.

### 4.3 Type Safety

We have shown that $\mathbb{F}_A$ is type sound through the usual Progress and Preservation theorems. We use the judgment $\vdash (\Sigma; A; e)$ ok to denote a well-formed abstract machine state. Details may be found in the companion technical report [11].

THEOREM 4.1 (Progress). *If* $\vdash (\Sigma; A; e)$ ok *then either the configuration is finished, or there exists another configuration* $\Sigma'; A'; e'$ *such that* $\Sigma; A; e \mapsto \Sigma'; A'; e'$.

THEOREM 4.2 (Preservation). *If* $\vdash (\Sigma; A; e)$ ok *and* $\Sigma; A; e \mapsto \Sigma'; A'; e'$, *then* $\Sigma'$ *and* $A'$ *extend* $\Sigma$ *and* $A$ *such that* $\vdash (\Sigma'; A'; e')$ ok.

## 5. Interpreting PolyAML in $\mathbb{F}_A$

We give an operational semantics to well-typed PolyAML programs by defining a type-directed translation into the $\mathbb{F}_A$ language. This translation is defined by the following mutually recursive judgments for over terms, types, patterns, declarations and point cut designators. The translation was significantly inspired by those in found in WZL [39] and Dantas and Walker [10]. Much of the translation is straightforward so we only sketch it here. The complete translation appears in the companion technical report [11].

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

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

$$\frac{}{\Delta; \Gamma \vdash \mathbf{stack} : \mathsf{stack}} \qquad \frac{}{\Delta; \Gamma \vdash \bullet : \mathsf{stack}}$$

$$\frac{\ell{:}\overline{\alpha}.\tau \in \Gamma \qquad \Delta \vdash \tau_i \qquad \Delta; \Gamma \vdash v_1 : \tau[\overline{\tau}/\overline{\alpha}] \qquad \Delta; \Gamma \vdash v_2 : \mathsf{stack}}{\Delta; \Gamma \vdash \ell[\overline{\tau}][\![v_1]\!]{::}v_2 : \mathsf{stack}}$$

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

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

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

$$\frac{\Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma'}{\Delta; \Gamma \vdash \_{::}\rho \dashv \Delta'; \Gamma'}$$

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

**Figure 9.** Stack typing

| | |
|---|---|
| $\Delta \vdash \mathtt{t} \xRightarrow{\text{type}} \tau'$ | Translation of source types into target types |
| $\Delta; \Phi; \Gamma \vdash \mathtt{p} \xRightarrow{\text{pat}} \rho \dashv \Delta'; \Gamma'; \Xi$ | Translation of stack patterns, producing a mapping between source and target variables |
| $\Delta; \Phi; \Gamma \vdash^{\text{loc}} \mathtt{e} : \mathtt{t} \xRightarrow{\text{term}} e'$ | Translation of locally-typed terms |
| $\Delta; \Phi; \Gamma \vdash \mathtt{e} : \mathtt{t} \xRightarrow{\text{term}} e'$ | Translation of other terms |
| $\Delta; \Phi; \Gamma \vdash \mathtt{d}; \mathtt{e} : \mathtt{t} \xRightarrow{\text{decs}} e'$ | Translation of declarations |
| $\mathtt{e} : \mathtt{t} \xRightarrow{\text{prog}} e'$ | Translation of programs |

**Figure 10.** Translation judgments

The basic idea of the translation is that join points must be made explicit in $\mathbb{F}_A$. Therefore, we translate functions so that that they include explicitly labeled join points at their entry and exit and so that they store information on the stack as they execute. More specifically, for each function we create three labels $\mathtt{f}_{\mathbf{before}}$, $\mathtt{f}_{\mathbf{after}}$ and $\mathtt{f}_{\mathbf{stk}}$ for these join points. So that source language programs can refer to the entry point of any function, all labels $\mathtt{f}_{\mathbf{before}}$ are derived from a distinguished label $\mathcal{U}_{\mathbf{before}}$. Likewise, $\mathcal{U}_{\mathbf{after}}$ and $\mathcal{U}_{\mathbf{stk}}$ are the parents of $\mathtt{f}_{\mathbf{after}}$ and $\mathtt{f}_{\mathbf{stk}}$.

The most interesting part of the encoding is the translation of pointcuts, functions and advice declarations, shown in Figure 11. Pointcuts are translated into triples of $\mathbb{F}_A$ pointcuts. The pointcut **any** becomes a triples of pointcuts containing the parents of all **before**, **after**, and **stk** labels respectively. Sets of functions are translated into triples of pointcuts containing their associated **before**, **after**, and **stk** labels.

The translation of functions begins by creating the labels, $\mathtt{f}_{\mathbf{before}}$, $\mathtt{f}_{\mathbf{after}}$, and $\mathtt{f}_{\mathbf{stk}}$ for the functions join points. Inside the body of the translated function, a **store** statement marks the function's stack frame. Labeled join points are wrapped around the function's input

$$\frac{}{\Delta;\Phi;\Gamma \vdash^{\text{loc}} \textbf{any} : \textbf{pc}\ (\textbf{all}\ \mathtt{a}.\mathtt{a}, \textbf{all}\ \mathtt{a}.\mathtt{a}) \xrightarrow{\text{term}} \langle\{\mathcal{U}_{\textbf{before}}\},\{\mathcal{U}_{\textbf{stk}}\},\{\mathcal{U}_{\textbf{after}}\}\rangle}$$

$$\frac{\forall \mathtt{i} \quad \mathtt{f}_\mathtt{i} \in \Phi \quad \Gamma(\mathtt{f}_\mathtt{i}) = \textbf{all}\ \overline{\mathtt{a}}.\mathtt{t}_{1,\mathtt{i}} \texttt{->} \mathtt{t}_{2,\mathtt{i}} \quad \Delta \vdash \mathtt{s}_1 \preceq \textbf{all}\ \overline{\mathtt{a}}.\mathtt{t}_{1,\mathtt{i}} \quad \Delta \vdash \mathtt{s}_2 \preceq \textbf{all}\ \overline{\mathtt{a}}.\mathtt{t}_{2,\mathtt{i}}}{\Delta;\Phi;\Gamma \vdash^{\text{loc}} \{\overline{\mathtt{f}}\} : (\mathtt{s}_1, \mathtt{s}_2)\ \textbf{:}\ \textbf{pc}\ (\mathtt{s}_1, \mathtt{s}_2) \xrightarrow{\text{term}} \langle\{\overline{\mathtt{f}_{\textbf{before}}}\},\{\overline{\mathtt{f}_{\textbf{stk}}}\},\{\overline{\mathtt{f}_{\textbf{after}}}\}\rangle}$$

$$\frac{\begin{array}{c}\overline{\mathtt{a}} = (\mathrm{FTV}(\mathtt{t}_1) \cup \mathrm{FTV}(\mathtt{t}_2)) - \Delta \\ \Delta, \overline{\mathtt{a}} \vdash \mathtt{t}_1 \texttt{->} \mathtt{t}_2 \xrightarrow{\text{type}} \tau_1' \to \tau_2' \\ \Delta, \overline{\mathtt{a}}; \Phi, \mathtt{f}; \Gamma, \mathtt{f} :: \mathtt{t}_1 \texttt{->} \mathtt{t}_2, \mathtt{x} :: \mathtt{t}_1 \vdash \mathtt{e}_1 : \mathtt{t}_2 \xrightarrow{\text{term}} e_1' \\ \Delta; \Phi, \mathtt{f}; \Gamma, \mathtt{f} :: \textbf{all}\ \overline{\mathtt{a}}.\mathtt{t}_1 \texttt{->} \mathtt{t}_2 \vdash \mathtt{e}_2 : \mathtt{t} \xrightarrow{\text{term}} e_2'\end{array}}{\begin{array}{l}\Delta;\Phi;\Gamma \vdash \textbf{rec}\ \mathtt{f}\ (\mathtt{x}:\mathtt{t}_1)\ \textbf{:}\ \mathtt{t}_2\ \textbf{=}\ \mathtt{e}_1;\mathtt{e}_2 : \mathtt{t} \xrightarrow{\text{decs}} \\ \quad \textbf{let}\ \mathtt{f}_{\textbf{before}} : (\overline{\alpha}.\tau_1' \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \quad\quad \textbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{stack} \times \mathsf{string}) \le \mathcal{U}_{\textbf{before}}\ \textbf{in} \\ \quad \textbf{let}\ \mathtt{f}_{\textbf{after}} : (\overline{\alpha}.\tau_2' \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \quad\quad \textbf{new}\ (\overline{\alpha}.\tau_2' \times \mathsf{stack} \times \mathsf{string}) \le \mathcal{U}_{\textbf{after}}\ \textbf{in} \\ \quad \textbf{let}\ \mathtt{f}_{\textbf{stk}} : (\overline{\alpha}.\tau_1' \times \mathsf{string})\ \mathsf{label} = \\ \quad\quad \textbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{string}) \le \mathcal{U}_{\textbf{stk}}\ \textbf{in} \\ \\ \quad \textbf{let}\ \mathtt{f} : \forall\overline{\alpha}.\tau_1' \to \tau_2' = \textbf{fix}\ \mathtt{f} : \forall\overline{\alpha}.\tau_1' \to \tau_2'. \\ \quad\quad \Lambda\overline{\alpha}.\lambda\mathtt{x}{:}\tau_1'.\textbf{store}\ \mathtt{f}_{\mathtt{stk}}[\overline{\alpha}][\![\langle \mathtt{x}, \text{“f”}\rangle]\!]\ \textbf{in} \\ \quad\quad\quad \textbf{let}\ \langle \mathtt{x}, \_, \_\rangle = \mathtt{f}_{\textbf{before}}[\overline{\alpha}][\![\langle \mathtt{x}, \textbf{stack}, \text{“f”}\rangle]\!]\ \textbf{in} \\ \quad\quad\quad\quad \textbf{let}\ \langle \mathtt{x}, \_, \_\rangle = \mathtt{f}_{\textbf{after}}[\overline{\alpha}][\![\langle e_1', \textbf{stack}, \text{“f”}\rangle]\!]\ \textbf{in}\ \mathtt{x} \\ \quad \textbf{in}\ e_2'\end{array}}$$

$$\frac{\begin{array}{c}\Delta;\Phi;\Gamma \vdash^{\text{loc}} \mathtt{e}_1 : \textbf{pc}\ \mathtt{pt} \xrightarrow{\text{term}} e_1' \qquad \pi(\mathtt{tm}, \mathtt{pt}) = \textbf{all}\ \overline{\mathtt{a}}.\mathtt{t}_1 \\ \pi(\mathtt{tm}, e_1') = e_1'' \qquad \overline{\mathtt{a}} = \mathrm{FTV}(\mathtt{t}_1) - \Delta \qquad \Delta, \overline{\mathtt{a}} \vdash \mathtt{t}_1 \xrightarrow{\text{type}} \tau_1' \\ \Delta, \overline{\mathtt{a}}; \Phi; \Gamma, \mathtt{x}{:}\mathtt{t}_1, \mathtt{y}{:}\textbf{stack}, \mathtt{z}{:}\textbf{string} \vdash \mathtt{e}_2 : \mathtt{t}_1 \xrightarrow{\text{term}} e_2' \\ \Delta;\Phi;\Gamma \vdash \mathtt{e}_3 : \mathtt{t}_2 \xrightarrow{\text{term}} e_3'\end{array}}{\begin{array}{l}\Delta;\Phi;\Gamma \vdash \textbf{advice}\ \mathtt{tm}\ \mathtt{e}_1\ (\mathtt{x}:\mathtt{t}_1, \mathtt{y}, \mathtt{z})\ \textbf{=}\ \mathtt{e}_2;\mathtt{e}_3 : \mathtt{t}_2 \xrightarrow{\text{decs}} \\ \quad \textbf{let}\ \_ : 1 = \Uparrow \{e_1''.\overline{\alpha}\mathtt{x}{:}(\tau_1' \times \mathsf{stack} \times \mathsf{string}) \to \\ \quad\quad \textbf{let}\ \langle \mathtt{x}, \mathtt{y}, \mathtt{z}\rangle = \mathtt{x}\ \textbf{in}\ \langle e_2', \mathtt{y}, \mathtt{z}\rangle\}\ \textbf{in}\ e_3'\end{array}}$$

**Figure 11.** Translation of pointcuts, functions, and advice

and body respectively to implement for **before** and **after** advice. Because PolyAML advice expects the current stack and a string of the function name, we also insert **stack**s and string constants into the join points.

The most significant difference between advice in PolyAML 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 triple of $\mathbb{F}_A$ pointcuts, the `tm` is used to determine which component is used. The translation also splits the input of the advice into the three arguments that PolyAML expects and immediately installs the advice.

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

THEOREM 5.1 (Translation defined on well-typed programs). *If* $\cdot;\cdot;\cdot;\mathtt{e} \vdash \mathtt{t} \Rightarrow \Theta$; *then* $\Theta(\mathtt{e}) : \Theta(\mathtt{t}) \xrightarrow{\text{prog}} e'$

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

THEOREM 5.2 (Translation type soundness). *If* $\mathtt{e} : \mathtt{t} \xrightarrow{\text{prog}} e'$ *then* $\cdot;\cdot \vdash e' : \tau'$ *where* $\cdot \vdash \mathtt{t} \xrightarrow{\text{type}} \tau'$.

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

THEOREM 5.3 (PolyAML safety). *Suppose* $\mathtt{e} : \mathtt{t} \xrightarrow{\text{prog}} e'$ *then either* $e'$ *fails to terminate or there exists a sequence of reductions* $\cdot;\cdot;e' \mapsto^* \Sigma;\mathsf{A};e''$ *to a finished configuration.*

Details for the above proofs may be found in the companion technical report [11].

## 6. Related work

Over the last several years, researchers have begun to build semantic foundations for aspect-oriented programming paradigms [40, 12, 5, 19, 20, 26, 39, 13, 4]. As mentioned earlier, our work builds upon the framework proposed by Walker, Zdancewic, and Ligatti [39], 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 a novel type inference algorithm that is conservative over Hindley-Milner inference, one thing that was 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,[2] Tatsuzawa, Masuhara and Yonezawa [36] have implemented an aspect-oriented version of core O'Caml they call Aspectual Caml. Their implementation effort is impressive and deals with several features we have not considered here including curried functions and datatypes. Although there are similarities between PolyAML and Aspectual O'Caml, there are also many differences:

- Point cut designators in PolyAML can only reference names that are in scope. PolyAML names are indivisible and $\alpha$-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 point cut 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 point cuts are second-class citizens. It is not possible to write down the type of a point cut in Aspectual Caml, or pass a point cut 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 point cuts 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 PolyAML. PolyAML's type system has been proven sound with respect to its operational semantics.

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 [25]. 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

[2] 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, Tatsuzawa et al.'s work first appeared in March 2005.

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 [2]. 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 [1]. In earlier work [9], 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. Building on concurrent work by Washburn and Weirich [41] and Dantas and Walker [10], we are working on extending the language defined in this paper to include abstract types and protection mechanisms that ensure abstractions are respected, even in the presence of type-analyzing advice.

Tucker and Krishnamurthi [37] 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.

## 7. Conclusions

This paper defines PolyAML, 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 point cuts are necessary even when the underlying code base is completely monomorphic. Otherwise, there is no way to assemble a collection of joins point 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 PolyAML that handles first-class polymorphic pointcuts in a simple but effective way, allowing programmers to write convenient security, profiling or debugging libraries. We give PolyAML a semantics by compiling it into a typed intermediate calculus. We have proven the intermediate calculus is type-safe. The reason for giving PolyAML 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.

## Acknowledgments

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

## References

[1] J. Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In *Workshop on Foundations of Aspect-Oriented Languages*, Mar. 2004.

[2] J. Aldrich. Open modules: Reconciling extensibility and information hiding. In *Proceedings of the Software Engineering Properties of Languages for Aspect Technologies*, Mar. 2004.

[3] L. Bauer, J. Ligatti, and D. Walker. Composing security policies in polymer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005. To appear.

[4] G. Bruns, R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. muABC: A minimal aspect calculus. In *Concur*, pages 209–224, Apr. 2004.

[5] C. Clifton and G. T. Leavens. Assistants and observers: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, Apr. 2002.

[6] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 54–66, Boston, Jan. 2000. ACM Press.

[7] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of the Third International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *ACM Symposium on Principles of Programming Languages*, pages 207–212, Albuquerque, New Mexico, 1982.

[9] D. S. Dantas and D. Walker. Aspects, information hiding and modularity. Technical Report TR-696-04, Princeton University, Nov. 2003.

[10] D. S. Dantas and D. Walker. Harmless advice. In *Workshop on Foundations of Object-Oriented Languages*, Jan. 2005.

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

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

[13] R. Douence, O. Motelet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Conference on Aspect-Oriented Software Development*, pages 141–150, Mar. 2004.

[14] Úlfar. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, Sept. 1999.

[15] Úlfar. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.

[16] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[17] R. E. Filman and D. P. Friedman. *Aspect-Oriented Software Development*, chapter Aspect-Oriented Programming is Quantification and Obliviousness. Addison-Wesley, 2005.

[18] M. Fiuczynski, Y. Cody, R. Grimm, and D. Walker. Patch(1) considered harmful. In *HotOS*, July 2005. To appear.

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

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

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

[22] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[23] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, June 1999.

[24] X. Leroy. The Objective Caml system: Documentation and user's manual, 2000. With Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Available from `http://caml.inria.fr`.

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

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

[27] D. Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.

[28] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.

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

[30] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. Submitted to the Journal of Functional Programming, 2005.

[31] S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: Practical type inference for generalised algebraic dataypes. Available at `http://www.cis.upenn.edu/~geoffw/research/`, July 2004.

[32] B. C. Pierce and D. N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–265, San Diego, CA, 1998.

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

[34] V. Simonet and F. Pottier. Constraint-based type inference for guarded algebraic data types. Technical Report Research Report 5462, INRIA, Jan. 2005.

[35] P. J. Stuckey and M. Sulzmann. Type inference for guarded recursive data types. Submitted for publication, Feb. 2005.

[36] H. Tatsuzawa, H. Masuhara, and A. Yonezawa. Aspectual Caml: An aspect-oriented functional language. In *Workshop on Foundations of Aspect Oriented Languages*, pages 39–50, Mar. 2005.

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

[38] D. Vytiniotis, S. Weirich, and S. Peyton Jones. Boxy type inference for higher-rank types and impredicativity. Available at `http://www.cis.upenn.edu/~dimitriv/boxy/`, April 2005.

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

[40] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 2003.

[41] G. Washburn and S. Weirich. Generalizing parametricity using information flow. In *The 20th Annual IEEE Symposium on Logic in Computer Science (LICS 2005)*, Chicago, IL, June 2005.

`$Id: poly-aspect.tex 723 2005-08-27 20:52:38Z geoffw $`