Analyzing polymorphic advice

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

Geoffrey Washburn
Stephanie Weirich
University of Pennsylvania
{geoffw, sweirich}@cis.upenn.edu

Princeton University Technical Report TR-717-04
December 2004

Abstract

We take one of the first steps towards developing a practical, statically-typed, functional, aspect-oriented programming language by showing how to integrate polymorphism and type analysis with aspect-oriented programming features. In particular, we demonstrate how to define type-safe polymorphic advice using pointcuts that unify a collection of polymorphic join points. We also introduce a new mechanism for specifying context-sensitive advice that involves pattern matching against the current stack of activation records, and meshes well with functional programming idioms. We give our language meaning via a type-directed translation into an expressive, but fairly simple, 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 the collection of all its children. Type safety requires that the type of each child is a generic instance of the type of the polymorphic parent. 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.

1 Introduction

Aspect-Oriented Programming Languages (AOPL) allow programmers to independently specify what computations to perform as well as when to perform them. For example, AspectJ [14] 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 *point cut*. A collection of point cuts and advice organized to perform a coherent task is called an *aspect*.

The profiler described above could be implemented without aspects by placing the profiling code into 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. Second, there may be some specific convention concerning how to call the profiling functions, and when calls to these functions are spread throughout the code base, it may be difficult to maintain these conventions correctly. For example, IBM experimented with aspects in their middleware product line, finding that aspects aided in the consistent application of cross-cutting features such as profiling among others [5]. Third, the profiled code becomes "tangled" with the rest of the code involved in the main computation, potentially obscuring the central algorithm. This problem gets much worse when code for several different tasks such as profiling, debugging, distribution, access control and others are mixed together. Fourth, in some situations, one does not have access to the source code or does not have the right to modify it and consequently manual insertion of function calls is out of the question.

Although aspects are increasingly popular in object-oriented languages, aside from a couple of toy projects, they have not yet been incorporated into any statically-typed functional language. One of the challenges along the way lies in developing a typing discipline appropriate for functional languages that is safe, yet sufficiently flexible to fit aspect-oriented programming idioms. In some situations, typing is straightforward. For instance, when a piece of advice advises 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) any function call, regardless of the type of the function's argument (or result). 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. In order 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 will be highly polymorphic and the type of each element will be a generic instance of the collection's type.

In addition to finding polymorphic types for advice, we wish to allow advice to change its behavior depending upon the type of the advised function. For instance, our otherwise generic profiling advice might be specialized so that on any call to a function with an integer argument, it keeps track of the distribution of calls with particular arguments. This and other similar examples require that the advice be able to determine the type of the function argument. In AspectJ, where object-orientation is the underlying programming paradigm, downcasts are used to determine types, but in a functional language, we believe that intentional type analysis is the appropriate mechanism.and

Finally, in order to emulate the context-sensitive advice found in languages such as AspectJ, we propose a simple yet general mechanism for analyzing the contents of a stack of polymorphic activation records. Once again, following the spirit of functional programming, the stack is treated as a functional data structure and the programmer may use recursive functions and pattern matching to determine its contents.

In this paper, we analyze these programming features and develop a simple language that contains the essential elements of a polymorphic functional programming language with before and after advice. The language we define is a true aspect-oriented language according to the definition given by Filman and Friedman [11] as it is *oblivious.* In order to specify the semantics of our language, we give a type-directed translation from the source into a type-safe intermediate language, following previous work by Walker, Zdancewic and Ligatti (WZL) [19], who define the semantics of a monomorphic language in this way.[1] This translation helps to modularize the semantics for the source and could be used as the first step in a compilation strategy.

The core language, though it builds directly on WZL, is itself an important contribution of our work. One of the novelties of the core language are its first-class, polymorphic labels, which can be used to mark any control-flow point in a program. Unlike in WZL, where the 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 getting or setting references; groups for raising or catching exceptions, etc. Polymorphism is crucial for defining these groups as the type of each member of a group (*i.e.,* child of an internal tree node) is a polymorphic instance of the type of the parent. In addition, polymorphism is used in conjunction with many other features of the language: point cuts, which assemble sets of labels, advice, and functions. Overall, we have worked hard to gives a clean semantics to each feature in this language, and to separate unrelated concerns. We believe this will faciltate further exploration and extension of language.

2 Programming with aspects in PolyAML

The language PolyAML (Figure 1) contains the essential features of a polymorphic aspect-oriented functional language. For clarity in the examples below, we add language features, such as recursion and I/O, and elide

[1] The intermediate language is not oblivious. This does not detract from the properties of the source in any way, and after all, any oblivious language is always compiled into a non-oblivious language.

(*polytypes*)	s	::=	forall $\overline{\texttt{a}}$.t
(*monotypes*)	t	::=	a $\mid$ unit $\mid$ string $\mid$ stack $\mid$ t_1 -> t_2
(*terms*)	e	::=	x $\mid$ () $\mid$ f [$\overline{\texttt{t}}$] $\mid$ $e_1 e_2$ $\mid$ ds e
		$\mid$	stkcase e_1 ($\overline{\texttt{p=>e}}$ $\mid$_=> e_2)
		$\mid$	typecase a ($\overline{\texttt{t=>e}}$ $\mid$_=> e)
(*patterns*)	p	::=	nil $\mid$ x $\mid$ _::p $\mid$ pt(x:t,n)::p
(*declarations*)	ds	::=	. $\mid$ let f (x:t_1):t_2 = e in ds
		$\mid$	time pt(x:t,s,n) = e in ds
(*point cut designators*)	pt	::=	{$\overline{\texttt{f}}$} $\mid$ any
(*trigger time*)	time	::=	before $\mid$ after
(*programs*)	prog	::=	ds e

Figure 1: Syntax of PolyAML

some type information. Although PolyAML is explicitly typed, we restrict polymorphism to be predicative, merely to simplify type inference.

An aspect in PolyAML is composed of several pieces of *advice*. Advice in PolyAML is second-class and includes two parts: the body which specifies what to do, and a *point-cut designation*, which specifies when to do it. A point-cut designation may either be a set of function names, which triggers the advice `before` or `after` any of the functions in the set are called, or it may be `any`, which is triggers the advice when any function is called. For uniformity, all functions in PolyAML must be named.

When `before` advice is triggered, the body of the advice receives the argument of the function, the name of the function that was called as a string, and a reification of the execution stack. (The call that triggers the advice is at the top of the stack.) Likewise, when `after` advice is triggered by the return of a function, the body receives the result of the function, as well as the name of the function that triggered the advice and the current stack.

One of the simplest uses of aspect-oriented programming is to add tracing information to functions—statements that are executed whenever a function is called or returns. For example, we can advise the program below to display messages before any function is called and after the functions `f` and `g` return.

```
let f (x:int) = x + 1 in
let g (x:bool) = if x then f 1 else f 0 in
let h (x:a) = (x,x) in
before any (x:a, s:stack, n:string) =
   print "entering"; println n; x
after { f,g }(x:a, s:stack, n:string) =
   print "leaving"; println n; x
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 the these functions and they have different types, the advice must be polymorphic. Moreover, since the result type of functions `f` and `g` have no type structure in common, the argument `x` of the after advice must be completely abstract.[2] If, on the other hand, the result types of both functions were pairs, say (`int*bool`) and (`bool*bool`), the type of the after advice argument `x` could be the more specific type (`a*bool`). In general, the type of the advice argument may be the most specific type τ such that all functions referenced in the point cut are instances of τ.[3]

We might also want the tracing routine to print not only the name of the function that is called, but also its argument. Therefore, PolyAML allows the programmer to specify many different pieces of advice that

[2] We indicate this by annotating `x` with type variable `a`, which is implicitly quantified.

[3] Unless the programmer intends to define type-analyzing advice as explained in the next paragraph. In this case, the type annotating the argument may be more specific.

are triggered based on the specific type of the argument. (For simplicity, all advice that is applicable to a program point is triggered in the order in which it is declared.)

```
  before any (x:a, s:stack, n:string) =
     print "entering "; print n; x
  before any (x:int, s:stack, n:string) =
     print " with arg "; println (itos x); x
  before any (x:bool, s:stack, n:string) =
     print " with arg "; println (if b then "true" else "false"); x
```

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 run-time. However, without this ability we cannot implement this tracing aspect. Because of this example and many others, a polymorphic aspect-oriented programming language is of limited use without type analysis. For further flexibility, PolyAML also includes a typecase construct to analyze type variables directly.

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

```
before { f } (x:a, s:stack, n:string) =
  (stkcase s of
    _ :: { g }(y:bool, m:string) :: s' =>
         if y then print "entering f from g" else ()
  | _ => ()); x
```

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.

```
before any (x:a, s:stack, n:string) =
   let rec walk s =
      stkcase s of
        nil => abort ()
      | any (y:a, nf:string) :: s' =>
          if enables nf n then () else walk s'
   in walk s; x
```

As mentioned in the introduction, the semantics of PolyAML is given the translation into an expressive polymorphic core language. In the next two sections, we describe the semantics of $\mathbb{F}_{\mathsf{A}}$ in detail. In Section 5, we describe the translation from PolyAML into the core.

3 The core language and polymorphism

The core language $\mathbb{F}_{\mathsf{A}}$ 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. One of the features of the language is the fact that all constructs are defined orthogonally to one another. One advantage of this design is that we can easily experiment with the 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. Due to lack of space, the complete semantics $\mathbb{F}_{\mathsf{A}}$ appears in Appendix A.

3.1 The semantics of explicit join points

For exposition, to describe the semantics of $\mathbb{F}_A$ we start here with a simple version similar to WZL and extend it in the following sections. The syntax of this language is summarized below.

$$\begin{array}{rcl} \tau & ::= & 1 \mid \mathsf{string} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \ldots \times \tau_n \mid \alpha \mid \forall\alpha.\tau \mid \tau\ \mathsf{label} \mid \tau\ \mathsf{pc} \mid \mathsf{advice} \\ e & ::= & \langle\rangle \mid s \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \langle\overline{e}\rangle \mid \mathbf{let}\ \langle\overline{x}\rangle = e_1\ \mathbf{in}\ e_2 \\ & \mid & \Lambda\alpha.e \mid e[\tau] \mid \ell \mid \mathbf{new}\ \tau \leq e \mid e_1[\,]\llbracket e_2\rrbracket \mid\ \Uparrow e \mid \{e_1.x{:}\tau \to e_2\} \end{array}$$

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

As in WZL, Labeled join points $\ell[\,]\llbracket e\rrbracket$ are the essential mechanism of $\mathbb{F}_A$. The labels, drawn from some infinite set of identifiers, serve two purposes: They mark program points where advice may be triggered and they provide markers for contextual analysis. For example, in the expression $v_1 + \ell[\,]\llbracket e_2\rrbracket$, after e_2 has been evaluated to a value v_2, evaluation of the resulting subterm $\ell[\,]\llbracket v_2\rrbracket$ causes any advice associated with ℓ to be triggered. to New labels may be generated at run time, with the expression $\mathbf{new}\ \tau \leq e$. (We describe the role of e in Section 4.1.) In this way, scoping may be used to reason about what advice may be triggered at a particular location, when the label is unknown.

Advice is a computation that exchanges data with a particular join point, and so is similar to a function. The advice $\{\ell.x{:}\mathsf{int} \to e\}$ is triggered when control flow reaches a join point labeled with ℓ. The variable x is bound to the the data at that point and evaluation proceeds into the body of the advice. For example, if this advice has been installed in the program's dynamic environment, $v_1 + \ell[\,]\llbracket v_2\rrbracket$ evaluates to $v_1 + e[v_2/x]$.

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

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 a set Σ of labels (and their associated types) and A, an ordered list of installed advice. The abstract machine states of the operational semantics are triples $\Sigma; A; e$.

We use evaluation contexts, E, to 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 give the primitive β-reductions for this calculus that describe how terms evaluate in context.

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

The β-reductions for functions, type abstractions and pairs are standard. We discuss the rules for label creation and point cuts in the next section.

Type system. The type system of $\mathbb{F}_A$ maintains the connection between labels, join points and advice. Because it is necessary to pass information back and forth between the join point of interest and the advice, the advice and control flow points must agree about type of data that will be exchanged.

The judgement $\Delta; \Gamma \vdash e : \tau$ indicates that the term e can be given the type τ, where free type variables appear in Δ and the types of term variables and labels appear in Γ. Unit, string, tuple, function and polymorphic term typing are standard.

The type system assigns the type $\tau\ \mathsf{label}$ to labels, which describes the type of expressions they may label at join points. As point cuts are merely labels in this simple calculus, any expression of type $\tau\ \mathsf{label}$ may be considered to have type $\tau\ \mathsf{pc}$. In Section 4 we will generalize the definition of point cuts.

Advice associated with a point cut of type τ pc is constructed from code that expects a variable of type τ. The body of advice must produce a result suitable for returning to the point from which the advice was triggered. Thus, the body of the advice must itself be of type τ. The expression $\Uparrow e$, which installs advice in the run-time environment has type 1 when e has type advice.

We have shown that $\mathbb{F}_A$ (including extensions discussed below) is type sound through the usual Progress and Preservation theorems.

Theorem 3.1 (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 3.2 (Preservation). *If* $\vdash (\Sigma; A; e)$ *ok and* $\Sigma; A; e \mapsto \Sigma'; A'; e'$, *then* Σ' *and* A' *extend* Σ *and* A *such that* $\vdash (\Sigma'; A'; e')$ *ok.*

3.2 Polymorphic labels and advice

Although we have based our core language on a polymorphic λ-calculus, the language discussed above is not flexible enough to encode the examples in Section 2. Advice can only apply to program points with the same type. We make advice more flexible by generalizing the type of point cuts, as shown in the syntax below, to include a vector of type variables, bound within the type of point cut.

$$\begin{array}{lcl} \tau & ::= & \ldots \mid (\overline{\alpha}.\tau)\ \mathsf{label} \mid (\overline{\alpha}.\tau)\ \mathsf{pc} \\ e & ::= & \ldots \mid \{e_1.\overline{\alpha}x{:}\tau \to e_2\} \mid \textbf{new}\ \overline{\alpha}.\tau \leq e \mid e_1[\overline{\tau}][\![e_2]\!] \end{array}$$

Advice that is triggered by such a point cut must abstract those type variables in its argument and return type.

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

Likewise, because point cuts are just labels, we similarly generalize the label type. When labels are attached to program points, these type arguments must be instantiated.

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

Intuitively, when the join point $\ell[\overline{\tau}][\![v]\!]$ triggers the advice $\{\ell.\overline{\alpha}x{:}\tau \to e\}$, $\overline{\tau}$ will replace $\overline{\alpha}$ and v will replace x in the body of the advice. (In section 4.1, where we generalize point cuts this process becomes more complicated.)

This modification to the point cut type provides flexibility in the use of advice. For example, the following code creates a new label, installs advice for this label (that is an identity function) and then uses this label to mark three join points in the program, one of which is located in a polymorphic function.

$$\begin{array}{l} \textbf{let}\ l = \textbf{new}\ \alpha.\alpha \leq \mathcal{U}\ \textbf{in} \\ \textbf{let}\ \_ = \Uparrow \{l.\alpha x{:}\alpha \to x\}\ \textbf{in} \\ \quad \langle \Lambda\beta.\lambda x{:}\beta.l[\beta][\![x]\!],\ l[\mathsf{int}][\![3]\!],\ l[\mathsf{bool}][\![\textbf{true}]\!]\rangle \end{array}$$

There are several issues that arose leading to this design. The first is in seeing why standard polymorphism is not enough for the above code. For example, it is not immediately clear why we cannot use types such as $(\forall\alpha.\alpha)$ label, $\forall\alpha.(\alpha$ label$)$, or even (in calculus with existential types) $(\exists\alpha.\alpha)$ label instead.

However, the type $(\forall\alpha.\alpha)$ label does not allow α to be bound in the body of advice that is triggered by this label. This label can only mark point cuts of type $\forall\alpha.\alpha$. The type $\forall\alpha.(\alpha$ label$)$ must create a new label whenever it is instantiated, because the type of label to use is not known until then. It also does not allow advice to be polymorphic. Finally, the existential type $(\exists\alpha.\alpha)$ label requires that the labeled expression evaluate to an existential package. If all join points must have an abstract types, it will significantly restrict the locations of a program that may be labeled.

Another issue that arose in our design was keeping run-time type analysis orthogonal from join points and advice. We wanted the only mechanism that could analyze run-time type information to be the $\boldsymbol{typecase}$ term, described below. However, this means that we could not allow advice to be conditionally triggered by the type of the join point. More subtly, we had to ensure that polymorphic point cuts were instantiated only at join points, so that we could rule out the following type-analyzing code:

$$\begin{array}{l}\textbf{let}\ l = \textbf{new}\ \alpha.\alpha \leq \mathcal{U}\ \textbf{in} \\ \textbf{let}\ \_ = \Uparrow \{(l[\mathsf{string}]).x{:}\mathsf{string} \rightarrow \textbf{print}\ x; x\}\ \textbf{in} \\ \ \ \Lambda\beta.\lambda x{:}\beta.l[\beta][\![x]\!]\end{array}$$

Therefore, $\boldsymbol{typecase}$ is the only mechanism in $\mathbb{F}_A$ that allows for dynamic pattern matching against types. The semantics of this operator is fairly standard. The typing rule for $\boldsymbol{typecase}$ is below.

$$\frac{\Delta, \alpha \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \Delta' = \mathrm{FTV}(\tau_3) \qquad \Delta, \Delta'; \Gamma \vdash e_1[\tau_3/\alpha] : \tau_1[\tau_3/\alpha] \qquad \Delta, \alpha; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash \boldsymbol{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) : \tau_1[\tau_2/\alpha]}\ \textsf{wft:tcase}$$

A $\boldsymbol{typecase}$ expression consists of a type τ_2 to match against a type pattern τ_3. The type matches a pattern if there is some substitution for the free variables in the pattern that makes it equal to τ_2. In the case of a match, e_1 is executed, otherwise execution continues with e_2. The $\alpha.\tau_1$ annotation is used for type checking and describes the type of the branches. In the branch e_1 we know that τ_2 is equal to τ_3, so we can let the result type of this branch mention τ_3 instead of τ_2.

4 Extensions

WZL investigated two generalizations of the basic aspect framework. First, they allowed advice to be triggered by multiple labels, using label sets as point cuts. Second, they permitted run-time inspection of the labels appearing in the call stack. Both of these extensions are necessary to support the PolyAML as described in Section 2, so we describe how these extensions interact with polymorphism. In doing so, we make two new contributions to these extensions.

4.1 Generalizing point cuts

In PolyAML, advice may be triggered by a set of function names. To support this mechanism in $\mathbb{F}_A$ we must generalize point cuts from single labels to sets of labels. Advice may then be triggered by any label in the set. To do so, we extend the syntax of the language with expressions to create a set of labels and a union operation for sets.

$$e \quad ::= \quad \ldots \mid \{\overline{e}\} \mid e_1 \cup e_2$$

In WZL, labels grouped together must have the same type, because any of the labels could trigger the advice. With polymorphic advice we can be more flexible in label set formation. Label sets may be composed of labels with different types if we can find some type that is more polymorphic than the types of the constituent labels. In the typing rule below, we use the instance relation $\Delta \vdash \tau_1 \prec \tau_2$ to mean that τ_2 is more specific than τ_1. This instance relation (defined below) is similar to that used in Hindley-Damas-Milner type inference [6].

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

$$\frac{\Delta, \overline{\alpha} \vdash \tau_1 \qquad \Delta, \overline{\beta} \vdash \tau_2 \qquad \Delta \vdash \tau_i \qquad \exists \overline{\tau}.\tau_1[\overline{\tau}/\overline{\alpha}] = \tau_2}{\Delta \vdash \overline{\alpha}.\tau_1 \prec \overline{\beta}.\tau_2}\ \textsf{gen}$$

For example, given labels ℓ_1 of type (1×1) label and ℓ_2 of type $(1 \times \mathsf{bool})$ label, a label set containing them can be given the type $(\alpha.1 \times \alpha)$ pc because this type can be instantiated to that of either of the labels. The formation rule for the union operation, $e_1 \cup e_2$, also employs this instance relation.

Polymorphic advice enables another generalization of point cuts, not considered by WZL; we can arrange all labels into single hierarchy, or tree structure. With such a hierarchy, a join point $\ell[\overline{\tau}][\![e]\!]$ triggers advice $\{\ell'.\overline{\alpha}x{:}\tau' \to e\}$ if the label ℓ is lower in the hierarchy than the label ℓ'.

With this extension, we can use a point cut to refer to all labels lower in the tree, without specifying each such label individually. This mechanism is essential to support $\mathsf{PolyAML}$ advice that should be triggered on entry to *any* function. The advice cannot create this set—not all labels that mark the beginnings functions may be in scope where the advice is specified. With a label hierarchy, we can refer to all such labels if they all descend from a single label, $\mathcal{U}_{\texttt{before}}$.

The label hierarchy is extended when labels are created with $\textbf{\textit{new}}\ \alpha.\tau \leq e$. The argument e becomes the parent of the new label. For soundness, there must be a connection between the type of the new label and the type of the parent label. As above, the new label must have a more specific type than its parent.

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

For completeness, $\mathbb{F}_A$ includes a start label $\mathcal{U}$ that is the ancestor of all labels and has the most polymorphic label type, $\alpha.\alpha\ \mathsf{label}$.

Now that we have described label sets and the label hierarchy we can precisely specify the operational semantics for when advice is triggered. When a join point is reached in β-reduction, an auxiliary judgement, $\Sigma; A; \ell; \tau \Rightarrow v'$, examines the installed advice to create a function v' to apply to the value of the join point.

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

This judgment (advice composition) is described by three rules. The first 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 ℓ descends from one of the labels in the label set, then that advice is triggered. The head advice is composed with the function produced from examining the rest of the advice in the list. Not only does advice composition determine if ℓ is lower in the hierarchy than some label in the label set, but it also determines the substitution for the abstract types $\overline{\alpha}$ in the body of the advice. The typing rules ensure that if the advice is triggered, this substitution will always exist, so the execution of this rule does not require run-time type information.

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

$$\frac{\Sigma; A; \ell; \tau_2 \Rightarrow v_2 \qquad \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \exists\overline{\tau}.\tau_2 = \tau_1[\overline{\tau}/\overline{\alpha}]}{\Sigma; A, \{\{\overline{\ell}\}.\overline{\alpha}x{:}\tau_1 \to e\}; \ell; \tau_2 \Rightarrow \lambda x{:}\tau.v_2(e[\overline{\tau}/\overline{\alpha}])}\ \mathsf{adv{:}cons1}$$

$$\frac{\Sigma; A; \ell; \tau_2 \Rightarrow v_2 \qquad \Sigma \vdash \ell \not\leq \ell_i}{\Sigma; A, \{\{\overline{\ell}\}.\overline{\alpha}x{:}\tau_1 \to e\}; \ell; \tau_2 \Rightarrow v_2}\ \mathsf{adv{:}cons2}$$

4.2 Context analysis

Languages such as AspectJ include pointcut operators such cflow to enable advice to be triggered in a context-sensitive fashion. In our language, we provide direct access to the run-time stack as a functional data structure and we allow programmers to pattern match against this data structure, in much the same way that one pattern matches against a list. WZL's monomorphic core language also contained the ability to query the stack, but the stack was not first-class and the queries had to be formulated as regular expressions. Our pattern matching facilities are simpler and therefore easier to use and describe. Moreover, they fit perfectly within the functional programming idiom, and overall are a substantial improvement over previous work.

Below are the necessary new additions to the syntax of the language for storing type and value information on the stack, capturing and representing the current stack as a data structure, and analyzing a reified stack.

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

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 $\overline{\tau}$. In the operational semantics, the term $\mathbf{stack}$ captures this data stored in the execution context as a first-class data structure.

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

This context is converted, using the auxiliary function $\mathit{data}(\cdot)$, into an ordered list represented by the stack nil $\bullet$ and stack cons :: terms. The type of the returned value is stack. A list of stored stack information may be analyzed with the pattern matching term $\mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3)$. This term attempts to match the pattern ρ against e_1, a reified stack. Note that stack patterns, ρ, include first-class point cuts so they must be evaluated to pattern values, φ, 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 replaced with the corresponding part of the stack. Otherwise execution continues with e_3. The following two β-rules encode this operation. These rules rely on the stack matching relation $\Sigma \vdash v \simeq \varphi \rhd \Theta$ that compares a stack pattern value φ with a reified stack v to produce a substitution Θ.

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

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

The typing rule for stack analysis requires that e_1 be a first-class stack. It also determines the free variables in the pattern ρ, with the relation $\Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma'$, and binds them in the branch e_2.

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e_1 : \mathsf{stack} \\ \Delta; \Gamma \vdash \rho \dashv \Delta'; \Gamma' \qquad \Gamma', \Delta' \text{ linear} \qquad \Delta, \Delta'; \Gamma, \Gamma' \vdash e_2 : \tau \qquad \Delta; \Gamma, x{:}\mathsf{stack} \vdash e_3 : \tau\end{array}}{\Delta; \Gamma \vdash \mathbf{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3) : \tau}\ \textsf{wft:scase}$$

5 Translation

We give a semantics to well-typed $\mathsf{PolyAML}$ 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.

$\Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$	Injection of source types into target types
$\Delta; \Gamma \vdash \mathtt{pt} \overset{\mathtt{time}}{\Longrightarrow} e; \overline{\mathtt{a}}.\mathtt{t}$	Translation of point cut designators to target point cuts and their types
$\Delta; \Gamma \vdash \mathtt{p} \overset{\mathtt{pat}}{\Longrightarrow} \rho \dashv \Delta'; \Gamma'; \Phi$	Translation of stack patterns, producing a mapping between source and target variables
$\Delta; \Gamma \vdash \mathtt{e} : \mathtt{t} \overset{\mathtt{exp}}{\Longrightarrow} e$	Translation of terms
$\Delta; \Gamma \vdash \mathtt{ds}; \mathtt{e} : \mathtt{t} \overset{\mathtt{decs}}{\Longrightarrow} e$	Translation of declarations
$\Delta; \Gamma \vdash \mathtt{ds}\ \mathtt{e} : \mathtt{t} \overset{\mathtt{prog}}{\Longrightarrow} e$	Translation of programs

$$\frac{\begin{array}{c}\overline{\mathtt{a}} = \mathrm{FTV}(\mathtt{t}_1,\mathtt{t}_2) - \Delta \qquad \Delta,\overline{\mathtt{a}} \vdash \mathtt{t}_1 \overset{\mathtt{type}}{\Longrightarrow} \tau_1' \\ \Delta,\overline{\mathtt{a}} \vdash \mathtt{t}_2 \overset{\mathtt{type}}{\Longrightarrow} \tau_2' \qquad \Delta;\Gamma,\mathtt{f}{:}\mathtt{forall}\ \overline{\mathtt{a}}.\mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \vdash \mathtt{ds};\mathtt{e}_2 : \mathtt{t} \overset{\mathtt{decs}}{\Longrightarrow} e_2' \qquad \Delta,\overline{\mathtt{a}};\Gamma,\mathtt{x}{:}\mathtt{t}_1 \vdash \mathtt{e}_1 : \mathtt{t}_2 \overset{\mathtt{exp}}{\Longrightarrow} e_1'\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash \mathtt{let\ f\ (x{:}t_1){:}t_2\ =\ e_1\ in\ ds;e_2 : t} \overset{\mathtt{ds}}{\Longrightarrow} \\ \quad \mathbf{let}\ f_{\mathtt{before}} : (\overline{\alpha}.\tau_1' \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{stack} \times \mathsf{string}) \leq \mathcal{U}_{\mathtt{before}}\ \mathbf{in} \\ \quad \mathbf{let}\ f_{\mathtt{after}} : (\overline{\alpha}.\tau_2' \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.\tau_2' \times \mathsf{stack} \times \mathsf{string}) \leq \mathcal{U}_{\mathtt{after}}\ \mathbf{in} \\ \quad \mathbf{let}\ f_{\mathtt{stk}} : (\overline{\alpha}.\tau_1' \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{string}) \leq \mathcal{U}_{\mathtt{stk}}\ \mathbf{in} \\ \quad \mathbf{let}\ f : \forall\overline{\alpha}.\tau_1' \to \tau_2' = \\ \qquad \Lambda\overline{\alpha}.\lambda x{:}\tau_1.\mathbf{store}\ f_{\mathtt{stk}}[\overline{\alpha}][\![\langle x, \text{“f”}\rangle]\!]\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ \langle x, \_, \_\rangle = f_{\mathtt{before}}[\overline{\alpha}][\![\langle x, \mathbf{stack}, \text{“f”}\rangle]\!]\ \mathbf{in} \\ \qquad\qquad \mathbf{let}\ \langle x, \_, \_\rangle = f_{\mathtt{after}}[\overline{\alpha}][\![\langle e_1', \mathbf{stack}, \text{“f”}\rangle]\!]\ \mathbf{in}\ x \\ \quad \mathbf{in}\ e_2'\end{array}}\ \textsf{tds:let}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{ds};\mathtt{e}_2 : \mathtt{t}_2 \overset{\mathtt{decs}}{\Longrightarrow} e_2' \qquad \Delta;\Gamma \vdash \mathtt{pt} \overset{\mathtt{time}}{\Longrightarrow} e';\overline{\mathtt{a}}.\mathtt{t}_3 \qquad \exists\overline{\mathtt{t}}.\mathtt{t}_3[\overline{\mathtt{t}}/\overline{\mathtt{a}}] = \mathtt{t}_1 \qquad \Delta' = \mathrm{FTV}(\mathtt{t}_1) \\ \Delta,\Delta' \vdash \mathtt{t}_1 \overset{\mathtt{type}}{\Longrightarrow} \tau_1' \qquad \Delta,\overline{\mathtt{a}} \vdash \mathtt{t}_3 \overset{\mathtt{type}}{\Longrightarrow} \tau_3' \qquad \Delta,\Delta';\Gamma,\mathtt{x}{:}\mathtt{t}_1,\mathtt{s}{:}\mathtt{stack},\mathtt{n}{:}\mathtt{string} \vdash \mathtt{e}_1 : \mathtt{t}_1 \overset{\mathtt{exp}}{\Longrightarrow} e_1'\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash \mathtt{time\ pt(x{:}t_1,s,n)\ =\ e_1\ in\ ds;e_2 : t_2} \overset{\mathtt{ds}}{\Longrightarrow} \\ \quad \mathbf{let}\ \_ : 1 = \Uparrow \{e'.\overline{\alpha}x{:}\tau_3' \to \mathbf{let}\ \langle x, s, n\rangle = x\ \mathbf{in} \\ \qquad (\mathbf{typecase}[\gamma.\gamma \to \gamma]\ \tau_3'\ (\tau_1' \Rightarrow \lambda x{:}\tau_1'.e_1', \gamma \Rightarrow \lambda x{:}\gamma.x))x\} \\ \quad \mathbf{in}\ e_2'\end{array}}\ \textsf{tds:ad}$$

Figure 2: Translation of function and advice declarations

The translation was significantly inspired by those in found in WZL [19] and Dantas and Walker [8]. Much of the translation is straightforward so we only sketch it here. The complete translation appears in Appendix C.

The basic idea of the translation is that join points must be made explicit in the source language. 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 $f_{\mathtt{before}}$, $f_{\mathtt{after}}$ and $f_{\mathtt{stk}}$ for these join points. So that source language programs can refer to the entry point of any function all labels $f_{\mathtt{before}}$ are derived from a distinguished label $\mathcal{U}_{\mathtt{before}}$. Likewise, $\mathcal{U}_{\mathtt{after}}$ and $\mathcal{U}_{\mathtt{stk}}$ are the parents of $f_{\mathtt{after}}$ and $f_{\mathtt{stk}}$.

The most interesting part of the encoding is the translation of function and advice declarations, shown in Figure 2. The translation of functions first proceeds recursively on the various pieces of the declaration. Then the labels, $f_{\mathtt{before}}$, $f_{\mathtt{after}}$, and $f_{\mathtt{stk}}$ are created. 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 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 biggest difference between advice in PolyAML and $\mathbb{F}_A$ is that PolyAML advice may pattern match on the type of its argument to decide whether to execute, but $\mathbb{F}_A$ advice may not. In the translation, a **typecase** expression in the body of the advice determines if the type matches and defaults to an identity function if it does not. The translation also splits the input into the three arguments that PolyAML expects and immediately installs the advice.

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

Theorem 5.1 (Program translation type soundness). *If* $\cdot;\cdot \vdash \mathtt{ds\ e : t} \overset{\mathtt{prog}}{\Longrightarrow} e$ *then* $\cdot;\cdot \vdash e : \tau$ *where* $\cdot \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$.

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

Theorem 5.2 (PolyAML safety). *Suppose* $\cdot;\cdot \vdash \mathtt{ds\ e : t} \overset{\mathtt{prog}}{\Longrightarrow} \mathsf{e}$ *then either e fails to terminate or there exists a sequence of reductions* $\cdot;\cdot;\mathsf{e} \mapsto^* \Sigma;\mathsf{A};\mathsf{e}'$ *to a finished configuration.*

6 Related work

Over the last several years, researchers have begun to build semantic foundations for aspect-oriented programming paradigms [20, 9, 4, 12, 13, 16, 19, 10, 3]. As mentioned earlier, our work builds upon the framework proposed by Walker et al. [19], 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.

To our knowledge, the only 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 [15, 17]. They extend a variant of AspectJ with a form of module that allows programmers to choose the join points (i.e., control-flow points) that are exposed to external aspects. Aspectual Collaborations has parameterized aspects that resemble the parameterized classes of Generic Java. When a parameterized aspect is linked into a module, concrete class names replace the parameters. Since types are merely names, the sort of polymorphism necessary is much simpler (at least in certain ways) than required by a functional programming language. For instance, there is no need to develop a generalization relation and type analysis may be replaced by conventional object-oriented down-casts. Overall, the differences between functional and object-oriented language structure have caused our two groups to find quite different solutions to the problem of constructing generic advice.

Closely related to Aspectual Collaborations is Aldrich's notion of Open Modules [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 [7], 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 [21] and Dantas and Walker [8], 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 [18] developed a variant of Scheme with aspect-oriented features. They demonstrate the pleasures of programming with point-cuts and advice as first-class objects. For simplicity's sake, PolyAML only has second-class point cuts and advice. We believe it is straightforward to make these features first-class since they are first-class in our core language.

7 Conclusion

This paper demonstrates the synergy between polymorphism and aspect-oriented programming—the combination is clearly more expressive than the sum of its parts. At the simplest level, this extension permit join points to be located in polymorphic code. More importantly, because polymorphic aspects may be triggered by join points in many more contexts than monomorphic aspects, we have been able to significantly increase the flexibility of point-cut designation. For example, our label hierarchy, which allows us to form groups of related control flow points, wouldn't be definable with only monomorphic labels. Also, explicit label sets may refer to join points of many different types.

Furthermore, we make an additional contribution with respect to stack pattern matching. Our version 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.

Acknowledgements

This research was supported in part by ARDA Grant no. NBCHC030106, National Science Foundation grants CCR-0238328 and CCR-0208601 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.

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

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

[5] A. Colyer and A. Clement. Large-scale aosd for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65. ACM Press, 2004.

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

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

[8] D. S. Dantas and D. Walker. Harmless advice, 2004. Submitted for publication, September 2004.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] 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.

[17] J. Ovlinger. *Modular Programming with Aspectual Collaborations.* PhD thesis, Northeastern University, 2003.

[18] 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.

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

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

[21] G. Washburn and S. Weirich. Generalizing parametricity using information flow. Available at `http://www.cis.upenn.edu/~sweirich/`, July 2004.

A The $\mathbb{F}_A$ language

A.1 Grammar

$$
\begin{array}{lcl}
\multicolumn{3}{l}{(\textit{types})} \\
\tau & ::= & 1 \mid \mathsf{string} \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall\alpha.\tau \mid (\overline{\alpha}.\tau)\ \mathsf{label} \mid (\overline{\alpha}.\tau)\ \mathsf{pc} \\
 & \mid & \mathsf{advice} \mid \mathsf{stack} \mid \tau_1 \times \ldots \times \tau_n \\
\multicolumn{3}{l}{(\textit{terms})} \\
e & ::= & \langle\rangle \mid s \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2 \mid \Lambda\alpha.e \mid e[\tau] \mid \langle\overline{e}\rangle \mid \mathbf{let}\ \langle\overline{x}\rangle = e_1\ \mathbf{in}\ e_2 \mid \ell \\
 & \mid & e_1[\overline{\tau}][\![e_2]\!] \mid \mathbf{new}\ \overline{\alpha}.\tau \leq e \mid\ \Uparrow e \mid \{e_1.\overline{\alpha}x{:}\tau \rightarrow e_2\} \\
 & \mid & \mathbf{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) \mid \{\overline{e}\} \mid e_1 \cup e_2 \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) \\
\multicolumn{3}{l}{(\textit{values})} \\
v & ::= & \langle\rangle \mid s \mid \lambda x{:}\tau.e \mid \Lambda\alpha.e \mid \langle\overline{v}\rangle \mid \ell \mid \{v.\overline{\alpha}x{:}\tau \rightarrow e\} \mid \{\overline{v}\} \mid \bullet \mid \ell[\overline{\tau}][\![v]\!]{::}v \\
\multicolumn{3}{l}{(\textit{patterns})} \\
\rho & ::= & \bullet \mid e[\overline{\alpha}][\![y]\!]{:}\tau{::}\rho \mid x \mid \_{::}\rho \\
\multicolumn{3}{l}{(\textit{pattern values})} \\
\varphi & ::= & \bullet \mid v[\overline{\alpha}][\![y]\!]\tau{::}\varphi \mid x \mid \_{::}\varphi \\
\multicolumn{3}{l}{(\textit{evaluation contexts})} \\
E & ::= & [] \mid Ee \mid vE \mid E[\tau] \mid \langle E, \ldots, e\rangle \mid \langle v, \ldots, E\rangle \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\} \mid \mathbf{new}\ \overline{a}.\tau \leq E \mid \mathbf{store}\ E[\overline{\tau}][\![e_1]\!]\ \mathbf{in}\ e_2 \\
 & \mid & \mathbf{store}\ v[\overline{\tau}][\![E]\!]\ \mathbf{in}\ e \mid \mathbf{store}\ v_1[\overline{\tau}][\![v_2]\!]\ \mathbf{in}\ E \mid \{E, \ldots, e\} \mid \{v, \ldots, E\} \\
 & \mid & E \cup e \mid v \cup 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) \\
\multicolumn{3}{l}{(\textit{pattern evaluation contexts})} \\
P & ::= & E[\overline{\alpha}][\![y]\!]{:}\tau{::}\varphi \mid e[\overline{\alpha}][\![y]\!]{:}\tau{::}P \mid \_{::}P \\
\multicolumn{3}{l}{(\textit{type variable contexts})} \\
\Delta & ::= & \cdot \mid \Delta, \alpha \\
\multicolumn{3}{l}{(\textit{term variable and label contexts})} \\
\Gamma & ::= & \mathcal{U}{:}\alpha.\alpha \mid \Gamma, x{:}\tau \mid \Gamma, \ell{:}\overline{\alpha}.\tau \\
\multicolumn{3}{l}{(\textit{label heap})} \\
\Sigma & ::= & \mathcal{U}{:}\alpha.\alpha \leq \mathcal{U} \mid \Sigma, \ell{:}\overline{\alpha}.\tau \leq \ell' \\
\multicolumn{3}{l}{(\textit{advice heap})} \\
A & ::= & \cdot \mid A, \{v.\overline{\alpha}x{:}\tau \rightarrow e\} \\
\multicolumn{3}{l}{(\textit{substitutions})} \\
\Theta & ::= & \cdot \mid \Theta, \tau/\alpha \mid \Theta, e/x
\end{array}
$$

A.2 Static Semantics

A.2.1 Types

$$
\frac{\alpha \in \Delta}{\Delta \vdash \alpha}\ \mathsf{wftp{:}var}
\qquad
\frac{}{\Delta \vdash 1}\ \mathsf{wftp{:}unit}
\qquad
\frac{}{\Delta \vdash \mathsf{string}}\ \mathsf{wftp{:}str}
\qquad
\frac{\Delta \vdash \tau_1 \qquad \Delta \vdash \tau_2}{\Delta \vdash \tau_1 \rightarrow \tau_2}\ \mathsf{wftp{:}arr}
$$

$$
\frac{\Delta, \alpha \vdash \tau}{\Delta \vdash \forall\alpha.\tau}\ \mathsf{wftp{:}all}
\qquad
\frac{\Delta \vdash \tau_i}{\Delta \vdash \tau_1 \times \ldots \times \tau_n}\ \mathsf{wftp{:}prod}
\qquad
\frac{\Delta, \overline{\alpha} \vdash \tau}{\Delta \vdash (\overline{\alpha}.\tau)\ \mathsf{label}}\ \mathsf{wftp{:}lab}
$$

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

A.2.2 Generalization

$$\dfrac{\Delta, \overline{\alpha} \vdash \tau_1 \qquad \Delta, \overline{\beta} \vdash \tau_2 \qquad \Delta \vdash \tau_i \qquad \exists \overline{\tau}.\tau_1[\overline{\tau}/\overline{\alpha}] = \tau_2}{\Delta \vdash \overline{\alpha}.\tau_1 \prec \overline{\beta}.\tau_2}\ \mathsf{gen}$$

A.2.3 Label subsumption

$$\dfrac{\ell{:}\overline{\alpha}.\tau \leq \ell' \in \Sigma}{\Sigma \vdash \ell \leq \ell}\ \mathsf{labsb{:}refl} \qquad \dfrac{\Sigma \vdash \ell_1 \leq \ell_2 \qquad \Sigma \vdash \ell_2 \leq \ell_3}{\Sigma \vdash \ell_1 \leq \ell_3}\ \mathsf{labsb{:}trans} \qquad \dfrac{\ell_1{:}\overline{\alpha}.\tau \leq \ell_2 \in \Sigma}{\Sigma \vdash \ell_1 \leq \ell_2}\ \mathsf{labsb{:}def}$$

A.2.4 Term variable and Label Contexts

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

A.2.5 Label heaps

$$\dfrac{}{\vdash (\mathcal{U}{:}\alpha.\alpha \leq \mathcal{U}) : (\mathcal{U}{:}\alpha.\alpha)}\ \mathsf{wflh{:}base} \qquad \dfrac{\ell_2{:}\overline{\beta}.\tau_2 \leq \ell_3 \in \Sigma \qquad \cdot \vdash \overline{\beta}.\tau_2 \prec \overline{\alpha}.\tau_1 \qquad \vdash \Sigma : \Gamma}{\vdash (\Sigma, \ell_1{:}\overline{\alpha}.\tau_1 \leq \ell_2) : (\Gamma, \ell_1{:}\overline{\alpha}.\tau_1)}\ \mathsf{wflh{:}cons}$$

A.2.6 Advice heaps

$$\dfrac{}{\Gamma \vdash \cdot\ \mathrm{ok}}\ \mathsf{wfah{:}base} \qquad \dfrac{\cdot; \Gamma \vdash \nu : \mathsf{advice} \qquad \Gamma \vdash A\ \mathrm{ok}}{\Gamma \vdash A, \nu\ \mathrm{ok}}\ \mathsf{wfah{:}cons}$$

A.2.7 Terms

$$\dfrac{x{:}\tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}\ \mathsf{wft{:}var} \qquad \dfrac{}{\Delta; \Gamma \vdash \langle\rangle : 1}\ \mathsf{wft{:}unit} \qquad \dfrac{\Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2 \qquad \Delta \vdash \tau_1}{\Delta; \Gamma \vdash \lambda x{:}\tau_1.e : \tau_1 \to \tau_2}\ \mathsf{wft{:}abs}$$

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

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

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

$$\dfrac{\Delta; \Gamma \vdash e_i : (\overline{\alpha}_i.\tau_i)\ \mathsf{label} \qquad \Delta \vdash \overline{\beta}.\tau \prec \overline{\alpha}_i.\tau_i}{\Delta; \Gamma \vdash \{\overline{e}\} : (\overline{\beta}.\tau)\ \mathsf{pc}}\ \mathsf{wft{:}pc} \qquad \dfrac{\Delta; \Gamma \vdash e_i : (\overline{\alpha}.\tau_i)\ \mathsf{pc} \qquad \Delta \vdash \overline{\beta}.\tau \prec \overline{\alpha}.\tau_i}{\Delta; \Gamma \vdash e_1 \cup e_2 : (\overline{\beta}.\tau)\ \mathsf{pc}}\ \mathsf{wft{:}union}$$

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

$$\dfrac{\Delta; \Gamma \vdash e_1 : (\overline{\alpha}.\tau)\ \mathsf{label} \qquad \Delta \vdash \tau_i \qquad \Delta; \Gamma \vdash e_2 : \tau[\overline{\tau}/\overline{\alpha}]}{\Delta; \Gamma \vdash e_1[\overline{\tau}]\llbracket e_2 \rrbracket : \tau[\overline{\tau}/\overline{\alpha}]}\ \mathsf{wft{:}cut} \qquad \dfrac{\Delta; \Gamma \vdash e : \mathsf{advice}}{\Delta; \Gamma \vdash\Uparrow e : 1}\ \mathsf{wft{:}adv\text{-}inst}$$

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

$$\frac{\Delta,\alpha \vdash \tau_1 \qquad \Delta \vdash \tau_2 \qquad \Delta' = \mathrm{FTV}(\tau_3) \qquad \Delta,\Delta';\Gamma \vdash e_1[\tau_3/\alpha] : \tau_1[\tau_3/\alpha] \qquad \Delta,\alpha;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash \boldsymbol{typecase}[\alpha.\tau_1]\ \tau_2\ (\tau_3 \Rightarrow e_1, \alpha \Rightarrow e_2) : \tau_1[\tau_2/\alpha]}\ \textsf{wft:tcase}$$

$$\frac{\Delta;\Gamma \vdash e_1 : (\overline{\alpha}.\tau)\ \textsf{label} \qquad \Delta \vdash \tau_i \qquad \Delta;\Gamma \vdash e_2 : \tau[\overline{\tau}/\overline{\alpha}] \qquad \Delta;\Gamma \vdash e_3 : \tau'}{\Delta;\Gamma \vdash \boldsymbol{store}\ e_1[\overline{\tau}]\llbracket e_2 \rrbracket\ \boldsymbol{in}\ e_3 : \tau'}\ \textsf{wft:store}$$

$$\frac{}{\Delta;\Gamma \vdash \boldsymbol{stack} : \textsf{stack}}\ \textsf{wft:stk} \qquad\qquad \frac{}{\Delta;\Gamma \vdash \bullet : \textsf{stack}}\ \textsf{wft:stk-nil}$$

$$\frac{\ell{:}\overline{\alpha}.\tau \in \Gamma \qquad \Delta \vdash \tau_i \qquad \Delta;\Gamma \vdash v_1 : \tau[\overline{\tau}/\overline{\alpha}] \qquad \Delta;\Gamma \vdash v_2 : \textsf{stack}}{\Delta;\Gamma \vdash \ell[\overline{\tau}]\llbracket v_1 \rrbracket{::}v_2 : \textsf{stack}}\ \textsf{wft:stk-cons}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \textsf{stack}\\ \Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma' \qquad \Gamma',\Delta'\ \text{linear} \qquad \Delta,\Delta';\Gamma,\Gamma' \vdash e_2 : \tau \qquad \Delta;\Gamma,x{:}\textsf{stack} \vdash e_3 : \tau\end{array}}{\Delta;\Gamma \vdash \boldsymbol{stkcase}\ e_1\ (\rho \Rightarrow e_2, x \Rightarrow e_3) : \tau}\ \textsf{wft:scase}$$

A.2.8 Patterns

$$\frac{}{\Delta;\Gamma \vdash \bullet \dashv \cdot;\cdot}\ \textsf{wfpt:nil} \qquad \frac{}{\Delta;\Gamma \vdash x \dashv \cdot;\cdot, x{:}\textsf{stack}}\ \textsf{wfpt:var} \qquad \frac{\Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma'}{\Delta;\Gamma \vdash \_{::}\rho \dashv \Delta';\Gamma'}\ \textsf{wfpt:wild}$$

$$\frac{\Delta;\Gamma \vdash e : (\overline{\alpha}.\tau)\ \textsf{pc} \qquad \Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma'}{\Delta;\Gamma \vdash e[\overline{\alpha}]\llbracket x \rrbracket{:}\tau{::}\rho \dashv \Delta',\overline{\alpha};\Gamma', x : \tau}\ \textsf{wfpt:store}$$

A.2.9 Machine configurations

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

A.3 Dynamic Semantics

A.3.1 Stack Data

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

A.3.2 β-reductions

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

$$\frac{}{\Sigma; A; \mathbf{let}\ \langle\overline{x}\rangle = \langle\overline{v}\rangle\ \mathbf{in}\ e \mapsto_\beta \Sigma; A; e[\overline{v}/\overline{x}]}\ \textsf{evb:let} \qquad \frac{}{\Sigma; A; \{\overline{\ell_1}\} \cup \{\overline{\ell_2}\} \mapsto_\beta \Sigma; A; \{\overline{\ell_1 \ell_2}\}}\ \textsf{evb:union}$$

$$\frac{\ell' \notin \mathrm{dom}(\Sigma)}{\Sigma; A; \boldsymbol{new}\ \overline{\alpha}.\tau \leq \ell \mapsto_\beta \Sigma, \ell'{:}\overline{\alpha}.\tau \leq \ell; A; \ell'}\ \textsf{evb:new} \qquad \frac{}{\Sigma; A; \Uparrow v \mapsto_\beta \Sigma; A, v; \langle\rangle}\ \textsf{evb:adv-comp}$$

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

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

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

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

$$\frac{}{\Sigma; A; \mathbf{store}\ \ell[\overline{\tau}]\llbracket v_1 \rrbracket\ \mathbf{in}\ v_2 \mapsto_\beta \Sigma; A; v_2}\ \textsf{evb:store} \qquad \frac{\ell{:}\overline{\alpha}.\tau \leq \ell' \in \Sigma \qquad \Sigma; A; \ell; \tau[\overline{\tau}/\overline{\alpha}] \Rightarrow v'}{\Sigma; A; \ell[\overline{\tau}]\llbracket v \rrbracket \mapsto_\beta \Sigma; A; v'\,v}\ \textsf{evb:cut}$$

A.3.3 Context reductions

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

A.3.4 Stack matching

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

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

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

A.3.5 Advice composition

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

$$\frac{\Sigma; A; \ell; \tau_2 \Rightarrow v_2 \qquad \Sigma \vdash \ell \leq \ell_i \text{ for some } i \qquad \exists \overline{\tau}.\tau_2 = \tau_1[\overline{\tau}/\overline{\alpha}]}{\Sigma; A, \{\{\overline{\ell}\}.\overline{\alpha}x{:}\tau_1 \rightarrow e\}; \ell; \tau_2 \Rightarrow \lambda x{:}\tau.v_2(e[\overline{\tau}/\overline{\alpha}])}\ \textsf{adv:cons1}$$

$$\frac{\Sigma; A; \ell; \tau_2 \Rightarrow v_2 \qquad \Sigma \vdash \ell \not\leq \ell_i}{\Sigma; A, \{\{\overline{\ell}\}.\overline{\alpha}x{:}\tau_1 \rightarrow e\}; \ell; \tau_2 \Rightarrow v_2}\ \textsf{adv:cons2}$$

B The meta-theory of $\mathbb{F}_A$

Lemma B.1 (Inversion). *The rules in the following judgments are invertible: well-formed types, generalization, variable contexts, label heaps, advice heaps, term typing, patterns, machine configurations, stack data, β -reductions, context reductions, and stack matching. The rules in the judgements for the label subsumption and advice composition rules are not invertible.*

Proof. By inspection of the rules for each judgement. □

Lemma B.2 (Label subsumption). *If* $\vdash \Sigma : \Gamma$ *and* $\Sigma \vdash \ell_1 \leq \ell_2$ *then* $\ell_1{:}\overline{\alpha}.\tau_1 \leq \ell_1' \in \Sigma$ *and* $\ell_2{:}\overline{\beta}.\tau_2 \leq \ell_2' \in \Sigma$.

Proof. Straightforward induction on the structure of $\Sigma \vdash \ell_1 \leq \ell_2$. □

Lemma B.3 (Label generalization). *If* $\vdash \Sigma : \Gamma$ *and* $\Sigma \vdash \ell_1 \leq \ell_2$ *and* $\ell_1{:}\overline{\alpha}.\tau_1 \leq \ell_1' \in \Sigma$ *and* $\ell_2{:}\overline{\beta}.\tau_2 \leq \ell_2' \in \Sigma$ *then* $\cdot \vdash \overline{\beta}.\tau_2 \prec \overline{\alpha}.\tau_1$.

Proof. By induction on the structure of $\Sigma \vdash \ell_1 \leq \ell_2$, with use of Lemma B.1 and B.2. □

Lemma B.4 (Generalization transitivity). *If* $\Delta \vdash \overline{\alpha}.\tau_1 \prec \overline{\beta}.\tau_2$ *and* $\Delta \vdash \overline{\beta}.\tau_2 \prec \overline{\gamma}.\tau_3$ *then* $\Delta \vdash \overline{\alpha}.\tau_1 \prec \overline{\gamma}.\tau_3$.

Proof. Straightforward, with uses of Lemma B.1. □

Lemma B.5 (Point cut match progress). *If* $\vdash \Sigma : \Gamma$ *and* $(\cdot \vdash \overline{\tau_i})^{1 \leq i \leq n}$ *and* $\ell{:}\overline{a}.\tau \leq \ell' \in \Sigma$ *and* $\cdot; \Gamma \vdash \{\overline{\ell}\} : (\overline{\beta}.\tau')\ \textsf{pc}$ *and* $\Sigma \vdash \ell \leq \ell_j$ *and* $(\cdot \vdash \overline{\tau_i'})^{1 \leq i \leq n}$ *then* $\tau[\overline{\tau}/\overline{a}] = \tau'[\overline{\tau'}/\overline{\beta}]$.

Proof. Straightforward, with uses of Lemma B.1, B.3 and B.4. □

Lemma B.6 (Cut progress). *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A, \{\{\overline{\ell}\}.\overline{\beta}x{:}\tau' \rightarrow e\}$ *ok and* $\cdot; \Gamma \vdash \ell[\overline{\tau}][\![v]\!] : \tau[\overline{\tau}/\overline{\alpha}]$ *and* $\Sigma \vdash \ell \leq \ell_j$ *and* $(\cdot \vdash \overline{\tau_i'})^{1 \leq i \leq n}$ *then* $\tau[\overline{\tau}/\overline{a}] = \tau'[\overline{\tau'}/\overline{\beta}]$.

Proof. Straightforward use of Lemma B.5, with uses of Lemma B.1. □

Lemma B.7 (Stack-case progress). *If* $\vdash \Sigma : \Gamma$ *and* $\cdot; \Gamma \vdash \textbf{stkcase}\ \ell[\overline{\tau}][\![v_1]\!]{::}v_2\ (\{\overline{\ell}\}[\overline{\beta}][\![x]\!]{:}\tau'{::}\rho \Rightarrow e_2, x \Rightarrow e_3) : \tau$ *and* $\Sigma \vdash \ell \leq \ell_j$ *and* $(\cdot \vdash \overline{\tau_i'})^{1 \leq i \leq n}$ *then* $\tau[\overline{\tau}/\overline{a}] = \tau'[\overline{\tau'}/\overline{\beta}]$.

Proof. Straightforward use of Lemma B.5, with uses of Lemma B.1. □

Lemma B.8 (Canonical forms). *Suppose that* $v : \tau$ *is a closed, well-formed value and* τ *is a closed, well-formed type.*

- *If* $\tau = 1$, *then* $v = \langle\rangle$.
- *If* $\tau = \textsf{string}$, *then* $v = s$.

- *If* $\tau = \tau_1 \rightarrow \tau_2$*, then* $v = \lambda x{:}\tau_1.e$.
- *If* $\tau = \forall\alpha.\tau'$*, then* $v = \Lambda\alpha.e$.
- *If* $\tau = (\overline{\alpha}.\tau')$ label*, then* $v = \ell$.
- *If* $\tau = (\overline{\alpha}.\tau')$ pc*, then* $v = \{\overline{v}\}$.
- *If* $\tau =$ advice*, then* $v = \{v'.\overline{\alpha}x{:}\tau' \rightarrow e\}$.
- *If* $\tau =$ stack*, then either* $v = \bullet$ *or* $\ell[\overline{\tau'}][\![v']\!]{::}v''$.
- *If* $\tau = \tau_1 \times \ldots \times \tau_n$*, then* $v = \langle\overline{v}\rangle$.

Proof. By induction on the structure of $\Delta;\Gamma \vdash v : \tau$, using the fact that v is a value. □

Lemma B.9 (Context decomposition). *If* $\vdash \Sigma : \Gamma$ *and* $\cdot;\Gamma \vdash e : t$ *then* e *is a value or* $E[e']$ *where* e' *is either* $\mathtt{stack}$ *or the left-hand side of one of the* β*-reduction rules.*

Proof. By induction on on the structure of $\cdot;\Gamma \vdash e : t$ □

Lemma B.10 (Progress lemma). *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ *ok and* $\cdot;\Gamma \vdash e : \tau$ *then either* e *is a value, or there exists another configuration* $\Sigma';A';e'$ *such that* $\Sigma;A;e \mapsto \Sigma';A';e'$.

Proof. By induction on the structure of $\Delta;\Gamma \vdash e : \tau$, with uses of Lemma B.1, B.6, B.7, B.8, and B.9. □

Theorem B.11 (Progress). *If* $\vdash (\Sigma;A;e)$ *ok then either* e *is a value, or there exists another configuration* $\Sigma';A';e'$ *such that* $\Sigma;A;e \mapsto \Sigma';A';e'$.

Proof. Straightforward use of Lemma B.10, with uses of Lemma B.1. □

Definition B.12 (Γ' extends Γ). *If* $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$ *and* $\forall x \in \mathrm{dom}(\Gamma), \Gamma(x) = \Gamma'(x)$*, and* $\forall l \in \mathrm{dom}(\Gamma), \Gamma(l) = \Gamma'(l)$*, then* Γ' *extends* Γ.

Definition B.13 (Σ' extends Σ). *If* $\mathrm{dom}(\Sigma) \subseteq \mathrm{dom}(\Sigma')$ *and* $\forall l \in \mathrm{dom}(\Sigma), \Sigma(l) = \Sigma'(l)$*, then* Σ' *extends* Σ.

Definition B.14 (A' extends A). *If* $\forall v \in A,\ v \in A'$*, then* A' *extends* A.

Lemma B.15 (Evaluation context inversion). *If* $\Delta;\Gamma \vdash E[e] : \tau$ *then* $\Delta;\Gamma \vdash e : \tau'$.

Proof. By induction on the structure of E, with uses of Lemma B.1. □

Lemma B.16 (Evaluation context substitution). *If* $\Delta;\Gamma \vdash E[e] : \tau$ *and* $\Delta;\Gamma \vdash e : \tau'$ *and* $\Delta;\Gamma' \vdash e' : \tau'$ *and* Γ' *extends* Γ *then* $\Delta;\Gamma' \vdash E[e'] : \tau$.

Proof. By induction on the structure of E, with uses of Lemma B.1. □

Lemma B.17 (Data function typing). *If* $\cdot;\Gamma \vdash E[e] : \tau$ *and* $\mathtt{data}(E) = v$ *then* $\cdot;\Gamma \vdash v :$ stack.

Proof. By induction on the structure of the $\mathtt{data}(E)$ function, with uses of Lemma B.1 and B.15. □

Lemma B.18 (Pattern matching). *If* $\vdash \Sigma : \Gamma''$ *and* Γ *extends* Γ'' *and* $\Delta \vdash \Gamma$ *and* $\Delta;\Gamma \vdash v :$ stack *and* $\Delta;\Gamma \vdash \rho \dashv \Delta';\Gamma'$ *and* $\Delta,\Delta';\Gamma,\Gamma' \vdash e : \tau$ *and* $\Sigma \vdash v \simeq \rho \rhd \Theta$ *then* $\Delta;\Gamma \vdash \Theta(e) : \tau$.

Proof. By induction on the structure of $\Sigma \vdash v \simeq \rho \rhd \Theta$, with uses of Lemma B.1. □

Lemma B.19 (Advice composition). *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ *ok and* $\cdot;\Gamma \vdash \ell : (\overline{\alpha}.\tau)$ label *and* $\Sigma;A;\ell;\tau[\overline{\tau}/\overline{\alpha}] \Rightarrow v$ *and* $(\cdot \vdash \tau_i)^{1 \leq i \leq n}$ *then* $\cdot;\Gamma \vdash v : \tau[\overline{\tau}/\overline{\alpha}] \rightarrow \tau[\overline{\tau}/\overline{\alpha}]$.

Proof. By induction on the structure of $\Sigma; A; \ell; \tau[\overline{\tau}/\overline{\alpha}] \Rightarrow \nu$, with uses of Lemma B.1. □

Lemma B.20 (β-redux preservation). *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ *ok and* $\cdot; \Gamma \vdash e : \tau$ *and* $\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'$ *then* $\vdash \Sigma' : \Gamma'$ *and* $\Gamma' \vdash A'$ *ok and* $\cdot; \Gamma' \vdash e' : \tau$ *and* Γ' *extends* Γ.

Proof. By induction on the structure of $\Sigma; A; e \mapsto_\beta \Sigma'; A'; e'$, with uses of Lemma B.1, B.4, B.18, and B.19. □

Lemma B.21 (Preservation lemma). *If* $\vdash \Sigma : \Gamma$ *and* $\Gamma \vdash A$ *ok and* $\cdot; \Gamma \vdash e : \tau$ *and* $\Sigma; A; e \mapsto \Sigma'; A'; e'$ *then* $\vdash \Sigma' : \Gamma'$ *and* $\Gamma' \vdash A'$ *ok and* $\cdot; \Gamma' \vdash e' : \tau$.

Proof. By induction on the structure of $\Sigma; A; e \mapsto \Sigma'; A'; e'$, with uses of Lemma B.1, B.15, B.16, B.17, and B.20. □

Theorem B.22 (Preservation). *If* $\vdash (\Sigma; A; e)$ *ok and* $\Sigma; A; e \mapsto \Sigma'; A'; e'$*, then* Σ' *and* A' *extend* Σ *and* A *such that* $\vdash (\Sigma'; A'; e')$ *ok.*

Proof. Straightforward use of Lemma B.21, with uses of Lemma B.1. □

C Translation

C.1 Polytypes

$$\frac{\Delta, \overline{\mathsf{a}} \vdash \mathtt{t} \overset{\mathsf{type}}{\Longrightarrow} \tau'}{\Delta \vdash \mathtt{forall}\ \overline{\mathtt{a}}.\mathtt{t} \overset{\mathsf{type}}{\Longrightarrow} \forall\overline{\alpha}.\tau'}\ \textsf{tpy:all}$$

C.2 Monotypes

$$\frac{\mathsf{a} \in \Delta}{\Delta \vdash \mathsf{a} \overset{\mathsf{type}}{\Longrightarrow} \alpha}\ \textsf{ttp:var} \qquad \frac{}{\Delta \vdash \mathtt{unit} \overset{\mathsf{type}}{\Longrightarrow} 1}\ \textsf{ttp:unit} \qquad \frac{}{\Delta \vdash \mathtt{string} \overset{\mathsf{type}}{\Longrightarrow} \mathsf{string}}\ \textsf{ttp:str}$$

$$\frac{}{\Delta \vdash \mathtt{stack} \overset{\mathsf{type}}{\Longrightarrow} \mathsf{stack}}\ \textsf{ttp:stk} \qquad \frac{\Delta \vdash \mathtt{t}_1 \overset{\mathsf{type}}{\Longrightarrow} \tau_1' \qquad \Delta \vdash \mathtt{t}_2 \overset{\mathsf{type}}{\Longrightarrow} \tau_1'}{\Delta \vdash \mathtt{t}_1\ \texttt{->}\ \mathtt{t}_2 \overset{\mathsf{type}}{\Longrightarrow} \tau_1' \to \tau_2'}\ \textsf{ttp:fun}$$

C.3 Pattern splitting helper

$$\begin{array}{rcl} \mathrm{split}(\cdot, e) & = & e \\ \mathrm{split}(\Phi, x \mapsto (y, z), e) & = & \mathrm{split}(\Phi, \mathbf{let}\ \langle y, z\rangle = x\ \mathbf{in}\ e) \end{array}$$

C.4 Terms

$$\frac{\mathtt{x{:}t} \in \Gamma}{\Delta;\Gamma \vdash \mathtt{x} : \mathtt{t} \overset{\text{exp}}{\Longrightarrow} x}\ \textsf{ttm:var} \qquad\qquad \frac{}{\Delta;\Gamma \vdash \mathtt{()} : \mathtt{unit} \overset{\text{exp}}{\Longrightarrow} \langle\rangle}\ \textsf{ttm:unit}$$

$$\frac{\mathtt{f{:}forall}\ \overline{\mathtt{a}}\mathtt{.t} \in \Gamma \qquad \Delta \vdash \mathtt{t}_i \overset{\text{type}}{\Longrightarrow} \tau'_i}{\Delta;\Gamma \vdash \mathtt{f}\,[\overline{\mathtt{t}}] : \mathtt{t}[\overline{\mathtt{t}}/\overline{\mathtt{a}}] \overset{\text{exp}}{\Longrightarrow} f[\overline{\tau'}]}\ \textsf{ttm:inst}$$

$$\frac{\Delta;\Gamma \vdash \mathsf{e}_1 : \mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2 \overset{\text{exp}}{\Longrightarrow} e'_1 \qquad \Delta;\Gamma \vdash \mathsf{e}_2 : \mathtt{t}_1 \overset{\text{exp}}{\Longrightarrow} e'_2}{\Delta;\Gamma \vdash \mathsf{e}_1\mathsf{e}_2 : \mathtt{t}_2 \overset{\text{exp}}{\Longrightarrow} e'_1 e'_2}\ \textsf{ttm:app}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathsf{e}_1 : \mathtt{stack} \overset{\text{exp}}{\Longrightarrow} e'_1 \\ \Delta;\Gamma \vdash \mathtt{p}_i \overset{\text{pat}}{\Longrightarrow} \rho'_i \dashv \Delta_i;\Gamma_i;\Phi_i \qquad \Delta_i,\Gamma_i\ \text{linear} \qquad \Delta,\Delta_i;\Gamma,\Gamma_i \vdash \mathsf{e}_i : \mathtt{t} \overset{\text{exp}}{\Longrightarrow} e'_i \qquad \Delta;\Gamma \vdash \mathsf{e}_2 : \mathtt{t} \overset{\text{exp}}{\Longrightarrow} e'_2\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{stkcase}\ \mathsf{e}_1\ \ (\overline{\mathtt{p=>e}}\ \mathtt{|\_=>}\ \mathsf{e}_2) : \mathtt{t} \overset{\text{e}}{\Longrightarrow} \\ \boldsymbol{stkcase}\ e'_1\ (\overline{\rho'} \Rightarrow \overline{\mathit{split}(\Phi, e')}, x \Rightarrow e'_2)\end{array}}\ \textsf{ttm:scase}$$

$$\frac{\begin{array}{c}\mathsf{a} \in \Delta \qquad \Delta \vdash \mathtt{t} \overset{\text{type}}{\Longrightarrow} \tau' \\ \Delta \vdash \mathtt{t}_i \overset{\text{type}}{\Longrightarrow} \tau'_i \qquad \Delta_i = \mathrm{FTV}(\mathtt{t}) \qquad \Delta,\Delta_i;\Gamma \vdash \mathsf{e}_i[\mathtt{t}_i/\mathsf{a}] : \mathtt{t}[\mathtt{t}_i/\mathsf{a}] \overset{\text{exp}}{\Longrightarrow} e'_i \qquad \Delta;\Gamma \vdash \mathsf{e} : \mathtt{t} \overset{\text{exp}}{\Longrightarrow} e'\end{array}}{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{typecase}\ \mathsf{a}\ (\overline{\mathtt{t=>e}}\ \mathtt{|\_=>}\ \mathsf{e}) : \mathtt{t} \overset{\text{e}}{\Longrightarrow} \\ \boldsymbol{typecase}[\alpha.\tau']\ \alpha\ (\overline{\tau'} \Rightarrow \overline{e'}, \alpha \Rightarrow e')\end{array}}\ \textsf{ttm:tcase}$$

$$\frac{\Delta;\Gamma \vdash \mathsf{ds};\mathsf{e} : \mathtt{t} \overset{\text{decs}}{\Longrightarrow} e'}{\Delta;\Gamma \vdash \mathsf{ds}\ \ \mathsf{e} : \mathtt{t} \overset{\text{exp}}{\Longrightarrow} e'}\ \textsf{ttm:ds}$$

C.5 Point cut designators

$$\frac{\mathtt{time} \in \{\mathtt{before},\mathtt{stk}\} \qquad \mathtt{f}_i\mathtt{{:}forall}\ \overline{\mathtt{a}}_i\,\mathtt{.t}_{1,i}\ \mathtt{->}\ \mathtt{t}_{2,i} \in \Gamma \qquad \Delta \vdash \overline{\mathsf{b}}.\mathsf{t} \prec \overline{\mathsf{a}}_i.\mathtt{t}_{1,i}}{\Delta;\Gamma \vdash \{\overline{\mathtt{f}}\} \overset{\mathtt{time}}{\Longrightarrow} \{\overline{\mathsf{f}_{\mathtt{time}}}\};\overline{\mathsf{b}}.\mathsf{t}}\ \textsf{tpt:set-befstk}$$

$$\frac{\mathtt{f}_i\mathtt{{:}forall}\ \overline{\mathtt{a}}_i\,\mathtt{.t}_{1,i}\ \mathtt{->}\ \mathtt{t}_{2,i} \in \Gamma \qquad \Delta \vdash \overline{\mathsf{b}}.\mathsf{t} \prec \overline{\mathsf{a}}_i.\mathtt{t}_{2,i}}{\Delta;\Gamma \vdash \{\overline{\mathtt{f}}\} \overset{\mathtt{after}}{\Longrightarrow} \{\overline{\mathsf{f}_{\mathtt{after}}}\};\overline{\mathsf{b}}.\mathsf{t}}\ \textsf{tpt:set-aft} \qquad\qquad \frac{}{\Delta;\Gamma \vdash \mathsf{any} \overset{\mathtt{time}}{\Longrightarrow} \{\mathcal{U}_{\mathtt{time}}\};\mathsf{a.a}}\ \textsf{tpt:any}$$

C.6 Patterns

$$\frac{}{\Delta;\Gamma \vdash \mathtt{nil} \overset{\text{pat}}{\Longrightarrow} \bullet \dashv \cdot;\cdot;\cdot}\ \textsf{tpat:nil} \qquad\qquad \frac{}{\Delta;\Gamma \vdash \mathtt{x} \overset{\text{pat}}{\Longrightarrow} x \dashv \cdot;\cdot,\mathtt{x{:}stack};\cdot}\ \textsf{tpat:var}$$

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

$$\frac{\Delta;\Gamma \vdash \mathtt{pt} \overset{\text{stk}}{\Longrightarrow} e';\overline{\mathsf{a}}.\mathsf{t} \qquad \Delta;\Gamma \vdash \mathtt{p} \overset{\text{pat}}{\Longrightarrow} \rho' \dashv \Delta';\Gamma';\Phi \qquad y\ \text{fresh}}{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{pt(x:t,n)::p} \overset{\text{p}}{\Longrightarrow} \\ e'[\overline{\alpha}][\![y]\!]{::}\rho' \dashv \Delta',\overline{\mathsf{a}};\Gamma',\mathtt{x{:}t},\mathtt{n{:}string};\Phi, y \mapsto (x,n)\end{array}}\ \textsf{tpat:cons}$$

C.7 Declarations

$$\frac{\Delta;\Gamma \vdash \mathsf{e} : \mathsf{t} \overset{\mathtt{exp}}{\Longrightarrow} e'}{\Delta;\Gamma \vdash .;\mathsf{e} : \mathsf{t} \overset{\mathtt{decs}}{\Longrightarrow} e'}\ \mathsf{tds{:}tm}$$

$$\frac{\begin{array}{c}\overline{\mathtt{a}} = \mathrm{FTV}(\mathtt{t}_1,\mathtt{t}_2) - \Delta \qquad \Delta,\overline{\mathtt{a}} \vdash \mathtt{t}_1 \overset{\mathtt{type}}{\Longrightarrow} \tau_1' \\ \Delta,\overline{\mathtt{a}} \vdash \mathtt{t}_2 \overset{\mathtt{type}}{\Longrightarrow} \tau_2' \qquad \Delta;\Gamma,\mathtt{f{:}forall}\ \overline{\mathtt{a}}.\mathtt{t}_1\ \mathtt{->}\ \mathtt{t}_2 \vdash \mathsf{ds};\mathsf{e}_2 : \mathsf{t} \overset{\mathtt{decs}}{\Longrightarrow} e_2' \qquad \Delta,\overline{\mathtt{a}};\Gamma,\mathtt{x{:}t}_1 \vdash \mathsf{e}_1 : \mathtt{t}_2 \overset{\mathtt{exp}}{\Longrightarrow} e_1'\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash \mathtt{let\ f\ (x{:}t_1){:}t_2\ =\ e_1\ \ in}\ \mathsf{ds};\mathsf{e}_2 : \mathsf{t} \overset{\mathtt{ds}}{\Longrightarrow} \\ \quad \mathbf{let}\ \mathsf{f}_{\mathtt{before}} : (\overline{\alpha}.\tau_1' \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{stack} \times \mathsf{string}) \leq \mathcal{U}_{\mathtt{before}}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathsf{f}_{\mathtt{after}} : (\overline{\alpha}.\tau_2' \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.\tau_2' \times \mathsf{stack} \times \mathsf{string}) \leq \mathcal{U}_{\mathtt{after}}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathsf{f}_{\mathtt{stk}} : (\overline{\alpha}.\tau_1' \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\overline{\alpha}.\tau_1' \times \mathsf{string}) \leq \mathcal{U}_{\mathtt{stk}}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathsf{f} : \forall\overline{\alpha}.\tau_1' \to \tau_2' = \\ \qquad \Lambda\overline{\alpha}.\lambda x{:}\tau_1.\mathbf{store}\ \mathsf{f}_{\mathtt{stk}}[\overline{\alpha}][\![\langle x, \text{“f”}\rangle]\!]\ \mathbf{in} \\ \qquad\quad \mathbf{let}\ \langle x, \_, \_\rangle = \mathsf{f}_{\mathtt{before}}[\overline{\alpha}][\![\langle x, \mathbf{stack}, \text{“f”}\rangle]\!]\ \mathbf{in} \\ \qquad\qquad \mathbf{let}\ \langle x, \_, \_\rangle = \mathsf{f}_{\mathtt{after}}[\overline{\alpha}][\![\langle e_1', \mathbf{stack}, \text{“f”}\rangle]\!]\ \mathbf{in}\ x \\ \quad \mathbf{in}\ e_2'\end{array}}\ \mathsf{tds{:}let}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathsf{ds};\mathsf{e}_2 : \mathtt{t}_2 \overset{\mathtt{decs}}{\Longrightarrow} e_2' \qquad \Delta;\Gamma \vdash \mathtt{pt} \overset{\mathtt{time}}{\Longrightarrow} e';\overline{\mathtt{a}}.\mathtt{t}_3 \qquad \exists\overline{\mathtt{t}}.\mathtt{t}_3[\overline{\mathtt{t}}/\overline{\mathtt{a}}] = \mathtt{t}_1 \qquad \Delta' = \mathrm{FTV}(\mathtt{t}_1) \\ \Delta,\Delta' \vdash \mathtt{t}_1 \overset{\mathtt{type}}{\Longrightarrow} \tau_1' \qquad \Delta,\overline{\mathtt{a}} \vdash \mathtt{t}_3 \overset{\mathtt{type}}{\Longrightarrow} \tau_3' \qquad \Delta,\Delta';\Gamma,\mathtt{x{:}t}_1,\mathtt{s{:}stack},\mathtt{n{:}string} \vdash \mathsf{e}_1 : \mathtt{t}_1 \overset{\mathtt{exp}}{\Longrightarrow} e_1'\end{array}}{\begin{array}{l}\Delta;\Gamma \vdash \mathtt{time\ pt(x{:}t_1,s,n)\ \ =\ e_1\ \ in}\ \mathsf{ds};\mathsf{e}_2 : \mathtt{t}_2 \overset{\mathtt{ds}}{\Longrightarrow} \\ \quad \mathbf{let}\ \_ : 1 = \Uparrow \{e'.\overline{\alpha}x{:}\tau_3' \to \mathbf{let}\ \langle x, s, n\rangle = x\ \mathbf{in} \\ \qquad (\mathbf{typecase}[\gamma.\gamma \to \gamma]\ \tau_3'\ (\tau_1' \Rightarrow \lambda x{:}\tau_1'.e_1', \gamma \Rightarrow \lambda x{:}\gamma.x))x\} \\ \quad \mathbf{in}\ e_2'\end{array}}\ \mathsf{tds{:}ad}$$

C.8 Programs

$$\frac{\Delta;\Gamma \vdash \mathsf{ds};\mathsf{e} : \mathsf{t} \overset{\mathtt{decs}}{\Longrightarrow} e'}{\begin{array}{l}\Delta;\Gamma \vdash \mathsf{ds}\ \ \mathsf{e} : \mathsf{t} \overset{\mathtt{prog}}{\Longrightarrow} \\ \quad \mathbf{let}\ \mathcal{U}_{\mathtt{before}} : (\alpha.\alpha \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\alpha.\alpha \times \mathsf{stack} \times \mathsf{string}) \leq \mathcal{U}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathcal{U}_{\mathtt{after}} : (\alpha.\alpha \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\alpha.\alpha \times \mathsf{stack} \times \mathsf{string}) \leq \mathcal{U}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathcal{U}_{\mathtt{stk}} : (\alpha.\alpha \times \mathsf{string})\ \mathsf{label} = \\ \qquad \mathbf{new}\ (\alpha.\alpha \times \mathsf{string}) \leq \mathcal{U}\ \mathbf{in}\ e'\end{array}}\ \mathsf{tprog}$$

D The meta-theory of the translation

Definition D.1 (Simple abbreviations).

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

Definition D.2 (Multi-arm $\mathbf{stkcase}$ abbreviation).

$$\begin{array}{l}
\mathbf{stkcase}\ e_1'\ (\overline{\rho} \Rightarrow \overline{e}, x \Rightarrow e_2') \triangleq \\
\quad \mathbf{let}\ y : \mathsf{stack} = e_1'\ \mathbf{in}\ \mathbf{stkcase}\ y \quad \begin{array}[t]{lll} (\rho_1 & \rightarrow & e_1 \\ \_ & \rightarrow & \ldots(\mathbf{stkcase}\ y \quad \begin{array}[t]{lll}(\rho_n & \rightarrow & e_n \quad)\ldots) \\ x & \rightarrow & e_2')\end{array}\end{array}
\end{array}$$

(*where* y *fresh*)

Definition D.3 (Multi-arm $\mathbf{typecase}$ abbreviation).

$$\begin{array}{l}
\mathbf{typecase}[\alpha.\tau_1']\ \alpha\ (\overline{\tau} \Rightarrow \overline{e}, \alpha \Rightarrow e') \triangleq \\
\quad \mathbf{typecase}[\alpha.\tau_1']\ \alpha \quad \begin{array}[t]{lll} (\tau_1 & \rightarrow & e_1 \\ \alpha & \rightarrow & \ldots(\mathbf{typecase}[\alpha.\tau_1']\ \alpha \quad \begin{array}[t]{lll}(\tau_n & \rightarrow & e_n \quad)\ldots) \\ \alpha & \rightarrow & e')\end{array}\end{array}
\end{array}$$

Definition D.4 (Type variable context translation).

$$\frac{\forall \mathsf{a} \in \Delta \Leftrightarrow \alpha \in \Delta'}{\Delta \Longrightarrow \Delta'}\ \textsf{ttctx}$$

Definition D.5 (Term variable context translation).

$$\frac{\Delta \Longrightarrow \Delta'}{\Delta \vdash \cdot \Longrightarrow \begin{array}[t]{l}\mathcal{U}_{\mathtt{before}}{:}(\alpha.\alpha \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label}, \\ \mathcal{U}_{\mathtt{after}}{:}(\alpha.\alpha \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label}, \\ \mathcal{U}_{\mathtt{stk}}{:}(\alpha.\alpha \times \mathsf{string})\ \mathsf{label}\end{array}}\ \textsf{tctx:empty}$$

$$\frac{\Delta \vdash \Gamma \Longrightarrow \Gamma' \qquad \Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau}{\Delta \vdash \Gamma, \mathtt{x}{:}\mathtt{t} \Longrightarrow \Gamma', x{:}\tau}\ \textsf{tctx:mono}$$

$$\frac{\Delta \vdash \Gamma \Longrightarrow \Gamma' \qquad \Delta, \overline{\mathtt{a}} \vdash \mathtt{t}_1 \overset{\mathtt{type}}{\Longrightarrow} \tau_1 \qquad \Delta, \overline{\mathtt{a}} \vdash \mathtt{t}_2 \overset{\mathtt{type}}{\Longrightarrow} \tau_2}{\Delta \vdash \Gamma, \mathtt{f}{:}\mathtt{forall}\ \overline{\mathtt{a}}.\mathtt{t}_1\ \mathtt{\text{->}}\ \mathtt{t}_2 \Longrightarrow \Gamma', \begin{array}[t]{l} f_{\mathtt{before}}{:}(\overline{\alpha}.\tau_1 \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label}, \\ f_{\mathtt{after}}{:}(\overline{\alpha}.\tau_2 \times \mathsf{stack} \times \mathsf{string})\ \mathsf{label}, \\ f_{\mathtt{stk}}{:}(\overline{\alpha}.\tau_1 \times \mathsf{string})\ \mathsf{label}, \\ f{:}\forall\overline{\alpha}.\tau_1 \rightarrow \tau_2\end{array}}\ \textsf{tctx:poly}$$

Definition D.6 (Splitting context translation).

$$\frac{}{\Delta; \cdot \vdash \cdot \Longrightarrow \cdot}\ \textsf{tsctx:empty} \qquad \frac{\Delta; \Gamma \vdash \cdot \Longrightarrow \Gamma' \qquad x \notin \mathrm{cod}(\Phi)}{\Delta; \Gamma, \mathtt{x}{:}\mathtt{stack} \vdash \cdot \Longrightarrow \Gamma', x{:}\mathsf{stack}}\ \textsf{tsctx:cons1}$$

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

$$\frac{\Delta; \Gamma \vdash \Phi \Longrightarrow \Gamma' \qquad \mathtt{y}{:}\mathtt{t} \in \Gamma \qquad \mathtt{z}{:}\mathtt{string} \in \Gamma \qquad \Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau}{\Delta; \Gamma \vdash \Phi, x \mapsto (y, z) \Longrightarrow \Gamma', x{:}\tau \times \mathsf{string},}\ \textsf{tsctx:cons3}$$

Lemma D.7 (Type translation soundness). *If* $\Delta \Longrightarrow \Delta'$ *and*

- $\Delta \vdash \mathtt{s} \overset{\mathtt{type}}{\Longrightarrow} \tau$ *then* $\Delta' \vdash \tau$.
- $\Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$ *then* $\Delta' \vdash \tau$.

Proof. By trivial induction on the structure of $\Delta \vdash \mathtt{s} \overset{\mathtt{type}}{\Longrightarrow} \tau$ or $\Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$. □

Lemma D.8 (Type translation and substitution). *If* $\Delta \Longrightarrow \Delta'$ *and*

- $\Delta, \mathsf{a} \vdash \mathtt{s} \overset{\mathtt{type}}{\Longrightarrow} \tau$ *and* $\Delta \vdash \mathtt{t}' \overset{\mathtt{type}}{\Longrightarrow} \tau'$ *iff* $\Delta \vdash \mathtt{s}[\mathtt{t}'/\mathsf{a}] \overset{\mathtt{type}}{\Longrightarrow} \tau[\tau'/\alpha]$.
- $\Delta, \mathsf{a} \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$ *and* $\Delta \vdash \mathtt{t}' \overset{\mathtt{type}}{\Longrightarrow} \tau'$ *iff* $\Delta \vdash \mathtt{t}[\mathtt{t}'/\mathsf{a}] \overset{\mathtt{type}}{\Longrightarrow} \tau[\tau'/\alpha]$.

Proof. In the forward direction, by induction over the structure of $\Delta, \mathtt{a} \vdash \mathtt{s} \overset{\mathtt{type}}{\Longrightarrow} \tau$ and $\Delta, \mathtt{a} \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$ respectively. In the backward direction, by induction over $\Delta \vdash \mathtt{s}[\mathtt{t}'/\mathsf{a}] \overset{\mathtt{type}}{\Longrightarrow} \tau[\tau'/\alpha]$ and $\Delta \vdash \mathtt{t}[\mathtt{t}'/\mathsf{a}] \overset{\mathtt{type}}{\Longrightarrow} \tau[\tau'/\alpha]$. □

Lemma D.9 (Type injection is deterministic and total). *Given* Δ *and*

- $\mathrm{FTV}(\mathtt{s}) \subseteq \Delta$ *then there exists a unique derivation* $\Delta \vdash \mathtt{s} \overset{\mathtt{type}}{\Longrightarrow} \tau$.
- $\mathrm{FTV}(\mathtt{t}) \subseteq \Delta$ *then there exists a unique derivation* $\Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$.

Proof. By trivial induction on the structure of $\mathtt{s}$ or $\mathtt{t}$. □

Lemma D.10 (Linearity preserved).

- *If* Δ *linear and* $\Delta \Longrightarrow \Delta'$ *then* Δ' *linear.*
- *If* Γ *linear and* $\Delta \vdash \Gamma \Longrightarrow \Gamma'$ *then* Γ' *linear.*

Proof. Straightforward. □

Lemma D.11 (Generalization commutes with translation). *If* $\Delta \Longrightarrow \Delta'$ *and* $\Delta \vdash \bar{\mathsf{a}}.\mathsf{t}_1 \prec \bar{\mathsf{b}}.\mathsf{t}_2$ *then* $\Delta' \vdash \bar{\alpha}.\tau_1 \prec \bar{\beta}.\tau_2$ *where* $\Delta, \bar{\mathsf{a}} \vdash \mathtt{t}_1 \overset{\mathtt{type}}{\Longrightarrow} \tau_1$ *and* $\Delta, \bar{\mathsf{b}} \vdash \mathtt{t}_2 \overset{\mathtt{type}}{\Longrightarrow} \tau_2$.

Proof. Straightforward use of Lemmas D.9 and D.8. □

Lemma D.12 (Generalization equivalence). $\Delta \vdash \bar{\alpha}.\tau_1 \prec \bar{\beta}.\tau_2$ *iff* $\Delta \vdash \bar{\alpha}.(\tau_1 \times \tau_3) \prec \bar{\beta}.(\tau_2 \times \tau_3)$ *and* $\Delta \vdash \tau_3$

Proof. Straightforward. □

Lemma D.13 (Splitting lemma). *If* $\Delta_1 \vdash \Gamma_1 \Longrightarrow \Gamma_1'$ *and* $\Delta_1, \Delta_2 \vdash \Gamma_2 \Longrightarrow \Gamma_2'$ *and* $\Delta_1, \Delta_2; \Gamma_2 \vdash \Phi \Longrightarrow \Gamma_3'$. *and* $\Delta_1', \Delta_2'; \Gamma_1', \Gamma_2' \vdash e : \tau$ *then* $\Delta_1', \Delta_2'; \Gamma_1', \Gamma_3' \vdash \mathsf{split}(\Phi, e) : \tau$.

Proof. By induction on Φ. □

Lemma D.14 (Point cut designator translation safety). *If* $\Delta \vdash \Gamma \Longrightarrow \Gamma'$ *and*

- $\Delta; \Gamma \vdash \mathsf{pt} \overset{\mathtt{before}}{\Longrightarrow} e; \bar{\mathsf{a}}.\mathsf{t}$ *then* $\Delta'; \Gamma' \vdash e : (\bar{\alpha}.\tau \times \mathsf{stack} \times \mathsf{string})\ \mathsf{pc}$
- $\Delta; \Gamma \vdash \mathsf{pt} \overset{\mathtt{after}}{\Longrightarrow} e; \bar{\mathsf{a}}.\mathsf{t}$ *then* $\Delta'; \Gamma' \vdash e : (\bar{\alpha}.\tau \times \mathsf{stack} \times \mathsf{string})\ \mathsf{pc}$
- $\Delta; \Gamma \vdash \mathsf{pt} \overset{\mathtt{stk}}{\Longrightarrow} e; \bar{\mathsf{a}}.\mathsf{t}$ *then* $\Delta'; \Gamma' \vdash e : (\bar{\alpha}.\tau \times \mathsf{string})\ \mathsf{pc}$

Proof. By induction on the structure of $\Delta; \Gamma \vdash \mathsf{pt} \overset{\mathtt{time}}{\Longrightarrow} e; \bar{\mathsf{a}}.\mathtt{t}$. □

Lemma D.15 (Pattern translation safety). *If* $\Delta \vdash \Gamma \Longrightarrow \Gamma'$ *and* $\Delta; \Gamma \vdash \mathsf{p}[\bar{\mathsf{t}}/\bar{\mathsf{a}}] \overset{\mathtt{pat}}{\Longrightarrow} \rho \dashv \Delta''; \Gamma''; \Phi$ *then* $\Delta'; \Gamma' \vdash \rho \dashv \Delta^*; \Gamma^*$ *where* $\Delta'' \Longrightarrow \Delta^*$ *and* $\Delta, \Delta''; \Gamma'' \vdash \Phi \Longrightarrow \Gamma^*$ *and* $\rho = \rho'[\bar{\tau}/\bar{\alpha}]$ *where* $\Delta \vdash \mathsf{t}_i \overset{\mathtt{type}}{\Longrightarrow} \tau_i$.

Proof. By induction on the structure of $\Delta; \Gamma \vdash \mathsf{p}[\bar{\mathsf{t}}/\bar{\mathsf{a}}] \overset{\mathtt{pat}}{\Longrightarrow} \rho \dashv \Delta''; \Gamma''; \Phi$ with uses of Lemma D.14. □

Lemma D.16 (Term and declaration translation safety). *If* $\Delta \vdash \Gamma \Longrightarrow \Gamma'$ *and*

- $\Delta; \Gamma \vdash \mathtt{e}[\overline{\mathtt{t}}/\overline{\mathtt{a}}] : \mathtt{t} \overset{\mathtt{exp}}{\Longrightarrow} e$ *then* $\Delta'; \Gamma' \vdash e : \tau$ *where* $\Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$ *and* $e = e'[\overline{\tau}/\overline{\alpha}]$ *where* $\Delta \vdash \mathtt{t}_i \overset{\mathtt{type}}{\Longrightarrow} \tau_i$.
- $\Delta; \Gamma \vdash \mathtt{ds}[\overline{\mathtt{t}}/\overline{\mathtt{a}}]; \mathtt{e}[\overline{\mathtt{t}}/\overline{\mathtt{a}}] : \mathtt{t} \overset{\mathtt{decs}}{\Longrightarrow} e$ *then* $\Delta'; \Gamma' \vdash e : \tau$ *where* $\Delta \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$ *and* $e = e'[\overline{\tau}/\overline{\alpha}]$ *where* $\Delta \vdash \mathtt{t}_i \overset{\mathtt{type}}{\Longrightarrow} \tau_i$.

Proof. By mutal induction over the structure of $\Delta; \Gamma \vdash \mathtt{ds}[\overline{\mathtt{t}}/\overline{\mathtt{a}}]; \mathtt{e}[\overline{\mathtt{t}}/\overline{\mathtt{a}}] : \mathtt{t} \overset{\mathtt{decs}}{\Longrightarrow} e$ and $\Delta; \Gamma \vdash \mathtt{e}[\overline{\mathtt{t}}/\overline{\mathtt{a}}] : \mathtt{t} \overset{\mathtt{exp}}{\Longrightarrow} e$, with uses of Lemmas D.14, D.15, and D.13. □

Theorem D.17 (Program translation safety). *If* $\cdot; \cdot \vdash \mathtt{ds}\ \mathtt{e} : \mathtt{t} \overset{\mathtt{prog}}{\Longrightarrow} e$ *then* $\cdot; \cdot \vdash e : \tau$ *where* $\cdot \vdash \mathtt{t} \overset{\mathtt{type}}{\Longrightarrow} \tau$.

Proof. Straightforward use of Lemma D.16. □