μABC: A Minimal Aspect Calculus

Glenn Bruns[1], Radha Jagadeesan[2⋆], Alan Jeffrey[2⋆⋆], and James Riely[2⋆⋆⋆]

[1] Bell Labs, Lucent Technologies
[2] DePaul University

Abstract. Aspect-oriented programming is emerging as a powerful tool for system design and development. In this paper, we study aspects as primitive computational entities on par with objects, functions and horn-clauses. To this end, we introduce μABC, a name-based calculus, that incorporates aspects as primitive. In contrast to earlier work on aspects in the context of object-oriented and functional programming, the only computational entities in μABC are aspects. We establish a compositional translations into μABC from a functional language with aspects and higher-order functions. Further, we delineate the features required to support an aspect-oriented style by presenting a translation of μABC into an extended π-calculus.

1 Introduction

Aspects [7, 21, 28, 23, 22, 3] have emerged as a powerful tool in the design and development of systems (e.g., see [4]). To explain the interest in aspects, we begin with a short example inspired by tutorials of AspectJ [1]. Suppose class L realizes a useful library, and that we want to obtain timing information about a method `foo()` of L. With aspects this can be done by writing *advice* specifying that, whenever `foo` is called, the current time should be logged, `foo` should be executed, and then the current time should again be logged. It is indicative of the power of the aspect framework that:

- the profiling code is localized in the advice,
- the library source code is left untouched, and
- the responsibility for profiling all `foo()` calls resides with the compiler and/or runtime environment.

The second and third items ensure that, in developing the library, one need not worry about advice that may be written in the future. In [13] this notion is called *obliviousness*. However, in writing the logging advice, one must identify the pieces of code that need to be logged. In [13] this notion is called *quantification*. These ideas are quite general and are independent of programming language paradigm.

The execution of such an aspect-program can intuitively be seen in a reactive framework as follows. View method invocations (in this case the `foo()` invocations) as events. View advice code (in this case the logging advice) as running in parallel with the

⋆ Supported by NSF grant #0244901.
⋆⋆ Supported by NSF grant #0208549.
⋆⋆⋆ Supported by NSF grant #0347542.

other source code and responding to occurrences of events (corresponding to method calls). This view of execution is general enough to accommodate dynamic arrival of new advice by treating it as dynamically created parallel components. In the special case that all advice is static, the implicit parallel composition of advice can be compiled away — in aspect-based languages, this compile-time process called *weaving*. Informally, the weaving algorithm replaces each call to `foo()` with a call to the advice code, thus altering the client code and leaving the library untouched.

Aspect-oriented extensions have been developed for object-oriented [21, 28], imperative [20], and functional languages [30, 31]. Furthermore, a diverse collection of examples show the utility of aspects. These range from the treatment of inheritance anomalies in concurrent object-oriented programming (eg. see [25] for a survey of such problems, and [24] for an aspect-based approach) to the design of flexible mechanisms for access control in security applications [5]. Recent performance evaluations of aspect languages [12] suggest that a combination of programming and compiler efforts suffices to manage any performance penalties.

Much recent work on aspects is aimed at improving aspect-oriented language design and providing solutions to the challenge of reasoning about aspect-oriented programs. For example, there is work on adding aspects to existing language paradigms [30, 31], on finding a parametric way to describe a wide range of aspect languages [10], on finding abstraction principles [11], on type systems [18], and on checking the correctness of compiling techniques using operational models [19] or denotational models [32]. A strategy in much of this work is to develop an calculus that provides a manageable setting in which to study the issues. Similarly to the way that aspect languages have been designed by adding aspects to an existing programming paradigm, these calculi generally extend a base calculus with a notion of aspect. For example, [19] is based on an untyped class-based calculus, [10] is based on the object calculus [2], and [31] is based on the simply-typed lambda calculus.

If one wishes to study aspects in the context of existing programming languages, then calculi of this style are quite appropriate. However, another role for an aspect calculus is to identify the essential nature of aspects and understand their relationship to other basic computational primitives. We follow the approach of the theory of concurrency — concurrency is not built on top of sequentiality because that would certainly make concurrency more complex rather than sequentiality. Rather, concurrency theory studies interaction and concurrency as primitive concepts and sequentiality emerges as a special case of concurrency.

Along these lines, we aim here to establish aspects as primitive computational entities on par with objects, functions, and horn clauses; separate from their integration into existing programming paradigms. To this end we have created a minimal aspect calculus called $\mu\mathsf{ABC}$.

We present $\mu\mathsf{ABC}$ as a sequential deterministic calculus, with all concurrency being implicit. The primitive entities of $\mu\mathsf{ABC}$ are names, in the style of the pi-calculus [26] and the join calculus [15]. It differs in the choice of the communication paradigm in two ways: firstly, messages are broadcast (somewhat in the style of CSP [16]) to all receivers; secondly, the joins of the join-calculus are generalized to permit receiver code (ie. advice) to be conditional on second-order predicates over messages.

We show that functions and objects can be realized using μABC, demonstrating that aspects are an expressive primitive. Interestingly, μABC achieves expressiveness without explicit use of concurrency, providing an analysis that differs from those familiar to the concurrency community. This is not to say that aspects are incompatible with concurrency. The addition of explicit concurrency does not alter the basic development of μABC — we eschew explicit concurrency in μABC in this extended abstract to make the presentation manageable to a reader unfamiliar with aspects.

Organization. The rest of the paper is organized as follows. We begin with an informal introduction to the techniques and results of the paper. The key technical ideas are developed in the rest of the paper. Section 2 describes the syntax and dynamic semantics of μABC. The two following sections describe encodings of the lambda-calculus, both with and without aspects. Finally, we describe the translation of μABC into a variant of the polyadic pi-calculus. In this extended abstract, we elide all proofs.

2 Minimal aspect-based calculus

Aspect-oriented languages add *advice* and *pointcuts* on top of events occurring in an underlying model of computation. For example, in an imperative model, the events might be procedure calls or expression evaluations. The pointcut language provides a logic for describing events or event sequences. Here we restrict our attention to single events, leaving the furtile ground of temporal pointcuts (such as AspectJ's `cflow`) for future work.

Advice associates a pointcut with executable code. When an event specified by the pointcut occurs, the advice "fires", intercepting the underlying event. Execution of the event itself is replaced with execution of the advice body. Advice may optionally *proceed* to execute the underlying event at any point during execution of the advice body. If many pieces of advice fire on the same event, then *advice ordering* indicates which piece of advice will be executed; in this case, a proceed will cause execution of the next piece of advice.

In μABC, computational events are messages sent from a source to a target. The source, message, and target are specified as names, represented as lower-case letters. An example message command is the following:

$$\mathsf{let}\, x = p \to q : m$$

The four names in the command have different purposes, so we develop some distinguishing terminology. The source, p, and the target, q, are *roles*; m is a *message*; x is a *variable*, which binds the value returned by the message. Messages include both advice $a, \ldots, e$ and labels $f, \ldots, \ell$. Commands may specify a sequence of messages. This is useful for modeling both traditional method calls, $p \to q : \ell$, and advice sequences, $p \to q : a, b$. For compatibility with declaration order, we read message sequences right to left; in the command $p \to q : m, n$, message n is processed before m.

The only other computational commands in μABC are return statements, which terminate all command sequences; for example, "$\mathsf{return}\, v$" returns name v.

Finally, the calculus includes commands for declaring roles and advice. An advice declaration binds an advice name and specifies a pointcut and advice body. For example, the following advice a causes any message k sent from p to q to be redirected as a message ℓ, sent from p to r. This is an "extreme" form of delegation. Messages to q are delegated to r before q even receives them.

$$\mathsf{advice}\ a[p \to q : k] = \mathsf{let}\, x = p \to r : \ell; \mathsf{return}\, x$$

The term between brackets is a *pointcut* indicating that message k should be intercepted when sent from p to q. The body of the advice is given after the first equality symbol.

The pointcut language allows for quantification over names. For example, the following variation captures every k-message sent to q, regardless of the sender. The advice resends the message to q, renaming it to ℓ; the sending role is unchanged.

$$\mathsf{advice}\ a[\exists z\,.\, z \to q : k] = \sigma y\,.\ \mathsf{let}\, x = y \to q : \ell; \mathsf{return}\, x$$

Here, z binds the source of the message in the pointcut, and y binds the source of the message in the body of the advice. The binder σ is mnemonic for "source". One may also quantify over the target of a message; the corresponding binder is τ, for "target". The following code converts every k-message into a ℓ-message with the same source and target:

$$\mathsf{advice}\ a[\exists z_s\,.\, \exists z_t\,.\, z_s \to z_t : k] = \sigma y_s\,.\, \tau y_t\,.\ \mathsf{let}\, x = y_s \to y_t : \ell; \mathsf{return}\, x$$

In all the examples given so far, the advice causes all other code associated with the event to be ignored. If we wish to allow many pieces of advice to trigger on a single event, then we must encode the ability to "proceed". The proceed binder, π, captures the names of any other advice triggered by a pointcut. The following code captures k-messages and executes other advice *after* redirecting the message to to ℓ.

$$\mathsf{advice}\ a[\exists z_s\,.\, \exists z_t\,.\, z_s \to z_t : k] = \sigma y_s\,.\, \tau y_t\,.\, \pi b\,.\ \mathsf{let}\, x = y_s \to y_t : b, \ell; \mathsf{return}\, x$$

Reversing the order of b and ℓ, "$y_s \to y_t : \ell, b$", causes other advice to execute *before* redirecting the message. In this case, the ℓ-message will only be sent if the other advice uses its proceed binder. In general, b will be replaced with a sequence of messages when the advice fires. If there is no other associated advice, then b will be replaced with the empty sequence, in which case "$y_s \to y_t : \ell, b$" and "$y_s \to y_t : b, \ell$" execute identically.

μABC allows bounded quantification in pointcuts. As motivation, consider a class-based language with advice, such as AspectJ. In such a language, one may specify pointcuts based on class; all objects inhabiting the class will cause the pointcut to fire. Both objects and classes are encoded in μABC as roles. In this case, a pointcut must specify a subset of roles as the source or target of a message. We achieve this by associating names with a partial order. The following declaration captures any k-message sent to q from a sub-name of t.

$$\mathsf{advice}\ a[\exists z \le t\,.\, z \to q : k] = \sigma y\,.\ \mathsf{let}\, x = y \to q : \ell; \mathsf{return}\, x$$

The partial order is established through role declarations, such as the following, which declares p to be a sub-name of q.

$$\mathsf{role}\, p < q$$

In examples throughout the paper, the reserved name top is the largest name with respect to this ordering. We therefore abbreviate "$\mathsf{role}\,p<\mathsf{top}$" as "$\mathsf{role}\,p$."

The role hierarchy is used extensively in the encoding of the class-based language given in the full version of the paper [9].

Dynamics. Consider the following sequence of declarations, $\vec{D}$.

$$\begin{array}{l}\mathsf{role}\,p;\ \mathsf{role}\,q;\ \mathsf{role}\,r;\\ \mathsf{advice}\,a[p\rightarrow q:k] = \sigma y_s\,.\,\tau y_t\,.\,\pi b\,.\,(\mathsf{let}\,z = y_t \rightarrow r:b; \mathsf{return}\,y_s);\end{array}$$

Consider the execution of the following program using $\vec{D}$.

$$\vec{D};\ \mathsf{let}\,x = p\rightarrow q: j,k;\ \mathsf{return}\,x$$

Messages are processed using two rules which differentiate the type of the leading name in the message list, in this case k. To distinguish these two forms of reduction, we impose a syntactic distinction between advice and other names. Advice is named only to simplify the semantics. The syntactic distinction makes it so that advice cannot fire based on the execution of other advice. The *advice lookup* rule replaces the leading label (or role) in a message list with the advice names that the label triggers. So k is replaced with a.

$$\vec{D};\ \mathsf{let}\,x = p\rightarrow q: j,a;\ \mathsf{return}\,x$$

The *advice invocation* rule replaces a message command with the appropriately instantiated body of the triggered advice. Further reducing the program by this rule, we obtain:

$$\vec{D};\ \mathsf{let}\,z = p\rightarrow r: j;\ \mathsf{return}\,p$$

The return variable has changed as the result of a double substitution. In the process of inserting the advice body, occurrences of the let variable x are replaced with the return value of the advice body. In this case, the return value, y_s, is itself replaced with the name of the source of the message, p.

2.1 Syntax and semantics of μABC

Syntax For any grammar X, define the grammars of lists as:

$$\begin{array}{ll}\vec{X}\ ::= X_1,\ldots,X_n & \textit{Comma-separated lists}\\ \vec{X};\ ::= X_1;\ldots X_n; & \textit{Semicolon-terminated lists}\end{array}$$

Write ε for the empty list.

We assume a set of *names*, ranged over by m, n. We further assume that names are partitioned into two disjoint and distinguishable sets. Advice names are associated with pointcuts and advice. Roles are use to name objects in the system as well message labels.

$$\begin{array}{ll}f,\ldots,\ell,p,\ldots,z & \textit{Role (or Label)}\\ a,\ldots,e & \textit{Advice name}\end{array}$$

Names may be roles or advice names.

$$m,n ::= \ell \mid a \qquad \textit{Name (or Message)}$$

The grammar for μABC programs is as follows. We discuss point cuts, ϕ, below.

$$\begin{array}{ll} P,Q,R ::= & \textit{Program} \\ \quad \mathsf{return}\, v & \quad \text{Return} \\ \quad \mathsf{let}\, x = p \to q : \vec{m}; P & \quad \text{Message Send} \\ \quad \mathsf{role}\, p < q; P & \quad \text{Role} \\ \quad \mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q; P & \quad \text{Advice} \end{array}$$

Let D and E range over *declarations*, which may be either role delcarations "$\mathsf{role}\, p < q$" or advice declarations "$\mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q$". Let B and C range over *commands*, which may be declarations "D" or message sends "$\mathsf{let}\, x = p \to q : \vec{m}$". Note that all programs have the form $\vec{B}; \mathsf{return}\, v$.

- The command "$\mathsf{let}\, x = p \to q : \vec{m}; P$" binds x in P. Execution causes messages $\vec{m}$ to be sent from p to q, binding the return value of last executed message to x.
- The declaration "$\mathsf{role}\, p < q; P$" binds p in P. It declares p as a subrole of q.
- The declaration "$\mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q; P$" binds a in Q and P; in addition, x, y and z are bound in Q. It declares a to be an association between pointcut ϕ and advice body $\sigma x\,.\,\tau y\,.\,\pi b\,.\,Q$.

Omitted binders in an advice declaration are assumed to be fresh, for example:

$$\begin{array}{l} \mathsf{advice}\ [\phi] = Q; P \triangleq \mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q; P \\ \qquad\qquad\qquad\qquad \text{where } \{a,x,y,b\} \cap \mathsf{fn}(Q) = \emptyset \text{ and } a \notin \mathsf{fn}(P) \end{array}$$

Define *bound* and *free* names as usual. Write $\stackrel{\alpha}{=}$ for the equivalence generated by consistent renaming of bound names, $[{}^{v}\!/\!{}_{x}]$ for the capture-free substitution of name v for free name x, and $[{}^{\vec{m}}\!/\!{}_{a}]$ for the capture-free substitution of the name list $\vec{m}$ for free name a. Denote simultaneous substitution as $[{}^{v}\!/\!{}_{x}, {}^{w}\!/\!{}_{y}]$.

Pointcuts The grammar for pointcuts is as follows.

$$\begin{array}{lll} \phi,\psi ::= & & \textit{Pointcut} \\ \quad \mathsf{true} & \mid \mathsf{false} & \quad \text{True, False} \\ \quad \phi \wedge \psi & \mid \phi \vee \psi & \quad \text{And, Or} \\ \quad \forall x \leq p\,.\,\phi & \mid \exists x \leq p\,.\,\phi & \quad \text{All, Some} \\ \quad p \to q : \ell & \mid \neg p \to q : \ell & \quad \text{Atom, Not Atom} \end{array}$$

The satisfaction relation, "$\vec{D}; \vdash p \to q : \ell \ \mathsf{sat}\ \phi$", states that message $p \to q : \ell$ satisfies ϕ assuming the role hierarchy given by $\vec{D}$. Satisfaction is defined in the standard way, building up from the atoms. We say that pointcuts ϕ and ψ *overlap* in $\vec{D}$; if for some p, q and ℓ, $\vec{D}; \vdash p \to q : \ell \ \mathsf{sat}\ \phi$ and $\vec{D}; \vdash p \to q : \ell \ \mathsf{sat}\ \psi$.

We write "$p\,.\,\ell$" for the pointcut which fires when p or one of its subroles receives message ℓ:

$$p\,.\,\ell \triangleq \exists x \leq \mathsf{top}\,.\,\exists y \leq p\,.\,x \to y : \ell$$

Dynamic semantics The reduction relation, $P \twoheadrightarrow P'$, is defined by two rules. The first defines *advice lookup*. The second defines *advice invocation*. Advice lookup replaces the message $p \to q : \ell$ with $p \to q : \vec{a}$, where $\vec{a}$ is the advice associated with $p \to q : \ell$. The order in the sequence of advice is the same as the declaration order. The rule treats the rightmost message in a sequence.

$$\begin{array}{c}\vec{D}; \mathsf{let}\, z = p \to q : \vec{m}, \ell; P \;\;\twoheadrightarrow\;\; \vec{D}; \mathsf{let}\, z = p \to q : \vec{m}, \vec{a}; P \\ \text{where } [\vec{a}] = \left[a \,\middle|\, \begin{array}{l}(\mathsf{advice}\, a[\phi] \cdots) \in \vec{D} \\ \text{and } \vec{D}; \vdash p \to q : \ell \;\mathsf{sat}\; \phi\end{array} \right]\end{array}$$

Advice invocation replaces the message $p \to q : a$ with the body of a. This requires a few substitutions to work. Suppose the body of a is "$\sigma x\,.\,\tau y\,.\,\pi b\,.\,Q$", where Q is "$\vec{B}; \mathsf{return}\, v$". Suppose further that we wish to execute "$\mathsf{let}\, z = p \to q : \vec{m}, a; P$". The source of the message is p, the target is q, the body to execute is $\vec{B}$ returning v, and the subsequent messages are $\vec{m}$. This leads us to execute $\vec{B}[p/x, q/y, \vec{m}/b]$ then $P[v/z]$. The substitution in P accounts for the returned value of Q. As a final detail, we must take care of collisions between the bound names of Q and P. We define the notation "$\mathsf{let}\, z = Q; P$" to abstract the details of the required renaming.

$$\begin{array}{l}\mathsf{let}\, z = Q; P \triangleq \vec{B}; P[v/z] \\ \qquad \text{where } \mathsf{bn}(\vec{B}) \cap \mathsf{fn}(P) = \emptyset \text{ and } Q \stackrel{\alpha}{=} \vec{B}; \mathsf{return}\, v\end{array}$$

With this notation, the rule can be written as follows.

$$\begin{array}{c}\vec{D}; \mathsf{let}\, z = p \to q : \vec{m}, a; P \;\;\twoheadrightarrow\;\; \vec{D}; \mathsf{let}\, z = Q[p/x, q/y, \vec{m}/b]; P \\ \text{where } (\mathsf{advice}\, a[\cdots] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q) \in \vec{D}\end{array}$$

Note that in the reduction semantics, the ordering of advice is significant only for overlapping pointcuts.

Garbage collection In the following sections, we present encodings that leave behind useless declarations as the terms reduce. In order to state correctness of the translations, we must provide a way to remove unused declarations from a term. For example, the following rule allows for collection of unused roles:

$$\vec{D}; \mathsf{role}\, p < q; P \stackrel{\mathsf{gc}}{\twoheadrightarrow} \vec{D}; P \qquad \text{where } p \notin \mathsf{fn}(P)$$

An adequate set of garbage collection rules is given in the full version of the paper [9].

3 Translation of other AOP languages into μABC

The small collection of basic orthogonal primitives of μABC make it a viable candidate to serve a role analogous to that of object-calculi in the study of object-oriented programming, provided that it is expressive enough. We establish the expressive power of μABC by *compositional* translations from the following languages that add aspects added on top of distinct underlying programming paradigms:

- A lambda-calculus with aspects — core minAML [31].
- An imperative class-based language (in the spirit of Featherweight Java [17], Middleweight Java [8], and Classic Java [14]) enhanced with aspects [19].

On one hand, the translations support our hypothesis that $\mu\mathsf{ABC}$ captures a significant portion of the world of aspects. On the other hand, they establish that aspects, in isolation, are indeed a full-fledged computational engine.

In this extended abstract, we discuss only minAML; the encoding of the class-based language is given in the full version [9]. We start with a discussion of functions and conditionals.

3.1 Functions and conditionals

The encodings in this section rely heavily on the following notation. In a context expecting a program, define "x" as the program which returns x, and define "$p \to q : \vec{m}$" as the program which returns the result of the message:

$$\begin{aligned} x &\triangleq \mathsf{return}\, x \\ p \to q : \vec{m} &\triangleq \mathsf{let}\, x = p \to q : \vec{m}; \mathsf{return}\, x \end{aligned}$$

Given this shorthand, we encode abstraction and application as follows, where f and g are fresh roles and "call" and "arg" are reserved roles that are not used elsewhere. The basic idea is to model an abstraction as a piece of advice that responds to "call" — in response to this method, the advice body invokes the argument by emitting "arg" to initiate evaluation of the argument. An application is encoded in a manner consistent with this protocol: in an application, the argument is bound to advice that triggers on "arg".

$$\begin{aligned} \lambda x \,.\, P &\triangleq \mathsf{role}\, f; \\ &\ \mathsf{advice}\ [f \,.\, \mathsf{call}] = \tau y \,.\, \mathsf{let}\, x = y \to y : \mathsf{arg}; P; \\ &\ \mathsf{return}\, f \\ R\, Q &\triangleq \mathsf{let}\, f = R; \\ &\ \mathsf{role}\, g < f; \\ &\ \mathsf{advice}\ [g \,.\, \mathsf{arg}] = Q; \\ &\ g \to g : \mathsf{call} \end{aligned}$$

Example 1. The encoding of $(\lambda x \,.\, P)\, Q$ is "$\vec{D}; g \to g : \mathsf{call}$", where $\vec{D};$ is as follows:

$$\begin{aligned} \vec{D}; = {} & \mathsf{role}\, f; \\ & \mathsf{advice}\ a[f \,.\, \mathsf{call}] = \tau y \,.\, \mathsf{let}\, x = y \to y : \mathsf{arg}; P; \\ & \mathsf{role}\, g < f; \\ & \mathsf{advice}\ b[g \,.\, \mathsf{arg}] = Q; \end{aligned}$$

This term reduces as:

$$\begin{aligned} (\lambda x \,.\, P)\, Q &= \vec{D}; g \to g : \mathsf{call} \\ &\rightarrow \vec{D}; g \to g : a \\ &\rightarrow \vec{D}; \mathsf{let}\, x = g \to g : \mathsf{arg}; P \\ &\rightarrow \vec{D}; \mathsf{let}\, x = g \to g : b; P \\ &\rightarrow \vec{D}; \mathsf{let}\, x = Q; P \\ &\xrightarrow{\mathsf{gc}} \mathsf{let}\, x = Q; P \end{aligned}$$

From here, Q is reduced to a value v, then computation proceeds to $P[{}^{v}\!/\!_{x}]$.

This is the expected semantics of call-by-value application, except for the presence of the declarations $\vec{D}$, which we garbage collect.

Example 2. We now give a direct encoding of the conditional. The encoding shows one use of advice ordering. Define "if $p \leq q$ then P else Q" as the following program, where r is a fresh role and "if" is a reserved role.

$$\begin{aligned}\mathsf{if}\ p \leq q\ \mathsf{then}\ P\ \mathsf{else}\ Q \triangleq\ & \mathsf{role}\, r;\\ & \mathsf{advice}\ [\exists x \leq \mathsf{top}\,.\, x \to r : \mathsf{if}] = Q;\\ & \mathsf{advice}\ [\exists x \leq q\,.\, x \to r : \mathsf{if}] = P;\\ & p \to r : \mathsf{if}\end{aligned}$$

Note that P makes no use of its proceed variable, and so if P fires it effectively blocks Q. We can verify the following.

$$\vec{D}; \mathsf{if}\ p \leq q\ \mathsf{then}\ P\ \mathsf{else}\ Q \rightarrow^{*} \xrightarrow{\mathsf{gc}} \begin{cases} P & \text{if } \vec{D}; \vdash p \leq q \\ Q & \text{otherwise} \end{cases}$$

3.2 Encoding core MinAML in μABC

We sketch an encoding into μABC of the function-based aspect language MinAML defined by Walker, Zdancewic and Ligatti [31]. We treat a subset of core MinAML which retains the essential features of the language. Our goal, in this extended abstract, is not to provide a complete translation, but rather to show that the essential features of [31] are easily coded in μABC. In particular, in [31], advice is considered to be a first-class citizen, where here we treat it as second-class.

Core MinAML extends the lambda calculus with:

- The expression new $p;P$ creates a new name r which acts as a hook.
- The expression $\{p\,.\,z \to Q\} \gg P$ attaches the advice $\lambda z\,.\,Q$ to the hook p. The new advice is executed after any advice that was previously attached to p.
- The expression $\{p\,.\,z \to Q\} \ll P$ is similar, except that the new advice is executed *before* any previously attached advice.
- The expression $p\langle P\rangle$ evaluates P and then runs the advice hooked on p.

The encoding into μABC directly follows these intuitions and is as follows, where p is a fresh role and "hook" is a reserved role. The subtle difference between the encoding of before and after previous advice is a paradigmatic use of the proceed binder in μABC.

$$\begin{aligned}\mathsf{new}\ p;P &\triangleq \mathsf{role}\, p; \mathsf{advice}\ [p\,.\,\mathsf{hook}] = \lambda x\,.\,x; P\\ \{p\,.\,x \to Q\} \ll P &\triangleq \mathsf{advice}\ [p\,.\,\mathsf{hook}] = \tau z\,.\,\pi b\,.\,(\lambda x\,.\,\mathsf{let}\, y = Q; (z \to z : b)(y)); P\\ \{p\,.\,x \to Q\} \gg P &\triangleq \mathsf{advice}\ [p\,.\,\mathsf{hook}] = \tau z\,.\,\pi b\,.\,(\lambda y\,.\,\mathsf{let}\, x = (z \to z : b)(y); Q); P\\ p\langle P\rangle &\triangleq (p \to p : \mathsf{hook})\, P\end{aligned}$$

Example 3. Walker, Zdancewic and Ligatti present the following example. We show the reductions under our encoding. For the purpose of this example, we extend μABC with integers and expressions in the obvious way.

$$\mathsf{new}\ p; \{p\,.\,x_1 \to x_1 + 1\} \ll \{p\,.\,x_2 \to x_2 * 2\} \gg p\langle 3\rangle$$

This translates to "$\vec{D}; (p \to p : \mathsf{hook})\,3$", where $\vec{D}$; is:

$$\begin{array}{l} \mathsf{role}\, p; \\ \mathsf{advice}\ a[p\,.\,\mathsf{hook}] = \lambda x_0\,.\,x_0; \\ \mathsf{advice}\ b[p\,.\,\mathsf{hook}] = \tau z\,.\,\pi b\,.\,\lambda x_1\,.\,\mathsf{let}\, y_1 = x_1 + 1; (z \to z : b)(y_1); \\ \mathsf{advice}\ c[p\,.\,\mathsf{hook}] = \tau z\,.\,\pi b\,.\,\lambda y_2\,.\,\mathsf{let}\, x_2 = (z \to z : b)(y_2); x_2 * 2; \end{array}$$

and reduction proceeds as follows.

$$\begin{array}{rl} & \vec{D}; (p \to p : \mathsf{hook})\,3 \\ & \rightarrowtriangle \vec{D}; (p \to p : a, b, c)\,3 \\ & \rightarrowtriangle \vec{D}; (\lambda y_2\,.\,\mathsf{let}\, x_2 = (p \to p : a, b)(y_2); x_2 * 2)\,3 \\ \rightarrowtriangle^* & \xrightarrow{\mathsf{gc}} \vec{D}; \mathsf{let}\, x_2 = (p \to p : a, b)(3); x_2 * 2 \\ & \rightarrowtriangle \vec{D}; \mathsf{let}\, x_2 = (\lambda x_1\,.\,\mathsf{let}\, y_1 = x_1 + 1; (p \to p : a)(y_1))(3); x_2 * 2 \\ \rightarrowtriangle^* & \xrightarrow{\mathsf{gc}} \vec{D}; \mathsf{let}\, x_2 = (\mathsf{let}\, y_1 = 3 + 1; (p \to p : a)(y_1)); x_2 * 2 \\ \rightarrowtriangle^* & \xrightarrow{\mathsf{gc}} \vec{D}; \mathsf{let}\, x_2 = (p \to p : a)(4); x_2 * 2 \\ & \rightarrowtriangle \vec{D}; \mathsf{let}\, x_2 = (\lambda x_0\,.\,x_0)(4); x_2 * 2 \\ \rightarrowtriangle^* & \xrightarrow{\mathsf{gc}} \vec{D}; \mathsf{let}\, x_2 = 4; x_2 * 2 \\ & \rightarrowtriangle \vec{D}; 8 \\ & \xrightarrow{\mathsf{gc}} 8 \end{array}$$

4 Polyadic π-calculus with pointcuts

We identify the features required to support an aspect-oriented style by presenting a translation of μABC into a variant of the polyadic π-calculus [26]. The motivation for this portion of the paper is the striking analogies between aspects and concurrency that go beyond our use of concurrency techniques (eg. names, barbed congruences) to study aspects.

Firstly, there are the superficial similarities. Both aspects and concurrency assign equal status to callers/senders and callees/receivers. Both cause traditional atomicity notions to break with the concomitant effects on reasoning — aspects do this by refining atomic method calls into potentially multiple method calls, and concurrency does this by the interleaving of code from parallel processes.

Secondly, there are deeper structural similarities. The idea of using parallel composition to modify existing programs without altering them is a well-understood modularity principle in concurrent programming. Such an analysis is an essential component of the design of synchronous programming languages [6]. Construed this way, the classical parallel composition combinator of concurrency theory suffices for the "obliviousness" criterion on aspect-oriented languages [13] — the behavior of a piece of program text must be amenable to being transformed by advice, without altering the program text.

If this informal reasoning is correct, one might expect that the only new feature that an expressive concurrent paradigm needs to encode μABC is a touch of "quantification", which has been identified as the other key ingredient of the aspect style [13].

Our translation into a polyadic π-calculus is an attempt to formalize this reasoning. Consider a variant of the π-calculus with a hierarchy on names, so the new name process now has the form $\mathsf{new}\ x{<}y; P$. We also consider a slight generalization of the match combinator present in early versions of the π-calculus [27], which permits matching on the hierarchy structure on names. The form of the match process is $[\vec{x}\ \mathsf{sat}\ \phi]\, P$ where ϕ is a formula in a pointcut language that is essentially a boolean algebra built from atoms of the form $\vec{x}$. The generalized match construct can express traditional (mis)matching via $[x = y]P = [x\ \mathsf{sat}\ y]\, P$.

The dynamics of π is unchanged, apart from an extra rule to handle the generalized matching construct that checks the hierarchy of names (written here as $\vec{D}$;) for facts relating to the names (here $\vec{z}$).

$$\vec{D} \vdash [\vec{z}\ \mathsf{sat}\ \phi]\, P \rightarrowtail P \quad \text{where } \vec{D} \vdash \vec{z}\ \mathsf{sat}\ \phi$$

We describe a compositional translation from μABC to the polyadic π-calculus with these mild extensions.

4.1 Syntax and semantics of π with pointcuts

Syntax The grammar for pointcuts is as for μABC, except for the atoms.

$$\begin{array}{lll} \phi, \psi & ::= & \ldots\ \textit{Pointcut (As for } \mu\mathsf{ABC)} \\ \quad \vec{x} & & \text{Atom} \\ \quad \neg\vec{x} & & \text{Not Atom} \end{array}$$

The grammar of processes is standard, except for a generalized match construct.

$$\begin{array}{lll} P, Q, R & ::= & \textit{Process} \\ \quad z\langle\vec{x}\rangle \mid z(\vec{x})\, P & & \text{Output, Input} \\ \quad \mathbf{0} \quad \mid P \mid Q & & \text{Termination, Parallel} \\ \quad !P & & \text{Replication} \\ \quad \mathsf{new}\ x{<}y; P & & \text{New Name} \\ \quad [\vec{x}\ \mathsf{sat}\ \phi]\, P & & \text{Match} \end{array}$$

The matching construct allows for both matching and mismatching. We can define "$[x = y]P$" as "$[x\ \mathsf{sat}\ y]\, P$" and "$[x \neq y]P$" as "$[x\ \mathsf{sat}\ \neg y]\, P$".

Dynamic semantics Let X range over partially ordered finite sets of names, and write $X \vdash x \leq y$ when $x \leq y$ can be derived from X. The semantics of pointcuts $X \vdash \vec{x}\ \mathsf{sat}\ \phi$ is as for μABC except for the atoms:

$$X \vdash z_1, \ldots, z_n\ \mathsf{sat}\ z_1, \ldots, z_n$$
$$X \vdash z_1, \ldots, z_n\ \mathsf{sat}\ \neg x_1, \ldots, x_m \text{ if } n \neq m \text{ or } z_i \neq x_i \text{ for some } i$$

The dynamic semantics $X \vdash P \rightarrowtail P'$ is given by the usual π-calculus rules, the only difference being that the semantics of pointcuts requires the partial order X in the reduction:

$$\frac{X \vdash \vec{z}\ \mathsf{sat}\ \phi}{X \vdash [\vec{z}\ \mathsf{sat}\ \phi]\, P \rightarrowtail P}$$

and so the structural rule for new must include the partial order:

$$\frac{X, x < y \vdash P \rightarrowtail P' \quad x \notin X}{X \vdash \mathsf{new}\ x{<}y; P \rightarrowtail \mathsf{new}\ x{<}y; P'}$$

The remainder of the dynamic semantics is as given in [26].

4.2 Encoding μABC in π

We now show that μABC can be translated (via a spaghetti-coded CPS transform [29]) into our polyadic π-calculus.

In our translation, following the intuitions expressed in the introduction, advice is simply placed in parallel with the advised code. However, we need to account for a couple of features that disallow the straightforward use of parallel composition and cause the superficial complexity of the translation. First, we need to do some programming to make a single message of interest activate potentially several pieces of advice. Second, the order of invocation of the advice is fixed, so we are forced to program up explicitly the order in which the message is passed down the advice chain.

We will, in fact, translate a sublanguage, but one which contains all programs we consider interesting. A program $P = \vec{D}; Q$ is *user code* whenever, for any call $p \to q : \vec{m}$ contained in P, we have:

- $\vec{m}$ is a role ℓ; or
- $\vec{m}$ is an advice name b bound as a proceed variable — that is, there is an enclosing advice declaration $\mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\cdots$.

Unfortunately, user code is not closed under reduction: if P is user code, and $P \twoheadrightarrow P'$, then P' is not necessarily user code. We defined user closed code, which is closed under reduction.

Definition 4. *A program $P = \vec{D}; Q$ is* user closed *whenever, for any call $p \to q : \vec{m}$ contained in P, we have either:*

- *$\vec{m}$ is a role ℓ; or*
- *$\vec{m}$ is an advice name b bound as a proceed variable; or*
- *$\vec{m}$ is a sequence "$\vec{a}$" and for some $\vec{b}$, r, s and ℓ:*

$$[\vec{a}, \vec{b}] = \left[c \,\middle|\, \begin{array}{l} \mathsf{advice}\, c[\phi] \cdots \in \vec{D} \\ \vec{D}; \vdash r \to s : \ell\ \mathsf{sat}\ \phi \end{array} \right]$$

Table 1 Translation from μABC to π

$$
\begin{aligned}
\vec{D};\vdash [\![D;P]\!](k,c,\rho) &= \mathsf{new}\ a < c; (Q \mid !a(r,s,\ell,x,y,k',c')\,(\\
&\qquad [r,s,\ell\ \mathsf{sat}\ [\![\phi]\!]]\,Q' \\
&\qquad \mid [r,s,\ell\ \mathsf{sat}\ \neg[\![\phi]\!]]\,c\langle r,s,\ell,x,y,k',c'\rangle \\
&\quad)) \\
&\quad \text{where } D = \mathsf{advice}\ a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,P' \\
&\quad \text{and } \vec{D};D;\vdash [\![P]\!](k,a,\rho) = Q \\
&\quad \text{and } \vec{D};D;\vdash [\![P']\!](k',c',\rho \cup \{b \mapsto (c,r,s,\ell)\}) = Q' \\
\vec{D};\vdash [\![D;P]\!](k,c,\rho) &= \mathsf{new}\ p < q; Q \\
&\quad \text{where } D = \mathsf{role}\ p < q \text{ and } \vec{D};D;\vdash [\![P]\!](k,c,\rho) = Q \\
\vec{D};\vdash [\![\mathsf{let}\,x = p \to q : \varepsilon; P]\!](k,c,\rho) &= \mathsf{new}\ k' < k; (\mathsf{error}\langle p,q,\ell,p,q,k',c\rangle \mid k'(x,c')\,Q) \\
\vec{D};\vdash [\![\mathsf{let}\,x = p \to q : \ell; P]\!](k,c,\rho) &= \mathsf{new}\ k' < k; (c\langle p,q,\ell,p,q,k',c\rangle \mid k'(x,c')\,Q) \\
&\quad \text{where } \vec{D};\vdash [\![P]\!](k,c',\rho) = Q \\
\vec{D};\vdash [\![\mathsf{let}\,x = p \to q : b; P]\!](k,c,\rho) &= \mathsf{new}\ k' < k; (d\langle r,s,\ell,p,q,k',c\rangle \mid k'(x,c')\,Q) \\
&\quad \text{where } \rho(b) = (d,r,s,\ell) \text{ and } \vec{D};\vdash [\![P]\!](k,c',\rho) = Q \\
\vec{D};\vdash [\![\mathsf{let}\,x = p \to q : \vec{a},b; P]\!](k,c,\rho) &= \mathsf{new}\ k' < k; (b\langle r,s,\ell,p,q,k',c\rangle \mid k'(x,c')\,Q) \\
&\quad \text{where } [\vec{a},b] = \left\lceil a \,\middle|\, \begin{array}{l} \mathsf{advice}\,a[\phi]\cdots \in \vec{D} \\ \vec{D};\vdash a \leq b \\ \vec{D};\vdash r \to s : \ell\ \mathsf{sat}\ \phi \end{array} \right\rfloor \\
&\quad \text{and } \vec{D};\vdash [\![P]\!](k,c',\rho) = Q \\
\vec{D};\vdash [\![\mathsf{return}\,v]\!](k,c,\rho) &= k\langle v,c\rangle
\end{aligned}
$$

Let ρ be a partial function from names to quadruples of names. We define the translation $\vec{D};\vdash [\![P]\!](k,c,\rho) = Q$ in Table 1. Write "$\vec{D};\vdash [\![P]\!] = Q$" as shorthand for "$\vec{D};\vdash [\![P]\!](\mathsf{result},\mathsf{error},\emptyset) = Q$".

The translation uses communication of seven-tuples $\langle r,s,\ell,x,y,k,c\rangle$. Here r is the original caller, s is the original callee, ℓ is the original method name, x is the current caller of a piece of advice, y is the current callee, k is a continuation c is the name of the most recently declared advice. Whenever a method is called, the translation goes through the list c, checking advice in order. This encodes advice lookup.

Note that the translation is partial: there exist programs P such that there is no Q for which $[\![P]\!] = Q$. However, on user closed programs, the translation is total: there always exists such a Q. Moreover, on user code, the translation is a function: Q is uniquely determined by P.

Theorem 5. *For any user code P, if* $[\![P]\!] = Q$ *then* $P \twoheadrightarrow^* \vec{D};\mathsf{return}\,v$ *iff* $\vdash Q \rightarrowtail^* \mathsf{new}\ \vec{x} < \vec{y}; (Q' \mid \mathsf{result}\langle v\rangle)$.

5 Conclusions and Future work.

μABC was deliberately designed to be a small calculus that embodies the essential features of aspects. However, this criterion makes μABC an inconvenient candidate to serve in the role of a meta-language that is the target of translations from "full-scale" aspect languages. There is recent work on such meta-languages (eg. [10] builds on top of the full object calculus), and the bridging of the gap between μABC and such work remains open for future study. In this vein, we are exploring the addition of temporal connectives to the pointcut logic of μABC. Such an approach provides a principled way to understand and generalize features in existing aspect languages, e.g. cflow in AspectJ, that quantify over sequences of events.

There is ample evidence that aspect-oriented programming is emerging as a powerful tool for system design and development. From the viewpoint of CONCUR, aspects provide two intriguing opportunities. First, the techniques and approaches that have been explored in concurrency theory provide the basis for a systematic foundational analysis of aspects. Our description of μABC and its expressiveness falls into this category. In a more speculative vein, the large suite of tools and techniques studied in concurrency theory are potentially relevant to manage the complexity of reasoning required by aspect-oriented programming. Our translation of μABC into the pi-calculus is a step in understanding this connection.

References

1. AspectJ website. `http://www.eclipse.org/aspectj/`.
2. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer Verlag, 1996.
3. M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting object-interactions using composition-filters. In *In object-based distributed processing, LNCS*, 1993.
4. Association of Computing Machinery. *Communications of the ACM*, Oct 2001.
5. Lujo Bauer, Jarred Ligatti, and David Walker. A calculus for composing security policies. Technical Report TR-655-02, Dept. of Computer Science, Princeton University, 2002.
6. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.
7. L. Bergmans. *"Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs"*. Ph.d. thesis, University of Twente, 1994. `http://wwwhome.cs.utwente.nl/~bergmans/phd.htm`.
8. G.M. Bierman, M.J. Parkinson, and A.M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.
9. G. Bruns, R. Jagadeesan, A. Jeffrey, and J. Riely. μABC: A minimal aspect calculus. Full version, available at `http://fpl.cs.depaul.edu/ajeffrey/papers/muABCfull.pdf`, 2004.
10. Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Submitted for publication, at `ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf`, oct 2003.
11. Daniel S. Dantas and David Walker. Aspects, information hiding and modularity. Submitted for publication, at `http://www.cs.princeton.edu/~dpw/papers/aspectml-nov03.pdf`, 2003.
12. Bruno Dufour, Christopher Goard, Laurie Hendren, Clark Verbrugge, Oege de Moor, and Ganesh Sittampalam. Measuring the dynamic behaviour of AspectJ programs, 2003.
13. R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.

14. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.
15. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, 1996. Springer-Verlag. LNCS 1119.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Int. Series in Computer Science. Prentice Hall, 1985.
17. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
18. R. Jagadeesan, A. Jeffrey, and J. Riely. A typed calculus for aspect-oriented programs. Submitted for publication, at `http://fpl.cs.depaul.edu/ajeffrey/papers/typedABL.pdf`, 2003.
19. Radha Jagadeesan, Alan Jeffrey, and James Riely. An untyped calculus of aspect oriented programs. In *Conference Record of ECOOP 03: The European Conference on Object-Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, 2003.
20. Gregor Kiczales and Yvonne Coady. `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`.
21. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
22. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.
23. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
24. C. V. Lopes. *"D: A Language Framework for Distributed Programming"*. Ph.d. thesis, Northestern University, 1997. `ftp://ftp.ccs.neu.edu/pub/people/lieber/theses/lopes/dissertation.pdf`.
25. Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
26. R. Milner. The polyadic pi-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
27. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–40, 1992.
28. H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.
29. Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.
30. David Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Conference Record of AOSD 03: The 2nd International Conference on Aspect Oriented Software Development*, 2003.
31. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Conference Record of ICFP 03: The ACM SIGPLAN International Conference on Functional Programming*, 2003.
32. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 2003. To appear.