μABC: A Minimal Aspect Calculus

Glenn Bruns, Bell Labs Lucent
Radha Jagadeesan, DePaul University
Alan Jeffrey, DePaul University
James Riely, 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 minimal 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 compositional translations into* μABC *from a class-based language with aspects and classes, and 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 pi-calculus.*

1 Introduction

Aspects [6, 14, 21, 16, 15, 2] have emerged as a powerful tool in the design and development of systems (e.g., see [3]). To explain the interest in aspects, we begin with a short example inspired by tutorials of AspectJ (`http://www.aspectj.org`). 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 [10] this notion is called *obliviousness*. However, in writing the logging advice, one must identify the pieces of code that need to be logged. In [10] this notion is called *quantification*. These ideas are quite general and are independent of programming language paradigm.

Aspect-oriented extensions have been developed for object-oriented [14, 21], imperative [13], and functional languages [22, 23]. 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 [18] for a survey of such problems, and [17] for an aspect-based approach) to the design of flexible mechanisms for access control in security applications [4]. Recent performance evaluations of aspect languages [9] 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 [22, 23], on finding a parametric way to describe a wide range of aspect languages [7], on finding abstraction principles [8], on type systems [11], and on checking the correctness of compiling techniques using operational models [12] or denotational models [24]. 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 calculi with a notion of aspect. For example, [12] is based on an untyped class-based calculus, [7] is based on the object calculus [1], and [23] 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 calculi 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 μABC. We show using μABC that functions and methods can be realized using aspects.

In the rest of this introduction, we outline the key technical ideas of the paper.

μABC. Our calculus, μABC, incorporates aspects as a primitive. The only entities in μABC apart from the two elements characteristic of aspect-oriented programming — advice and pointcuts — is an acyclic hierarchy on names.

The complete grammar for programs is given by:

$P, Q, R ::=$	(Program)
$\mathsf{let}\, x = p \to q : \vec{m}; P$	(Message)
$\mathsf{return}\, v$	(Return)
$\mathsf{role}\, p < q; P$	(New Role)
$\mathsf{advice}\ a[\phi] = \sigma x . \tau y . \pi b . Q; P$	(New Advice)

The declaration of a new name $\mathsf{role}\, p < q$ introduces the name p and places it below name q in the name ordering. We call names thus introduced *roles*. Roles do not carry any code, but rather are hooks upon which to place advice. The only command other than returns are messages. The message $p \to q : \vec{m}$ has three components: a sender name p, a receiver name q and the body $\vec{m}$ which is merely a sequence of names. This message syntax makes explicit the symmetry between sender and receiver in aspect-oriented programming, in contrast to the asymmetry in traditional programming (e.g., an object cannot usually restrict the type of the caller of its methods).

Advice names are disjoint from roles – advice names are unordered and have associated code. The advice declaration $\mathsf{advice}\ a[\phi] = \sigma x . \tau y . \pi b . Q$ introduces advice name a with body Q parameterized over the message source x, target y and the following piece of advice b, usually called "proceed" in aspect-oriented languages. The advice a is invoked when the condition described by the pointcut ϕ occurs. The pointcut ϕ is essentially an element of the boolean algebra generated by atoms $p \to q : \ell$, and bounded quantification over roles such as $\exists x \leq p . \phi$, to provide ways to talk about classes of names in the name hierarchy. Formally, the satisfaction relation has the form

$$\vec{D}; \vdash p \to q : \ell\ \mathsf{sat}\ \phi$$

and describes when an atom satisfies a pointcut in the context of the name order in declarations $\vec{D}$. This relation depends on both the sender and receiver names, thus maintaining the symmetry between between senders and receivers. Thus, the atom $p \to q : \ell$ is satisfied, in any declaration context, only when the sender, receiver and message names agree with p, q, ℓ respectively.

The dynamic semantics has only two rules. Executing $p \to q : \ell$ involves the lookup of the advice list $\vec{a}$ that holds for it in the declarations $\vec{D}$. The order of advice in $\vec{a}$ is determined by the order in the declarations $\vec{D}$.

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

An invocation of a piece of advice a is resolved by looking up its body in the declarations $\vec{D}$. The parameter substitution is performed as follows: the substitution ${}^{p}\!/\!_{x}$ handles the sender, the substitution ${}^{q}\!/\!_{y}$ handles the receiver and the substitution ${}^{\vec{m}}\!/\!_{b}$ ensures that the rest of the advice list is bound to the "proceed" variable b.

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

Expressiveness of 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 languages with aspects added on top of distinct underlying programming paradigms.

- We describe the mapping of a class-based language with aspects and classes into μABC.

- We describe a translation of the lambda-calculus into μABC, and extend it to a translation of a lambda-calculus with aspects.

These translations satisfy basic soundness properties. On one hand, the translations support our hypothesis that μABC 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.

Aspects and the pi-calculus. We identify the features required to support an aspect-oriented style by presenting a translation of μABC into a variant of the polyadic pi-calculus [19]. The motivation for this portion of the paper is the striking analogies between aspects and concurrency.

Firstly, there are the superficial similarities. For example, 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 [5]. Construed this way, the classical parallel composition combinator of concurrency theory suffices for the "obliviousness" criterion on aspect-oriented languages [10] — 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 [10].

Our translation into a polyadic pi-calculus is an attempt to formalize this reasoning. Consider a variant of the pi-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 pi-calculus [20], 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 pi 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 pi-calculus with these mild extensions.

Organization. The rest of the paper is organized as follows. We begin with a description of the syntax and dynamic semantics of μABC. The two following sections describe encodings of the lambda-calculus, both with and without aspects. Next we focus on a language with classes and objects, providing a translation into μABC. 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

We specify the syntax and semantics of μABC, defining a reduction relation $P \twoheadrightarrow P'$ and a garbage-collection relation $P \overset{\mathrm{gc}}{\twoheadrightarrow} P'$. We show that garbage-collection commutes with reduction. The following section presents examples.

Syntax We assume a set of *names*, ranged over by m, n. We further assume that names are partitioned into two disjoint and distinguishable sets:

$$\begin{array}{rl} f,\ldots,\ell,p,\ldots,z & \text{(Role)} \\ a,\ldots,e & \text{(Advice name)} \end{array}$$

We use m, n to range over any name (role or advice). For any grammar X, define the grammars of lists as:

$$\begin{array}{rcll} \vec{X} & ::= & X_1,\ldots,X_n & \text{(Comma-separated lists)} \\ \vec{X}; & ::= & X_1;\ldots X_n; & \text{(Semicolon-terminated lists)} \end{array}$$

Write ε for the empty list.

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

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

Intuitively, a program is a sequence of commands, terminated in a return statement. The commands are:

- The message command "$\mathsf{let}\, x = p \to q : \vec{m}$" causes messages $\vec{m}$ to be sent from source p to target q, and binds the result to x.
- The role declaration "$\mathsf{role}\, p < q$" declares p as a subrole of q.
- The advice declaration "$\mathsf{advice}\, a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q$" declares a as advice triggered by pointcut ϕ; the advice body is Q, with binders for the role of the source (σx), for the role of the target (τy) and for the proceed advice (πb). The proceed advice b is dynamically bound to a set of messages which may be called at any point in the execution of the advice body Q.

The following is an equivalent grammar for programs:

$$\begin{array}{ll} P,Q,R ::= & \text{(Program)} \\ \quad \vec{B}; \mathsf{return}\, v & \text{(Canonical Form)} \\ B,C ::= & \text{(Command)} \\ \quad D & \text{(Dec Command)} \\ \quad \mathsf{let}\, x = p \to q : \vec{m} & \text{(Message Command)} \\ D,E ::= & \text{(Declaration)} \\ \quad \mathsf{role}\, p < q & \text{(Role Declaration)} \\ \quad \mathsf{advice}\, a[\phi] = \sigma x\,.\,\tau y\,.\,\pi b\,.\,Q & \text{(Advice Declaration)} \end{array}$$

Table 1 Domain and Free Names

$$\begin{aligned}
\mathsf{dom}(B_1,\ldots,B_n) &= \mathsf{dom}(B_1)\cup\cdots\cup\mathsf{dom}(B_n)\\
\mathsf{dom}(\mathsf{role}\,p\cdots) &= \{p\}\\
\mathsf{dom}(\mathsf{advice}\,a[\phi]\cdots) &= \{a\}\\
\mathsf{dom}(\mathsf{let}\,x{=}\cdots) &= \{x\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{fn}(\vec{B};\mathsf{return}\,v) &= \mathsf{fn}(\vec{B})\cup(\{v\}\setminus\mathsf{dom}(\vec{B}))\\
\mathsf{fn}(\varepsilon) &= \emptyset\\
\mathsf{fn}(\vec{B};\vec{C}) &= \mathsf{fn}(\vec{B})\cup(\mathsf{fn}(\vec{C})\setminus\mathsf{dom}(\vec{B}))\\
\mathsf{fn}(\mathsf{role}\,p{<}q) &= \{q\}\\
\mathsf{fn}(\mathsf{advice}\,a[\phi]{=}\sigma x\,.\,\tau y\,.\,\pi b\,.\,P) &= \mathsf{fn}(\phi)\cup(\mathsf{fn}(P)\setminus\{x,y,b\})\\
\mathsf{fn}(\mathsf{let}\,x{=}p\to q:\vec{m}) &= \{p,q,\vec{m}\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{fn}(\mathsf{true}) = \quad \mathsf{fn}(\mathsf{false}) \quad &= \emptyset\\
\mathsf{fn}(p\to q:\ell) = \mathsf{fn}(\neg p\to q:\ell) &= \{p,q,\ell\}\\
\mathsf{fn}(\phi\wedge\psi) = \quad \mathsf{fn}(\phi\vee\psi) \quad &= \mathsf{fn}(\phi)\cup\mathsf{fn}(\psi)\\
\mathsf{fn}(\exists x\le p\,.\,\phi) = \mathsf{fn}(\forall x\le p\,.\,\phi) &= \{p\}\cup(\mathsf{fn}(\phi)\setminus\{x\})
\end{aligned}$$

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

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

We define the *domain* and the *free names* of a program in Table 1. Write $\stackrel{\alpha}{=}$ for the equivalence generated by consistent renaming of bound names, ${}^{v}\!/\!{}_{x}$ for the capture-free substitution of value v for free name x, and ${}^{\vec{m}}\!/\!{}_{a}$ for the capture-free substitution of message list $\vec{m}$ for free name a.

Pointcuts The grammar for pointcuts is as follows.

$\phi,\psi ::=$	(Pointcut)
true	(True)
false	(False)
$\phi\wedge\psi$	(Conjunction)
$\phi\vee\psi$	(Disjunction)
$\exists x\le p\,.\,\phi$	(Existential)
$\forall x\le p\,.\,\phi$	(Universal)
$p\to q:\ell$	(Call)
$\neg p\to q:\ell$	(Not Call)

Table 2 defines an ordering relation on names "$\vec{D};\vdash p\le q$" and a satisfaction relation for pointcuts: "$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \phi$". 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 $\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \psi$.

On the face of it, the definition of quantifiers in Table 2 relies on potential extensions of $\vec{D}$ and is not computationally realizable. We give an equivalent intensional definition in Appendix B that performs only finitely many tests.

Table 2 Point Cut Semantics

$\vec{D};\vdash p\le p$	always
$\vec{D};\vdash p\le r$	if for some q, $\vec{D};\vdash p\le q$ and $\mathsf{role}\,q{<}r\in\vec{D}$

$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \mathsf{true}$	always
$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ p\to q:\ell$	always
$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \neg p'\to q':\ell'$	if $(p\to q:\ell)\ne(p'\to q':\ell')$
$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \phi\wedge\psi$	if $\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \phi$ and $\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \psi$
$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \phi\vee\psi$	if $\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \phi$ or $\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \psi$
$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \exists x\le r\,.\,\phi$	if for some $\vec{E}$; and some s, $\vec{D};\vec{E};\vdash s\le r$ and $\vec{D};\vec{E};\vdash p\to q:\ell\ \mathsf{sat}\ \phi[{}^{s}\!/\!{}_{x}]$
$\vec{D};\vdash p\to q:\ell\ \mathsf{sat}\ \forall x\le r\,.\,\phi$	if for every $\vec{E}$; and every s, $\vec{D};\vec{E};\vdash s\le r$ implies $\vec{D};\vec{E};\vdash p\to q:\ell\ \mathsf{sat}\ \phi[{}^{s}\!/\!{}_{x}]$

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 rule treats the rightmost message in a sequence.

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

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 domains of Q and P. We define the notation "$\mathsf{let}\,z{=}Q;P$" to abstract the details of the required renaming.

$$\mathsf{let}\,z{=}Q;P \triangleq \vec{B};P[{}^{v}\!/\!{}_{z}]$$
$$\text{where } \mathsf{dom}(\vec{B})\cap\mathsf{fn}(P)=\emptyset \text{ and } Q\stackrel{\alpha}{=}\vec{B};\mathsf{return}\,v$$

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

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

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. Define garbage collection as the preorder generated by the following rules:

$\vec{D};\mathsf{role}\, p < q; P \overset{\mathsf{gc}}{\rightarrow} \vec{D}; P$
 where $p \notin \mathsf{fn}(P)$

$\vec{D};\mathsf{role}\, p < q; \vec{E}; \mathsf{advice}\,[\phi] \cdots ; P \overset{\mathsf{gc}}{\rightarrow} \vec{D}; \mathsf{role}\, p < q; \vec{E}; P$
 where $p \notin \mathsf{fn}(\vec{E}; P)$
 and $\phi = r \rightarrow p : \ell$
 or $\phi = \exists x \leq r . \exists y \leq p . x \rightarrow y : \ell$
 or $\phi = \exists x \leq p . \exists y \leq r . x \rightarrow y : \ell$

$\vec{D}; \mathsf{advice}\,[\phi] \cdots ; \vec{E}; P \overset{\mathsf{gc}}{\rightarrow} \vec{D}; \vec{E}; P$
 where $\mathsf{advice}\; a[\phi] = \sigma x . \tau y . R \in \vec{E}$

$\vec{B}; \mathsf{let}\, z = p \rightarrow q : \vec{m}, a, \vec{n}; P \overset{\mathsf{gc}}{\rightarrow} \vec{B}; \mathsf{let}\, z = p \rightarrow q : \vec{m}', a, \vec{n}; P$
 where $\mathsf{advice}\; a[\phi] = \sigma x . \tau y . R \in \vec{B}$

The first rule allows for the collection of unused roles. The second and third allow for the collection of advice with pointcuts that never fire. The last two rules allow for the removal of advice that is blocked by preceding advice that never proceeds. Garbage collection is confluent.

PROPOSITION 1 *If $P \overset{\mathsf{gc}}{\rightarrow} P'$ and $P \overset{\mathsf{gc}}{\rightarrow} P''$ then $P' \overset{\mathsf{gc}}{\rightarrow} Q$ and $P'' \overset{\mathsf{gc}}{\rightarrow} Q$ for some Q.*

Garbage collection commutes with reduction.

PROPOSITION 2 *If $P \rightarrow P'$ and $P \overset{\mathsf{gc}}{\rightarrow} Q$ then $Q \rightarrow Q'$ where $P' \overset{\mathsf{gc}}{\rightarrow} Q'$.*

3 Lambda calculus

We provide encodings of the call-by-value lambda calculus and conditional expressions.

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

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

Also, elide role bounds when uninteresting:

$$\mathsf{role}\, p \triangleq \mathsf{role}\, p < \mathsf{top}$$

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

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

Given this shorthand, we encode abstraction and application as follows, where f and g are fresh roles and "call" and "arg" are reserved roles.

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

For example, the encoding of $(\lambda x . P)\, Q$ is "$\vec{D}; g \rightarrow g : \mathsf{call}$", where $\vec{D};$ is as follows:

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

This term reduces as:

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

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.

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

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

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

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

4 Lambda calculus with advice

We sketch an encoding into μABC of the function-based aspect language MinAML defined by Walker, Zdancewic and Ligatti [23]. We treat a subset of core MinAML which retains the essential features of the language. Our goal here is not to provide a complete translation, but rather to show that the essential features of [23] are easily coded in μABC. In particular, in [23], 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 $\mathsf{new}\ p;P$ 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.

At first glance, it would seem that the ability to introduce advice both before and after previous advice would make core MinAML somewhat more expressive that μABC, where advice has a fixed order. However, the presence of the proceed binder in μABC allows us to encode both forms of introduction directly. The encoding is similar to the encoding of before and after advice in terms of around advice in AspectJ [12].

The encoding into μABC is as follows, where p is a fresh role and "hook" is a reserved role.

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

Walker, Zdancewic and Ligatti present the following example. We show the reductions under our encoding. For the purpose of this example, we use 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:

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

and reduction proceeds as follows.

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

5 Class-based calculus with advice

We define a class-based language and present its translation into μABC.

The grammar for class-based programs is given in Table 3. The reduction relation has the form

$$\vec{D};\vec{H};p\{P\} \mapsto \vec{D};\vec{H}';p\{P'\}$$

where p is the *controlling object* of program P. The controlling object is the source object for method invocations. The target object, instead, is gleaned from the syntax of method calls "$q.\ell(\vec{v})$". Advice may be attached to method call or field get operations.

The detailed description of the syntactic categories used in the dynamic semantics are in Appendix A. Table 4 defines the reduction relation, which has the form

$$\vec{D};\vec{H};p\{P\} \mapsto \vec{D}';\vec{H}';p\{P'\}.$$

We now discuss the rules for method invocation. The first rule involving a call to method ℓ looks up both the relevant advice a and the relevant method implementations $t{::}\ell$. The dynamic type of the object is used in both lookup operations. An execution order is constructed, making the advice have higher priority than the method implementations. Advice is ordered according to the declaration order. Implementations are ordered according to the class hierarchy.

$$\begin{aligned}
&\vec{D};\vec{H};p\{\mathsf{let}\,x = q.\ell(\vec{v});P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\,x = q.\vec{b},\vec{a}(\vec{v});P\}\\
&\quad\text{where } [\vec{a}] = \left[a \,\middle|\, \begin{array}{l}(\mathsf{advice}\,a[\phi]\cdots) \in \vec{D}\\ \text{and } \vec{D};\vec{H};\vdash p \to q:\ell\ \mathsf{sat}\ \phi\end{array}\right]\\
&\quad\text{and } [\vec{b}] = \left[t{::}\ell \,\middle|\, \begin{array}{l}(\mathsf{class}\,t <: u\,\{\ldots,\mathsf{method}\,\ell\cdots,\ldots\}) \in \vec{D}\\ \text{and } \vec{D};\ \vec{H};\vdash q:t\end{array}\right]
\end{aligned}$$

Table 3 Syntax of class-based programs

$P,Q,R ::=$	(Program)
$\quad B;P$	(Command Program)
$\quad \mathsf{if}\ p:t\ \mathsf{then}\ P\ \mathsf{else}\ Q$	(If Program)
$\quad \mathsf{return}\, v$	(Return Program)
$B,C ::=$	(Command)
$\quad \vec{H}$	(Heap Command)
$\quad \mathsf{let}\, x = p\{Q\}$	(Let Command)
$\quad \mathsf{let}\, x = p.\ell(\vec{v})$	(Call Method Command)
$\quad \mathsf{let}\, x = p.\vec{a}(\vec{v})$	(Call Advice Command)
$\quad \mathsf{let}\, x = p.f$	(Get Field Command)
$\quad \mathsf{let}\, x = p.\vec{m}$	(Get Advice Command)
$\quad \mathsf{set}\, p.f = v$	(Set Command)
$D,E ::=$	(Declaration)
$\quad \mathsf{class}\, t <: u\, \{\vec{M}\}$	(Class Declaration)
$\quad \mathsf{advice}\, a[\phi](\vec{x})\, \{Q\}$	(Advice Declaration)
$H,I ::=$	(Heap)
$\quad \mathsf{object}\, p:t\, \{\vec{F}\}$	(Object Declaration)
$F,G ::=$	(Field)
$\quad \mathsf{field}\, f = v$	(Field Declaration)
$M,N ::=$	(Method)
$\quad \mathsf{method}\, \ell(\vec{x})\, \{Q\}$	(Method Declaration)
$\phi,\psi ::=$	(Pointcut)
$\quad \cdots$	(As for μABC)
$\quad \exists x:t\,.\,\phi$	(Existential)
$\quad \forall x:t\,.\,\phi$	(Universal)

The advice is then executed in order

$$\begin{array}{l}\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{a},a(\vec{v});P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = p\{Q'\};P\}\\ \quad \text{where } (\mathsf{advice}\, a[\cdots](\vec{x})\,\{Q\}) \in \vec{D}\\ \quad \text{and } Q' = Q[{}^{p}\!/\!_{\mathsf{this}}, {}^{q}\!/\!_{\mathsf{target}}, {}^{\vec{a}}\!/\!_{\mathsf{proceed}}, {}^{\vec{v}}\!/\!_{\vec{x}}]\end{array}$$

until the advice is exhausted, at which point a method implementation is chosen.

$$\begin{array}{l}\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{a},t{::}\ell(\vec{v});P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = q\{Q'\};P\}\\ \quad \text{where } (\mathsf{class}\, t <: u\,\{\ldots,\mathsf{method}\,\ell(\vec{x})\,\{Q\},\ldots\}) \in \vec{D}\\ \quad \text{and } Q' = Q[{}^{q}\!/\!_{\mathsf{this}}, {}^{\vec{v}}\!/\!_{\vec{x}}, {}^{\vec{a}}\!/\!_{\mathsf{super}}]\end{array}$$

Our translation into μABC requires that a class be declared before any the advice which might effect it. For simplicity, we impose a stronger requirement. The class-based language allows updates to the heap but not to the set of declarations.

Note that in the class-based language, the heap is cyclic. This is needed because of field update, which can create cyclic heaps. In comparison, no declarations in μABC create cyclic scope.

Table 4 Class-based reduction

$$\begin{array}{l}
\vec{D};\vec{H};p\{\mathsf{if}\ q:t\ \mathsf{then}\ P\ \mathsf{else}\ Q\} \mapsto \vec{D};\vec{H};p\{R\}\\
\quad \text{where } R = P \text{ if } \vec{D};\ \vec{H};\vdash q:t \text{ and } R = Q \text{ otherwise}\\
\vec{D};\vec{H};p\{\vec{I};Q\} \mapsto \vec{D};\vec{H};\vec{I}';p\{Q'\}\\
\quad \text{where } \vec{I};Q \overset{\alpha}{=} \vec{I}';Q' \text{ and } \mathsf{dom}(\vec{H}) \cap \mathsf{dom}(\vec{I}') = \emptyset\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q\{Q\};P\} \mapsto \vec{D}';\vec{H}';p\{\mathsf{let}\, x = q\{Q'\};P\}\\
\quad \text{where } \vec{D};\vec{H};q\{Q\} \mapsto \vec{D}';\vec{H}';q\{Q'\}\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q\{\mathsf{return}\, v\};P\} \mapsto \vec{D};\vec{H};p\{P[{}^{v}\!/\!_{x}]\}\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\ell(\vec{v});P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{b},\vec{a}(\vec{v});P\}\\
\quad \text{where } [\vec{a}] = \left[a \,\middle|\, \begin{array}{l}(\mathsf{advice}\, a[\phi]\cdots) \in \vec{D}\\ \text{and } \vec{D};\vec{H};\vdash p \to q:\ell\ \mathsf{sat}\ \phi\end{array} \right]\\
\quad \text{and } [\vec{b}] = \left[t{::}\ell \,\middle|\, \begin{array}{l}(\mathsf{class}\, t <: u\,\{\ldots,\mathsf{method}\,\ell\cdots,\ldots\}) \in \vec{D}\\ \text{and } \vec{D};\ \vec{H};\vdash q:t\end{array} \right]\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{a},a(\vec{v});P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = p\{Q'\};P\}\\
\quad \text{where } (\mathsf{advice}\, a[\cdots](\vec{x})\,\{Q\}) \in \vec{D}\\
\quad \text{and } Q' = Q[{}^{p}\!/\!_{\mathsf{this}}, {}^{q}\!/\!_{\mathsf{target}}, {}^{\vec{a}}\!/\!_{\mathsf{proceed}}, {}^{\vec{v}}\!/\!_{\vec{x}}]\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{a},t{::}\ell(\vec{v});P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = q\{Q'\};P\}\\
\quad \text{where } (\mathsf{class}\, t <: u\,\{\ldots,\mathsf{method}\,\ell(\vec{x})\,\{Q\},\ldots\}) \in \vec{D}\\
\quad \text{and } Q' = Q[{}^{q}\!/\!_{\mathsf{this}}, {}^{\vec{v}}\!/\!_{\vec{x}}, {}^{\vec{a}}\!/\!_{\mathsf{super}}]\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q.f;P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = q.v,\vec{a};P\}\\
\quad \text{where } [\vec{a}] = \left[a \,\middle|\, \begin{array}{l}(\mathsf{advice}\, a[\phi]\cdots) \in \vec{D}\\ \text{and } \vec{D};\vec{H};\vdash p \to q:f\ \mathsf{sat}\ \phi\end{array} \right]\\
\quad \text{and } \vec{H} = (\vec{H}_1, \mathsf{object}\, q:t\,\{\vec{F}_1, \mathsf{field}\, f = v, \vec{F}_2\}, \vec{H}_2)\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{m},a;P\} \mapsto \vec{D};\vec{H};p\{\mathsf{let}\, x = p\{Q'\};P\}\\
\quad \text{where } (\mathsf{advice}\, a[\cdots](\vec{x})\,\{Q\}) \in \vec{D}\\
\quad \text{and } Q' = Q[{}^{p}\!/\!_{\mathsf{this}}, {}^{q}\!/\!_{\mathsf{target}}, {}^{\vec{m}}\!/\!_{\mathsf{proceed}}]\\
\vec{D};\vec{H};p\{\mathsf{let}\, x = q.\vec{m},v;P\} \mapsto \vec{D};\vec{H};p\{P[{}^{v}\!/\!_{x}]\}\\
\vec{D};\vec{H};p\{\mathsf{set}\, q.f = v;P\} \mapsto \vec{D};\vec{H}';p\{P\}\\
\quad \text{where } \vec{H} = (\vec{H}_1, \mathsf{object}\, q:t\,\{\vec{F}_1, \mathsf{field}\, f = w, \vec{F}_2\}, \vec{H}_2)\\
\quad \text{and } \vec{H}' = (\vec{H}_1, \mathsf{object}\, q:t\,\{\vec{F}_1, \mathsf{field}\, f = v, \vec{F}_2\}, \vec{H}_2)
\end{array}$$

Translation We now turn our attention to the translation into μABC. A semantic configuration $\vec{D};\vec{H};p\{P\}$ is translated as follows.

$$[\![\vec{D};\vec{H};p\{P\}]\!] = [\![\vec{D};]\!][\![\vec{H};]\!]p[\![P]\!]$$

The translation function is described in Table 5. The translation on pointcuts, declarations and heap elements is generated homomorphically.

The translation of methods and fields warrants comment. These functions record the class t of a method and the object p of a field. The special variables this and super may appear in the method body Q. The treatment of this is standard: this is the target of the method call. The treatment of super is non-standard: super is not an object variable, but a *method* variable, referencing the superclass method which

Table 5 Translation from class-based calculus into μABC

$$
\begin{aligned}
[\![\exists x : t \,.\, \phi]\!] &= \exists x \leq t \,.\, [\![\phi]\!] \\
[\![\forall x : t \,.\, \phi]\!] &= \forall x \leq t \,.\, [\![\phi]\!] \\
[\![\mathsf{class}\ t <: u\, \{\vec{M}\};]\!] &= \mathsf{role}\ t < u;\ \ t[\![\vec{M};]\!] \\
[\![\mathsf{object}\ p : t\, \{\vec{F}\};]\!] &= \mathsf{role}\ p < t;\ \ p[\![\vec{F};]\!] \\
\\
p[\![\mathsf{field}\ f = v;]\!] &= \mathsf{advice}\ a[\mathsf{false}] = \\
&\qquad \mathsf{return}\ v; \\
&\quad\ \mathsf{advice}\ [p\,.\,f] = \\
&\qquad \sigma x\,.\,\tau y\,.\,\pi b\,.\, x \to y : a, b; \\
t[\![\mathsf{method}\ \ell(\vec{x})\ \{Q\};]\!] &= \mathsf{advice}\ [t\,.\,\ell] = \\
&\qquad \tau \mathsf{this}\,.\,\pi \mathsf{super}. \\
&\qquad\ \ \lambda \vec{x}\,.\, \mathsf{this}[\![Q]\!]; \\
[\![\mathsf{advice}\ a[\phi](\vec{x})\ \{Q\};]\!] &= \mathsf{advice}\ a[[\![\phi]\!]] = \\
&\qquad \sigma \mathsf{this}\,.\,\tau \mathsf{target}\,.\,\pi \mathsf{proceed}. \\
&\qquad\ \ \lambda \vec{x}\,.\, \mathsf{this}[\![Q]\!]; \\
\\
p[\![\vec{H};P]\!] &= [\![\vec{H};]\!]\, p[\![P]\!] \\
p[\![\mathsf{if}\ q : t\ \mathsf{then}\ P\ \mathsf{else}\ Q]\!] &= \mathsf{if}\ q \leq t\ \mathsf{then}\ p[\![P]\!]\ \mathsf{else}\ p[\![Q]\!] \\
p[\![\mathsf{let}\, x = q\{Q\}; P]\!] &= \mathsf{let}\, x = q[\![Q]\!]; p[\![P]\!] \\
p[\![\mathsf{let}\, x = q.\ell(\vec{v}); P]\!] &= \mathsf{let}\, x = (p \to q : \ell)\, \vec{v}; p[\![P]\!] \\
p[\![\mathsf{let}\, x = q.\vec{a}(\vec{v}); P]\!] &= \mathsf{let}\, x = (p \to q : \vec{a})\, \vec{v}; p[\![P]\!] \\
p[\![\mathsf{let}\, x = q.f; P]\!] &= \mathsf{let}\, x = p \to q : f; p[\![P]\!] \\
p[\![\mathsf{let}\, x = p.\vec{m}; P]\!] &= \mathsf{let}\, x = p \to q : \vec{m}; p[\![P]\!] \\
p[\![\mathsf{set}\, q.f = v; P]\!] &= q[\![\mathsf{field}\ f = v;]\!]\, p[\![P]\!] \\
p[\![\mathsf{return}\, v]\!] &= \mathsf{return}\, v
\end{aligned}
$$

the subclass overrides. The encoding of field declarations uses two pieces of advice. The first provides a name a at which to store the value of the field. The second triggers the proceed advice on the field before calling a to return the stored value.

The translation function on programs records the controlling object p. The controlling object is used as the source role for messages.

The translation preserves and reflects reduction.

PROPOSITION 3 *If* $\vec{D};\vec{H};p\{P\} \mapsto \vec{D}';\vec{H}';p\{P'\}$ *and* $[\![\vec{D};\vec{H};p\{P\}]\!] = Q$ *then* $Q \twoheadrightarrow^* \overset{\mathsf{gc}}{\twoheadrightarrow} Q'$ *where* $[\![\vec{D}';\vec{H}';p\{P'\}]\!] = Q'$.

PROPOSITION 4 *If* $Q \twoheadrightarrow Q'$ *and* $[\![\vec{D};\vec{H};p\{P\}]\!] = Q$ *then* $\vec{D};\vec{H};p\{P\} \mapsto \vec{D}';\vec{H}';p\{P'\}$ *for some* $\vec{D}',\vec{H}',P'$.

THEOREM 5 *If* $[\![\vec{D};\vec{H};p\{P\}]\!] = Q$ *then* $\vec{D};\vec{H};p\{P\} \mapsto^* \vec{D}';\vec{H}';p\{\mathsf{return}\, v\}$ *iff* $Q \twoheadrightarrow^* \vec{E};\mathsf{return}\, v$.

6 Polyadic pi-calculus with pointcuts

We define an extended polyadic π-calculus with pointcuts. We show that a sublanguage of μABC can be translated into this π-calculus.

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

$\phi, \psi ::=$	(Pointcut)
$\cdots$	(As for μABC)
$\vec{x}$	(Atom)
$\neg\vec{x}$	(Not Atom)

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

$P, Q, R ::=$	(Process)
$z(\vec{x})\, P$	(Input Process)
$z\langle\vec{x}\rangle$	(Output Process)
$P \mid Q$	(Parallel Process)
$\mathbf{0}$	(Terminated Process)
$!P$	(Replicated Process)
$\mathsf{new}\ x < y; P$	(New Name Process)
$[\vec{x}\ \mathsf{sat}\ \phi]\, P$	(Match Process)

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:

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

The dynamic semantics $X \vdash P \rightarrowtail P'$ is given by the usual pi-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 [19] and summarized in Appendix C.

Define $\neg\phi$ as the DeMorgan dualized version of ϕ. The axioms for garbage collection are as follows.

$$X \vdash \mathsf{new}\ x < y; P \overset{\mathsf{gc}}{\rightarrowtail} P \quad \text{where } x \notin \mathsf{fn}(P)$$
$$X \vdash [\vec{x}\ \mathsf{sat}\ \phi]\, P \overset{\mathsf{gc}}{\rightarrowtail} \mathbf{0} \quad \text{where } X \vdash \vec{x}\ \mathsf{sat}\ \neg\phi$$

Translation We now show that μABC can be translated (via a spaghetti-coded cps transform) into our polyadic pi-calculus. In our translation, advice is simply placed in parallel with the advised code. The superficial complexity of the translation reflects the programming needed to make a single message of interest activate potentially several pieces of advice.

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 $p \rightarrow q : \vec{m}$ 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 $P \rightarrowtriangle P'$, then P' is not necessarily user code. We defined user closed code, which is closed under reduction.

DEFINITION 6 A program $P = \vec{D}; Q$ is *user closed* whenever, for any call $p \rightarrow q : \vec{m}$ 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 an empty sequence ε; or
- $\vec{m}$ is a sequence "$\vec{a}, b$" and for some r, s and ℓ:

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

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 6. 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.

Write $P \rightarrowtail Q$ as shorthand for $\vdash P \rightarrowtail Q$ and $P \overset{\mathsf{gc}}{\rightarrowtail} Q$ as shorthand for $\vdash P \overset{\mathsf{gc}}{\rightarrowtail} Q$.

Table 6 Translation from μABC to π

$$\begin{array}{l}
\vec{D}; \vdash [\![D; P]\!](k, c, \rho) \\
\quad = \mathsf{new}\; a < c; (Q \mid !a(r, s, \ell, x, y, k', c')\, (\\
\qquad\qquad [r, s, \ell \;\mathsf{sat}\; [\![\phi]\!]]\, Q' \\
\qquad\quad \mid [r, s, \ell \;\mathsf{sat}\; \neg[\![\phi]\!]]\, c\langle r, s, \ell, x, y, k', c' \rangle \\
\qquad\;)) \\
\qquad \text{where } D = \mathsf{advice}\; a[\phi] = \sigma x \,.\, \tau y \,.\, \pi b \,.\, P' \\
\qquad \text{and } \vec{D}; D; \vdash [\![P]\!](k, a, \rho) = Q \\
\qquad \text{and } \vec{D}; D; \vdash [\![P']\!](k', c', \rho \cup \{b \mapsto (c, r, s, \ell)\}) = Q'
\end{array}$$

$$\begin{array}{l}
\vec{D}; \vdash [\![D; P]\!](k, c, \rho) \\
\quad = \mathsf{new}\; p < q; Q \\
\qquad \text{where } D = \mathsf{role}\; p < q \\
\qquad \text{and } \vec{D}; D; \vdash [\![P]\!](k, c, \rho) = Q
\end{array}$$

$$\begin{array}{l}
\vec{D}; \vdash [\![\mathsf{let}\, x = p \rightarrow q : \varepsilon; P]\!](k, c, \rho) \\
\quad = \mathsf{new}\; k' < k; (\mathsf{error}\langle p, q, \ell, p, q, k', c \rangle \mid k'(x, c')\, Q)
\end{array}$$

$$\begin{array}{l}
\vec{D}; \vdash [\![\mathsf{let}\, x = p \rightarrow q : \ell; P]\!](k, c, \rho) \\
\quad = \mathsf{new}\; k' < k; (c\langle p, q, \ell, p, q, k', c \rangle \mid k'(x, c')\, Q) \\
\qquad \text{where } \vec{D}; \vdash [\![P]\!](k, c', \rho) = Q
\end{array}$$

$$\begin{array}{l}
\vec{D}; \vdash [\![\mathsf{let}\, x = p \rightarrow q : b; P]\!](k, c, \rho) \\
\quad = \mathsf{new}\; k' < k; (d\langle r, s, \ell, p, q, k', c \rangle \mid k'(x, c')\, Q) \\
\qquad \text{where } \rho(b) = (d, r, s, \ell) \\
\qquad \text{and } \vec{D}; \vdash [\![P]\!](k, c', \rho) = Q
\end{array}$$

$$\begin{array}{l}
\vec{D}; \vdash [\![\mathsf{let}\, x = p \rightarrow q : \vec{a}, b; P]\!](k, c, \rho) \\
\quad = \mathsf{new}\; k' < k; (b\langle r, s, \ell, p, q, k', c \rangle \mid k'(x, c')\, Q) \\
\qquad \text{where } [\vec{a}, b] = \left[a \;\middle|\; \begin{array}{l} \mathsf{advice}\, a[\phi] \cdots \in \vec{D} \\ \vec{D}; \vdash a \leq b \\ \vec{D}; \vdash r \rightarrow s : \ell \;\mathsf{sat}\; \phi \end{array} \right] \\
\qquad \text{and } \vec{D}; \vdash [\![P]\!](k, c', \rho) = Q
\end{array}$$

$$\begin{array}{l}
\vec{D}; \vdash [\![\mathsf{return}\; v]\!](k, c, \rho) \\
\quad = k\langle v, c \rangle
\end{array}$$

PROPOSITION 7 *For any user closed program P, if $[\![P]\!] = Q$ and $P \rightarrowtriangle P'$ then $Q \rightarrowtail^* \overset{\mathsf{gc}}{\rightarrowtail} Q'$ where $[\![P']\!] = Q'$.*

PROPOSITION 8 *For any user closed program P, if $[\![P]\!] = Q$ and $Q \rightarrowtail Q'$ then $P \rightarrowtriangle P'$ for some P'.*

THEOREM 9 *For any user code P, if $[\![P]\!] = Q$ then $P \rightarrowtriangle^* \vec{D}; \mathsf{return}\; v$ iff $Q \rightarrowtail^* \mathsf{new}\; \vec{x} < \vec{y}; (Q' \mid \mathsf{result}\langle v \rangle)$.*

7 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. [7] 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 LICS, aspects provide two intriguing opportunities. First, the techniques and approaches that have been explored at LICS 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] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer Verlag, 1996.

[2] 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.

[3] Association of Computing Machinery. *Communications of the ACM*, Oct 2001.

[4] L. Bauer, J. Ligatti, and D. Walker. A calculus for composing security policies. Technical Report TR-655-02, Dept. of Computer Science, Princeton University, 2002.

[5] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.

[6] 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`.

[7] C. Clifton, G. T. Leavens, and M. 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.

[8] D. S. Dantas and D. Walker. Aspects, information hiding and modularity. Submitted for publication, at `http://www.cs.princeton.edu/~dpw/papers/aspectml-nov03.pdf`, 2003.

[9] B. Dufour, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of aspectj programs, 2003.

[10] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.

[11] R. Jagadeesan, A. Jeffrey, and J. Riely. A typed calculus for aspect-oriented programs. Submitted for publication, at `ftp://fpl.cs.depaul.edu/pub/rjagadeesan/typedABL.pdf`, 2003.

[12] R. Jagadeesan, A. Jeffrey, and J. 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.

[13] G. Kiczales and Y. Coady. `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[16] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996.

[17] 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`.

[18] S. Matsuoka and A. 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.

[19] 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.

[20] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–40, 1992.

[21] 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.

[22] D. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Conference Record of AOSD 03: The 2nd International Conference on Aspect Oriented Software Development*, 2003.

[23] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Conference Record of ICFP 03: The ACM SIGPLAN International Conference on Functional Programming*, 2003.

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

A Class-based reduction

The syntactic categories involved in the reduction relation are described below.

Define the domain of a declarations and commands to be as for μABC with the addition of the following two rules.

$$
\begin{array}{rcl}
\mathsf{dom}(\mathsf{class}\, t \cdots) & = & \{t\} \\
\mathsf{dom}(\mathsf{set}\, p.f = v) & = & \emptyset
\end{array}
$$

The free names of a program are defined as follows.

$$
\begin{array}{rl}
\mathsf{fn}(B;P) = & \mathsf{fn}(B) \cup (\mathsf{fn}(P) \setminus \mathsf{dom}(B)) \\
\mathsf{fn}(\mathsf{if}\ p : t\ \mathsf{then}\ P\ \mathsf{else}\ Q) = & \{p,t\} \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}(\mathsf{return}\, v) = & \{v\} \\
\mathsf{fn}(\varepsilon) = & \emptyset \\
\mathsf{fn}(\vec{H},\vec{I}) = & (\mathsf{fn}(\vec{H}) \setminus \mathsf{dom}(\vec{I})) \cup (\mathsf{fn}(\vec{I}) \setminus \mathsf{dom}(\vec{H})) \\
\mathsf{fn}(\mathsf{let}\, x = p\{Q\}) = & \{p\} \cup \mathsf{fn}(Q) \\
\mathsf{fn}(\mathsf{let}\, x = p.\ell(\vec{v})) = & \{p,\ell,\vec{v}\} \\
\mathsf{fn}(\mathsf{let}\, x = p.\vec{a}(\vec{v})) = & \{p,\vec{a},\vec{v}\} \\
\mathsf{fn}(\mathsf{let}\, x = p.f) = & \{p,f\} \\
\mathsf{fn}(\mathsf{set}\, p.f = v) = & \{p,f,v\} \\
\mathsf{fn}(\mathsf{class}\, t <: u\,\{\vec{M}\}) = & \{u\} \cup \mathsf{fn}(\vec{M}) \\
\mathsf{fn}(\mathsf{advice}\, a[\phi](\vec{x})\,\{Q\}) = & \mathsf{fn}(\phi) \cup (\mathsf{fn}(Q) \setminus \{\vec{x}, \mathsf{target}, \mathsf{this}, \mathsf{proceed}\}) \\
\mathsf{fn}(\mathsf{object}\, p : t\,\{\vec{F}\}) = & \{t\} \cup \mathsf{fn}(\vec{F}) \\
\mathsf{fn}(\mathsf{field}\, f = v) = & \{f,v\} \\
\mathsf{fn}(\mathsf{method}\, \ell(\vec{x})\,\{Q\}) = & \{\ell\} \cup (\mathsf{fn}(Q) \setminus \{\vec{x}, \mathsf{this}, \mathsf{super}\})
\end{array}
$$

The instance-of relation $\vec{D};\ \vec{H};\vdash p : t$ indicates that object p is an instance of class t.

$$
\begin{array}{ll}
\vec{D};\ \vec{H};\vdash p : t & \text{if } \mathsf{object}\ p : t\,\{\vec{F}\} \in \vec{H} \\
\vec{D};\ \vec{H};\vdash p : u & \text{if } \mathsf{class}\ t <: u\,\{\vec{M}\} \in \vec{D} \text{ and } \vec{D};\ \vec{H};\vdash p : t
\end{array}
$$

The satisfaction relation $\vec{D};\vec{H};\vdash p \to q : \ell\ \mathsf{sat}\ \phi$ is the same as for μABC, except for the quantifiers.

$$
\begin{array}{ll}
\vec{D};\vec{H};\vdash p \to q : \ell\ \mathsf{sat}\ \exists x : t\,.\,\phi & \text{if for some } \vec{E};\text{ and some } s, \\
& \vec{D};\vec{E};\ \vec{H};\vdash s : t \text{ and} \\
& \vec{D};\vec{E};\vec{H};\vdash p \to q : \ell\ \mathsf{sat}\ \phi[s/x] \\
\vec{D};\vec{H};\vdash p \to q : \ell\ \mathsf{sat}\ \forall x : t\,.\,\phi & \text{if for every } \vec{E};\text{ and every } s, \\
& \vec{D};\vec{E};\ \vec{H};\vdash s : t \text{ implies} \\
& \vec{D};\vec{E};\vec{H};\vdash p \to q : \ell\ \mathsf{sat}\ \phi[s/x]
\end{array}
$$

B Pointcut Semantics

PROPOSITION 10 *The definition of pointcut semantics is equivalent to the following:*

$$
\frac{\begin{array}{l}\vec{D};\vdash p \to q : \ell\ \mathsf{sat}\ \phi[\bot/x] \\ \textit{or } \vec{D};\vdash p \to q : \ell\ \mathsf{sat}\ \phi[s/x] \\ \textit{for some } s \textit{ s.t. } \vec{D};\vdash s \le r\end{array}}{\vec{D};\vdash p \to q : \ell\ \mathsf{sat}\ \exists x \le r\,.\,\phi}
\qquad
\frac{\begin{array}{l}\vec{D};\vdash p \to q : \ell\ \mathsf{sat}\ \phi[\bot/x] \\ \textit{and } \vec{D};\vdash p \to q : \ell\ \mathsf{sat}\ \phi[s/x] \\ \textit{for every } s \textit{ s.t. } \vec{D};\vdash s \le r\end{array}}{\vec{D};\vdash p \to q : \ell\ \mathsf{sat}\ \forall x \le r\,.\,\phi}
$$

where we define $\phi[\bot/x]$ as generated homomorphically by:

$$
\begin{array}{l}
(\exists y \le p\,.\,\phi)[\bot/x] \\
\quad = \begin{cases} \phi[\bot/x, \bot/y] & \textit{if } p = x \\ \exists y \le p\,.\,(\phi[\bot/x]) & \textit{otherwise} \end{cases} \\
(\forall y \le p\,.\,\phi)[\bot/x] \\
\quad = \begin{cases} \phi[\bot/x, \bot/y] & \textit{if } p = x \\ \forall y \le p\,.\,(\phi[\bot/x]) & \textit{otherwise} \end{cases} \\
(p \to q : \ell)[\bot/x] \\
\quad = \begin{cases} \mathsf{false} & \textit{if } x \in \{p,q,\ell\} \\ p \to q : \ell & \textit{otherwise} \end{cases} \\
(\neg p \to q : \ell)[\bot/x] \\
\quad = \begin{cases} \mathsf{true} & \textit{if } x \in \{p,q,\ell\} \\ \neg p \to q : \ell & \textit{otherwise} \end{cases}
\end{array}
$$

C Semantics of the pi calculus

The domain of a process is generated homomorphically from the following.

$$
\begin{array}{rcl}
\mathsf{dom}(\mathsf{new}\ x < y; P) & = & \{x\} \\
\mathsf{dom}(z(\vec{x})\,P) & = & \{\vec{x}\}
\end{array}
$$

The free names of a process are defined as follows

$$
\begin{array}{rcl}
\mathsf{fn}(z(\vec{x})\,P) & = & \{z\} \cup \mathsf{fn}(P) \setminus \{\vec{x}\} \\
\mathsf{fn}(z\langle\vec{x}\rangle) & = & \{z,\vec{x}\} \\
\mathsf{fn}(P \mid Q) & = & \mathsf{fn}(P) \cup \mathsf{fn}(Q) \\
\mathsf{fn}(\mathbf{0}) & = & \emptyset \\
\mathsf{fn}(!P) & = & \mathsf{fn}(P) \\
\mathsf{fn}(\mathsf{new}\ x < y; P) & = & \{y\} \cup \mathsf{fn}(P) \setminus \{x\} \\
\mathsf{fn}([\vec{x}\ \mathsf{sat}\ \phi]\,P) & = & \{\vec{x}\} \cup \mathsf{fn}(\phi) \cup \mathsf{fn}(P)
\end{array}
$$

Alpha equivalence and substitution are standard. The structural equivalence is the least equivalence generated by the following.

$$
\begin{array}{rl}
P \equiv P' & \text{if } P \overset{\alpha}{=} P' \\
P \mid Q \equiv P' \mid Q' & \text{if } P \equiv P' \text{ and } Q \equiv Q' \\
\mathsf{new}\ x < y; P \equiv \mathsf{new}\ x < y; P' & \text{if } P \equiv P' \\
P \mid Q \equiv Q \mid P & \\
P \mid \mathbf{0} \equiv P & \\
!P \equiv P \mid !P & \\
\mathsf{new}\ x < y; (P \mid Q) \equiv P \mid \mathsf{new}\ x < y; Q & \text{if } x \notin \mathsf{fn}(P) \\
\mathsf{new}\ x < y; \mathsf{new}\ x' < y'; P \equiv \mathsf{new}\ x' < y'; \mathsf{new}\ x < y; P & \\
& \text{if } x \ne y' \text{ and } x' \ne y
\end{array}
$$

The reduction rules are as follows.

$$\frac{}{X \vdash z(\vec{x})\,P \mid z\langle\vec{y}\rangle \rightarrowtail P[\vec{y}/\vec{x}]}$$

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

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

$$\frac{X \vdash P \rightarrowtail P'}{X \vdash P \mid Q \rightarrowtail P' \mid Q}$$

$$\frac{X \vdash P \rightarrowtail Q \quad P \equiv P' \quad Q \equiv Q'}{X \vdash P' \rightarrowtail Q'}$$

Define $\neg\phi$ as the DeMorgan dualized version of ϕ. Garbage collection is defined as the least preorder generated from the following rules.

$$\frac{x \notin \mathsf{fn}(P)}{X \vdash \mathsf{new}\ x{<}y; P \overset{\mathsf{gc}}{\rightarrowtail} P}$$

$$\frac{X \vdash \vec{x}\ \mathsf{sat}\ \neg\phi}{X \vdash [\vec{x}\ \mathsf{sat}\ \phi]\,P \overset{\mathsf{gc}}{\rightarrowtail} P}$$

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

$$\frac{X \vdash P \overset{\mathsf{gc}}{\rightarrowtail} P'}{X \vdash P \mid Q \overset{\mathsf{gc}}{\rightarrowtail} P' \mid Q \quad X \vdash Q \mid P \overset{\mathsf{gc}}{\rightarrowtail} Q \mid P'}$$