# Typing For a Minimal Aspect Language

## Preliminary Report

Peter Hui James Riely

DePaul University

{phui,jriely}@cs.depaul.edu

## Abstract

We present a preliminary report on typing systems for polyadic $\mu$ABC, aspect oriented programming—pointcuts and advice—and nothing else. Tuples of uninterpreted names are used to trigger advice. The resulting language is remarkably unstructured: the least common denominator of the pi-calculus and Linda. As such, developing meaningful type systems is a substantial challenge.

Our work is guided by the translation of richly typed languages into $\mu$ABC, specifically function- and class-based languages augmented with advice. The "impedance mismatch" between source and target is severe, and this leads us to a novel treatment of types in $\mu$ABC.

## 1. Introduction

Research on the foundations of aspect-orientation has followed several directions. Much work has found inspiration in functional languages (for example [4, 9]), whereas others have looked to objects (for example [3]). These works follow the view that aspects transform code from some underlying paradigm. In line with our prior work, this paper follows a different route, attempting to understand aspects, in so far as possible, in isolation.

By removing the underlying computational mechanisms, however, it is not clear what aspects are meant to advise. In $\mu$ABC, which we study here, aspects advise tuples of names, drawing inspiration from the pi calculus and coordination languages, such as Linda. A surprisingly expressive computational model emerges, but it is not without difficulties. In particular, the language is almost shockingly unstructured, making any form of analysis seem rather hopeless. Here we make a first attempt at redressing this situation.

$\mu$ABC was introduced in [2] to study as aspects "as primitive computational entities on par with objects, functions and horn-clauses". In that paper, we sketched an encoding of core minAML [11, 9] into $\mu$ABC, but did not provide a full translation. Indeed, we observed that

> $\mu$ABC was deliberately designed to be a small calculus that embodies the essential features of aspects. However, this criterion makes $\mu$ABC an inconvenient candidate to serve in the role of a meta-language that is the target of translations from "full-scale" aspect languages.

Sixth International Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), March 13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM ISBN 1-59593-671-4/07/03 ... $5.00

Echoing this sentiment, Ligatti, Walker and Zdancewic [9] note that "it is unclear what sort of type theory would be needed to establish that [...] translations [into $\mu$ABC] are type-preserving." Here, we take the first steps at providing such a theory, in an attempt to bridge the gap between the chaos of pure aspects and the relatively well-behaved worlds of functions and objects.

As our point of departure, we use the polyadic version of $\mu$ABC introduced in [5]. In this variant, the events upon which advice triggers are otherwise uninterpreted tuples of ordered names. When modeling a functional language, the elements of the tuple may represent the function and its arguments. When modeling an object language, the elements instead may represent the source and target of a message, along with the message name and arguments. $\mu$ABC itself imposes no interpretation.

When defining a pointcut in $\mu$ABC, one must specify the arity of the events (ie, the length of the tuples) upon which it will trigger. One may treat all of the elements of the tuple as arguments—in which case the pointcut will trigger on any event of that arity—or one may fix some names in the pointcut so that the pointcut will only trigger on events that include the same name in the corresponding position. In addition, one may specify bounded matching, so that any name ordered below the one specified will serve to satisfy the pointcut. In this way $\mu$ABC advice resembles tuple matching Linda.

The chief insight of our work is that the ordering on names suffices to encode simple type systems for function and object languages, as long as one may impose some structure on names. By systematically selecting bounds, and relating the bounds between elements of a tuple, one may specify constraints on the tuple shapes which are allowable. For example, one might require that if the first element of a triple is a subname of $\mathsf{int}{\to}\mathsf{int}$, then the second must be a subname of $\mathsf{int}$, and the third a subname of $\mathsf{int}^{-1}$. Here $\mathsf{int}$, $\mathsf{int}{\to}\mathsf{int}$ and $\mathsf{int}^{-1}$ are simply names, albeit with some structure. One may view this as a protocol that imposes an interpretation on subnames of $\mathsf{int}$, $\mathsf{int}{\to}\mathsf{int}$ and $\mathsf{int}^{-1}$. Tuples that use such subnames must satisfy requirements such as that stated above.

One would expect that the protocol used by a functional language would be different than that of an object language, and again from a logic language. $\mu$ABC may be adapted to any system by specifying both a structure on certain names (ie, those that correspond to types) and a protocol for tuples that "match" them.

We are interested in discovering a general theory of such protocols and establishing its validity in $\mu$ABC. As of yet, we have not reached this ideal, but we have discovered some intermediate results that may be of interest to the FOAL community.

As a first step toward full typing, we have specified a *sorting* for $\mu$ABC, in the flavor of sorting systems for the pi calculus. The sorting is sufficient to guarantee that computation never terminates. We believe that the sorting system is an important first step toward developing a proper typing system; however, in terms of semantic equivalence, sorting itself is clearly inadequate: all well-sorted programs are indistinguishable! Being a continuation calculus, like pi, there is no clear notion of *value* at which one might terminate. Our vision of typing is that well-typed terms either nonterminate or terminate on a tuple of a particular shape, allowing a richer notion of equivalence based on barbed congruence [10]. However, we have not yet developed the technical machinery to specify this.

Further, we have defined translations from function and class based languages into well-sorted $\mu$ABC and, to a certain extent, established their correctness. We hesitate to say that we have *proved* the correctness of these because the results are stated with respect to a "structural congruence". This is a problem for two reasons.

First, as noted above, we have not yet developed techniques to specify that any structural congruence is "reasonable". To establish this requires a meaningful notion of semantic equivalence.

Second, the structural congruence presented here is defined, in part, using the translation itself. This is clearly inadequate. As we discuss in Section 4, the standard semantics of aspect-functional languages [9, 8] specify that proceed substitutions are performed early, at the time of advice lookup; whereas $\mu$ABC performs them late, on demand. This creates technical difficulties relating the two languages, which we believe are best solved by slowing down proceed substitutions in the function language, for example by using explicit substitutions [1]. We do not do this here, however, and therefore introduce a questionable structural rule to solve the problem.

The remainder of the paper proceeds as follows. In the next section we review the syntax and operational semantics of $\mu$ABC. Section 3 presents the sorting system. In Section 4, we present a $\lambda$-calculus with advice and its translation into $\mu$ABC. Given the caveats stated above, we demonstrate the correctness of the translation. The following section does the same for a small object language.

## 2. $\mu$ABC

In this section we present the syntax and evaluation semantics of polyadic $\mu$ABC. We give some examples of evaluation here; further examples can be found in Section 4.3.

We assume disjoint sets of *names*, ranged over by *a–y* and *proceed names*, ranged over by *z*. Names include int, self, Object as well as the structured names used in Sections 4 and 5. Integers and integer-valued expressions are also names, each a proper subname of int. This treatment of integer-valued expressions simplifies examples. The syntax of $\mu$ABC is as follows.

SYNTAX

| | |
|---|---|
| $U,V,S,T \ ::= \ \cdot \ \mid U,\hat{a} \ \mid U,a$ | Events |
| $P,Q \ ::= \ \cdot \ \mid P,\hat{a} \ \mid P,x{:}s$ | Pointcut Atoms |
| $A,B \ ::= \ \{z.P \to M\}$ | Advice |
| $D,E \ ::= \ \cdot \ \mid D;\mathsf{new}\, a{:}s \ \mid D;\mathsf{adv}\, A$ | Declarations |
| $M,N \ ::= \ \mathsf{call}\langle U\rangle \ \mid z\langle U\rangle \ \mid \vec{A}\langle U\rangle \ \mid D;M$ | Terms |

An *event*, $U$, is sequence event atoms, where "$\cdot$" is the empty sequence. *Event atoms* are either *exact* names, decorated with a circumflex, or *inexact* names, which have no decoration.

A *pointcut*, $P$, is a sequence of pointcut atoms. *Pointcut atoms* are either exact or inexact. The exact pointcut atom $\hat{a}$ matches only the exact event atom $\hat{a}$. The inexact pointcut atom $x{:}s$ matches any inexact event atoms whose name is a proper subname of name $s$. At runtime $x$ is bound to the matching name; for this reason we call it a *pointcut variable*.

Advice has the form $\{z.P \to M\}$, where $z$ is a *proceed variable*, $P$ is a pointcut, and $M$ is the body of the advice. The proceed variable $z$ and the pointcut variables in $P$ are bound in $M$. We elide the proceed variable when it does not occur free in the advice body, writing simply $\{P \to M\}$.

A *declaration sequence*, $D$, is a sequence of declarations. The declaration $\mathsf{new}\, a{:}s$ declares name $a$ as a fresh subname of $s$ ($s$ may also be fresh, in which case $a$ may only be matched exactly). The declaration $\mathsf{adv}\, A$ declares advice $A$.

A *term M* may be prefixed with a declaration sequence $D;M$. The new names declared in $D$ are bound in $M$. We identify $(D;E);M$ with $D;(E;M)$ and $\cdot\,;M$ with $M$.

Each term has exactly one "current" event. A term consists of a sequence of declarations, followed by the current event. Events are marked either with a call, a proceed variable, or an advice sequence. In $\vec{A}\langle U\rangle$, $U$ is the current event and $\vec{A}$ is a sequence of pending advice. Advice is executed right-to-left.

**Example 1.** Consider the advice

$$\{\underbrace{z.\hat{f},x{:}\mathsf{int},y{:}\mathsf{int}}_{\text{Pointcut}} \to \underbrace{z\langle\hat{f},y,x\rangle}_{\text{Body}}\}.$$

This advice is triggered when the current event is a triple whose first atom is exactly $\hat{f}$, and whose second and third atoms are both inexact atoms whose names are subnames of int. When it executes, it switches its second and third names, and proceeds on the new event. Supposing that this is the most recently declared advice, the term

$$\mathsf{call}\underbrace{\langle\hat{f},10,20\rangle}_{\text{Current Event}}$$

evaluates to

$$\vec{B},\{z.\hat{f},x{:}\mathsf{int},y{:}\mathsf{int} \to z\langle\hat{f},y,x\rangle\}\langle\hat{f},10,20\rangle$$

where $\vec{B}$ is previously declared advice that triggers on the same event. At this point the term evaluates to

$$\vec{B}\langle\hat{f},20,10\rangle. \qquad \square$$

**Example 2.** Consider the following declaration:

$$D \triangleq \mathsf{adv}\, A;\mathsf{adv}\, B$$
$$A \triangleq \{z.\hat{f},x{:}\mathsf{int} \to M\}$$
$$B \triangleq \{z.\hat{f},x{:}\mathsf{int} \to z\langle\hat{f},42\rangle\}$$

$D$ declares two pieces of advice. Both triggered when the first name of the current event is exactly $\hat{f}$, and the second name is a subname of int.

Evaluation of $D;\mathsf{call}\langle\hat{f},10\rangle$ proceeds as follows. The call triggers advice lookup. Since both pieces of advice match the current event, they are both enqueued. $B$ executes first.

$$D;\mathsf{call}\langle\hat{f},10\rangle \to D;A,B\langle\hat{f},10\rangle$$

$B$ changes the second atom of the event to 42, and proceeds on the next advice

$$\to D;A\langle\hat{f},42\rangle$$

and then the body of $A$ is executed.

$$\to D;M[x := 42] \qquad \square$$

We now present the operational semantics of the language. Declarations, $D$, are used in several definitions. We disallow alpha conversion on the new names in a declaration when the declaration is treated as independent syntax—while $a$ and $b$ are bound in the term "$\mathsf{new}\, a;\mathsf{new}\, b;\mathsf{call}\langle a,b\rangle$," they are free in the declaration sequence "$\mathsf{new}\, a;\mathsf{new}\, b$."

We begin by defining some auxiliary relations. We write $D \triangleright a{:}s$ to indicate that $a$ is properly below $s$ in the order defined by $D$. Thus if $D = \mathsf{new}\ s{:}t\,;\ \mathsf{new}\ a{:}s$ then $D \triangleright a{:}t$ and $D \triangleright a{:}s$. The relation is irreflexive; thus $D \triangleright a{:}a$ never holds.

The *match* relation takes a pointcut $P$, an event $U$, and returns an appropriate *substitution* $\sigma$, if one exists. A substitution is a finite mapping on names. For instance, if $D$ contains $\mathsf{new}\ a{:}\mathsf{int}$, then $D \triangleright match(x{:}\mathsf{int}, \hat{b})(a, \hat{b}) = (a := x)$. If $D$ contained $\mathsf{new}\ a{:}\mathsf{bool}$ instead, then $D \triangleright match(x{:}\mathsf{int}, \hat{b})(a, \hat{b})$ would be undefined. Matching is sensitive to exactness; thus $D \triangleright match(x{:}\mathsf{int}, \hat{b})(a, b)$ is always undefined, since $b$ is exact in the pattern but inexact in the event.

AUXILIARY RELATIONS $(D \triangleright a{:}s)$ $(D \triangleright match(P)(U) = \sigma)$

$\sigma ::= \cdot \mid \sigma, x := a$ Name Substitutions

$D \triangleright a{:}s$ if $D \ni \mathsf{new}\ a{:}s$
$D \triangleright a{:}s$ if $D \triangleright t{:}s$ and $D \ni \mathsf{new}\ a{:}t$

$$
\begin{array}{ll}
D \triangleright match(\cdot)(\cdot) = \cdot & \\
D \triangleright match(P, \hat{a})(U, \hat{a}) = \sigma & \text{if } D \triangleright match(P)(U) = \sigma \\
D \triangleright match(P, x{:}s)(U, a) = (\sigma, x := a) & \text{if } D \triangleright match(P)(U) = \sigma \\
 & \text{and } D \triangleright a{:}s \text{ and } x \notin dom(\sigma)
\end{array}
$$

Write "$D \triangleright match(P)(U)$" if $D \triangleright match(P)(U) = \sigma$ for some $\sigma$. Write "$D \triangleright match(A)(U)$" if $A = \{\_.P \to \_\}$ and $D \triangleright match(P)(U)$.

There are two evaluation rules, shown below:

EVALUATION $(M \to N)$

$$
\begin{array}{l}
D;\, \mathsf{call}\langle U \rangle \to D;\, \vec{A}\langle U \rangle \ \text{ if } \vec{A} = \big(A \mid D \triangleright match(A)(U) \text{ and } A \in D\big) \\
D;\, (\vec{A}, \{z.P \to M\})\langle U \rangle \to D;\, M[z := \vec{A}, \sigma] \ \text{ if } D \triangleright match(P)(U) = \sigma
\end{array}
$$

The first rule states that call is executed by searching $D$ for any advice triggered by the current event. All matching advice then advises the event. The second rule states that if there is advice advising the current event, then the current state is replaced by the body of the advice, with the appropriate substitutions. If the advice list is empty, or if the pointcut $P$ does not match the event $U$, then evaluation is *stuck*. The sorting system of the next section rules out stuck terms.

## 3. Sorting

To avoid getting stuck, two invariants must be preserved by evaluation:

1. if a piece of advice proceeds, there must be at least one piece of advice in the advice queue, and
2. if a piece of advice proceeds and it modifies the current event, it must be certain to do so in such a way that it still satisfies the pointcuts of any remaining advice in the advice queue.

**Example 3.** In the absence of any other advice declarations, the following $\mu$ABC program violates condition (1) above:

$$\mathsf{adv}\,\{z.\hat{f}, \hat{g} \to z\langle \hat{f}, \hat{g} \rangle\}\,;\ \mathsf{call}\langle \hat{f}, \hat{g} \rangle$$

The term evaluates to $\cdot\langle \hat{f}, \hat{g} \rangle$, which is stuck. Since there is no additional advice, the use of the proceed variable in the advice is malformed. □

**Example 4.** The following $\mu$ABC program violates condition (2) above:

$$
\begin{array}{l}
\mathsf{adv}\,\{z.\hat{f}, x{:}\mathsf{int} \to z\langle \hat{f}, 42 \rangle\}\,; \\
\mathsf{adv}\,\{z.\hat{f}, x{:}\mathsf{int} \to \mathsf{new}\ g\,;\, z\langle \hat{g} \rangle\}\,; \\
\mathsf{call}\langle \hat{f}, 10 \rangle.
\end{array}
$$

In this example, the current event is $\langle \hat{f}, 10 \rangle$, and after evaluation, there will be two pieces of advice queued up advising it. Since advice is queued in LIFO order, the second piece of advice will trigger first. It declares a new name $g$, changes the current event to the tuple $\langle \hat{g} \rangle$, and proceeds on the new event. The result is that the first piece of advice now advises the new event $\langle \hat{g} \rangle$, but its pointcut is no longer satisfied by the current event. □

In this section, we present a sorting system that guarantees progress and preservation.

The sort of an event is itself is given using the same syntax as events. To distinguish the two uses, we use $S$, $T$ for sorts and $U$, $V$ for events.

The sort of an event computes the bounds on inexact atoms in the expected way. For instance, if we have declared $\mathsf{new}\ a{:}s$ and $\mathsf{new}\ b{:}t$, then the event $\langle a, b, \hat{c} \rangle$ has sort $\langle s, t, \hat{c} \rangle$.

We sort advice based on its pointcut. For instance, the advice $\{f, x{:}a, y{:}b \to M\}$ has sort $\langle f, a, b \rangle$.

If an advice uses its proceed variable, we say that it is *nonfinal*; if it does not use its proceed variable, we say that it is *final*. Nonfinal advice of sort $S$ also has sort $S\ \mathsf{final}$. For example, $\{z.\hat{f}, x{:}\mathsf{int}, y{:}\mathsf{int} \to \mathsf{call}\langle \hat{g}, y, x \rangle\}$ can be given sorts $\langle \hat{f}, \mathsf{int}, \mathsf{int} \rangle$ and $\langle \hat{f}, \mathsf{int}, \mathsf{int} \rangle\ \mathsf{finalized}$; it ignores its proceed variable and hence any advice declared before. The advice $\{z.\hat{f}, x{:}\mathsf{int}, y{:}\mathsf{int} \to z\langle \hat{g}, y, x \rangle\}$, instead, can be given only sort $\langle \hat{f}, \mathsf{int}, \mathsf{int} \rangle$; this advice is *nonfinal*. If advice of sort $S$ final has been declared, we say that sort $S$ has been *finalized*.

We sort with respect to the *environment*, $\Gamma$, that keeps records the sorts of names and advice, as well as the sorts that have been finalized. These concepts are formalized below:

ENVIRONMENTS

$\Gamma, \Delta ::= \cdot \mid \Gamma, (a{:}s) \mid \Gamma, (z{:}S) \mid \Gamma, (S\ \mathsf{finalized})$

We require that all environments be well formed, in the sense that each name $a$ occur at most once on the lefthand side of a declaration $a{:}s$. Formally, the environment "$\Gamma, \Delta$" consisting of the union of $\Gamma$ and $\Delta$ is undefined if any name occurs in the domain of both $\Gamma$ and $\Delta$.

The sorting rules for pointcuts and events are as follows. Pointcuts produce an environment $\Delta$ which includes all the names bound by the pointcut.

SORTING $(\vdash P : S \triangleright \Delta)$ $(\Gamma \vdash U : S)$

$$
\begin{array}{ll}
\vdash (\cdot) : (\cdot) \triangleright (\cdot) & \\
\vdash (P, \hat{a}) : (T, \hat{a}) \triangleright (\Delta) & \text{if } \vdash P : T \triangleright \Delta \\
\vdash (P, x{:}s) : (T, s) \triangleright (\Delta, x{:}s) & \text{if } \vdash P : T \triangleright \Delta
\end{array}
$$

$$
\begin{array}{l}
\Gamma \vdash (\cdot) : (\cdot) \\
\Gamma \vdash (U, \hat{a}) : (S, \hat{a}) \ \text{ if } \Gamma \vdash U : S \\
\Gamma \vdash (U, a) : (S, s) \ \text{ if } \Gamma \vdash U : S \text{ and } \Gamma \ni a{:}s \\
\Gamma \vdash (U, a) : (S, s) \ \text{ if } \Gamma \vdash U : S \text{ and } \Gamma \ni a{:}t \text{ and } \Gamma \vdash t : s
\end{array}
$$

Similarly to pointcuts, the sorting of declarations extracts the sorts of the names declared in $D$, as well as the sorts have been finalized as a result of the advice declared in $D$. The judgement takes the form $\Gamma \vdash D \triangleright \Delta$.

SORTING $(\Gamma \vdash A : S)$ $(\Gamma \vdash A : S\ \mathsf{final})$ $(\Gamma \vdash D \triangleright \Delta)$ $(\Gamma \vdash M\ \mathsf{ok})$

$$
\begin{array}{ll}
\Gamma \vdash \{z.P \to M\} : S & \text{if } \vdash P : S \triangleright \Delta \text{ and } \Gamma, z{:}S, \Delta \vdash M\ \mathsf{ok} \\
\Gamma \vdash \{z.P \to M\} : S\ \mathsf{final} & \text{if } \vdash P : S \triangleright \Delta \text{ and } \Gamma, \Delta \vdash M\ \mathsf{ok}
\end{array}
$$

$$
\begin{array}{ll}
\Gamma \vdash \cdot \triangleright \cdot & \\
\Gamma \vdash D;\, \mathsf{new}\ a{:}s \triangleright \Delta, a{:}s & \text{if } \Gamma \vdash D \triangleright \Delta \text{ and } a \notin \Gamma, \Delta \\
\Gamma \vdash D;\, \mathsf{adv}\ A \triangleright \Delta, S\ \mathsf{finalized} & \text{if } \Gamma \vdash D \triangleright \Delta \text{ and } \Gamma, \Delta \vdash A : S\ \mathsf{final} \\
\Gamma \vdash D;\, \mathsf{adv}\ A \triangleright \Delta & \text{if } \Gamma \vdash D \triangleright \Delta \text{ and } \Gamma, \Delta \vdash A : S \\
 & \text{and } \Gamma \ni S\ \mathsf{finalized}
\end{array}
$$

$$\begin{array}{ll}
\Gamma \vdash \mathsf{call}\langle U\rangle\ \mathsf{ok} & \text{if } \Gamma \vdash U : S \text{ and } \Gamma \ni S\ \mathsf{finalized} \\
\Gamma \vdash \quad z\langle U\rangle\ \mathsf{ok} & \text{if } \Gamma \vdash U : S \text{ and } \Gamma \ni z{:}S \\
\Gamma \vdash \quad \vec{A}\langle U\rangle\ \mathsf{ok} & \text{if } \Gamma \vdash U : S \text{ and } \forall i.\, \Gamma \vdash A_i : S \text{ and } \exists i.\, \Gamma \vdash A_i : S\ \mathsf{final} \\
\Gamma \vdash \quad D\,;M\ \mathsf{ok} & \text{if } \Gamma \vdash D \triangleright \Delta \text{ and } \Gamma,\Delta \vdash M\ \mathsf{ok}
\end{array}$$

To see what it means for $D$ to be consistent with $\Gamma$, observe that in order to avoid error condition (1) above, the first declared advice for any given sort $S$ must be final— that is, it cannot proceed. If it were to proceed, it would be guaranteed that there would be no remaining advice, and a runtime error would result. Once a sort has been finalized, nonfinal advice of that sort may then be declared.

The proof of progress relies on the following properties:

- If $\vdash D \triangleright \Gamma$ and $\Gamma \vdash a : s$, then $D \triangleright a{:}s$.
- If $\vdash D \triangleright \Gamma$ and $\Gamma \vdash U : S$ and $\vdash P : S \triangleright \_$, then $D \triangleright \mathit{match}(P)(U)$.
- If $\vdash D \triangleright \Gamma$ and $\Gamma \ni S$ $\mathsf{finalized}$ and $\Gamma \vdash U : S$, then $D \ni \mathsf{adv}\,\{\_.P \to \_\}$ such that $D \triangleright \mathit{match}(P)(U)$.

**Theorem 5 (Progress).** *For any term $M$, if $\vdash M$ $\mathsf{ok}$, then $M \to M'$ for some $M'$.*

The proof of preservation relies on a substitution lemma, which in turn relies on *compatibility* between substitutions and typing environments.

**Definition 6 (Compatibility).** $\Gamma \vdash \sigma \sim \Delta$ if and only if $\mathit{dom}(\sigma) = \mathit{dom}(\Delta)$ and $\forall\, a \in \mathit{dom}(\sigma).\ \Gamma \vdash \sigma(a) : \Delta(a)$. □

For example, if $\sigma = [x := a, y := b]$ then $a{:}s, b{:}t \vdash \sigma \sim x{:}s, y{:}t$.

**Lemma 7 (Compatibility).** *If $\vdash D \triangleright \Gamma$ and $\Gamma \vdash U : S$ and $\vdash P : S \triangleright \Delta$ and $D \triangleright \mathit{match}(P)(U) = \sigma$, then $D \vdash \sigma \sim \Delta$.*

**Lemma 8 (Substitution).** *If $\Gamma, \Delta, z{:}S \vdash D; M$ $\mathsf{ok}$ and $\Gamma \vdash \sigma \sim \Delta$ and $\forall\, A \in \vec{A}.\ \Gamma \vdash A : S$, then $\Gamma \vdash D; M[\sigma, z := \vec{A}]$ $\mathsf{ok}$.*

**Theorem 9 (Preservation).** *If $\Gamma \vdash M$ $\mathsf{ok}$ and $M \to M'$ then $\Gamma \vdash M'$ $\mathsf{ok}$.*

## 4. A functional language with advice

In this section, we present an extension of the $\lambda$-calculus in which functions can be named and advised, in the style of [9, 8], and describe its translation into $\mu$ABC.

### 4.1 The source language

Due to space constraints, our presentation is necessarily brief. We give some examples of evaluation of the source language in Section 4.3; for further examples and narrative, see [8], which we follow closely. The language is very expressive. Although declarations are sequential, one can write mutually recursive functions using advice. As shown in [8], this language is also powerful enough to capture imperative features. For example, one can create a reference cell as function which accepts unit and returns the value of the cell; getting the stored value is achieved by calling the function; setting the value is achived by placing advice so that the function returns a different value in future calls.

We annotate each abstraction with its type to facilitate the translation presented in the following subsection. We often elide these annotations.

SYNTAX

$$\begin{array}{lll}
\mathrm{A,B} ::= \lambda x.\mathrm{M_T} & & \text{Abstractions} \\
\mathrm{D,E} ::= \mathsf{fun}\, f \mathsf{=} \mathrm{A} \mid \mathsf{adv}\,\{z.\hat{f} \to \mathrm{A}\} & & \text{Declarations} \\
\mathrm{V,U} ::= n \mid \mathsf{unit} \mid \mathrm{A} & & \text{Values} \\
\mathrm{M,N} ::= \mathrm{V} \mid \mathrm{V\,U} \mid z\,\mathrm{U} \mid \mathrm{D}\,;\mathrm{M} \mid \mathsf{let}\, x{:}\mathrm{T} \mathsf{=} \mathrm{M}\,; \mathrm{N} & & \text{Terms} \\
\mathrm{T,S} ::= \mathsf{Unit} \mid \mathrm{T{\to}S} & & \text{Types} \\
\Gamma,\Delta ::= \cdot \mid \Gamma, x{:}\mathrm{T} & & \text{Environments}
\end{array}$$

The language is simply typed; nevertheless non-termination is possible due to the imperative quality of aspects. The typing system need not be concerned with finality of advice, since only function names can be advised and functions themselves cannot proceed.

TYPING $(\Gamma \vDash \mathrm{A} : \mathrm{T})$ $(\Gamma \vDash \mathrm{D} \triangleright \Delta)$ $(\Gamma \vDash \mathrm{M} : \mathrm{T})$

$$\begin{array}{ll}
\Gamma \vDash \lambda x.\mathrm{M_{T\to S}} : \mathrm{T{\to}S} \ \text{if } \Gamma, x{:}\mathrm{T} \vDash \mathrm{M} : \mathrm{S} & \\
\Gamma \vDash \mathsf{fun}\, f \mathsf{=} \mathrm{A} \triangleright f{:}\mathrm{T} & \text{if } \Gamma, f{:}\mathrm{T} \vDash \mathrm{A} : \mathrm{T} \\
\Gamma \vDash \mathsf{adv}\,\{z.\hat{f} \to \mathrm{A}\} \triangleright \cdot & \text{if } \Gamma \vDash f : \mathrm{T} \text{ and } \Gamma, z{:}\mathrm{T} \vDash \mathrm{A} : \mathrm{T} \\
\Gamma \vDash \mathsf{unit} : \mathsf{Unit} & \\
\Gamma \vDash n : \mathrm{T} & \text{if } \Gamma \ni n{:}\mathrm{T} \\
\Gamma \vDash \mathrm{V\,U} : \mathrm{S} & \text{if } \Gamma \vDash \mathrm{V} : \mathrm{T{\to}S} \text{ and } \Gamma \vDash \mathrm{U} : \mathrm{T} \\
\Gamma \vDash z\,\mathrm{U} : \mathrm{S} & \text{if } \Gamma \vDash z : \mathrm{T{\to}S} \text{ and } \Gamma \vDash \mathrm{U} : \mathrm{T} \\
\Gamma \vDash \mathrm{D}\,; \mathrm{M} : \mathrm{T} & \text{if } \Gamma \vDash \mathrm{D} \triangleright \Delta \text{ and } \Gamma, \Delta \vDash \mathrm{M} : \mathrm{T} \\
\Gamma \vDash \mathsf{let}\, x{:}\mathrm{T} \mathsf{=} \mathrm{M}\,; \mathrm{N} : \mathrm{S} & \text{if } \Gamma \vDash \mathrm{M} : \mathrm{T} \text{ and } \Gamma, x{:}\mathrm{T} \vDash \mathrm{U} : \mathrm{S}
\end{array}$$

Evaluation is defined using *lookup*, notated $\vec{\mathrm{D}}(f) = \mathrm{A}$. Lookup resolves proceed variables, producing a single abstraction which includes the advice on the function in addition to the function body itself. Lookup is defined using the partial function *body* and total function *advise* with forms $\mathit{body}(\vec{\mathrm{D}})(f) = \mathrm{A}$ and $\mathit{advise}(\vec{\mathrm{D}})(f)(\mathrm{A}) = \mathrm{B}$.

EVALUATION $(\mathrm{M} \Rightarrow \mathrm{N})$

$$\begin{array}{ll}
\mathit{body}(\cdot)(f) = \text{undefined} & \\
\mathit{body}(\mathrm{D}\,; \vec{\mathrm{E}})(f) = \mathrm{A} & \text{if } \mathrm{D} = \mathsf{fun}\, f \mathsf{=} \mathrm{A} \\
\mathit{body}(\mathrm{D}\,; \vec{\mathrm{E}})(f) = \mathit{body}(\vec{\mathrm{E}})(f) & \text{otherwise} \\
\mathit{advise}(\cdot)(f)(\mathrm{A}) = \mathrm{A} & \\
\mathit{advise}(\mathrm{D}\,; \vec{\mathrm{E}})(f)(\mathrm{A}) = \mathit{advise}(\vec{\mathrm{E}})(f)(\mathrm{B}[z := \mathrm{A}]) & \text{if } \mathrm{D} = \mathsf{adv}\,\{z.\hat{f} \to \mathrm{B}\} \\
\mathit{advise}(\mathrm{D}\,; \vec{\mathrm{E}})(f)(\mathrm{A}) = \mathit{advise}(\vec{\mathrm{E}})(f)(\mathrm{A}) & \text{otherwise} \\
\vec{\mathrm{D}}(f) = \mathit{advise}(\vec{\mathrm{D}})(f)(\mathit{body}(\vec{\mathrm{D}})(f)) & \\
\vec{\mathrm{D}}\,; f\,\mathrm{V} \Rightarrow \vec{\mathrm{D}}\,; \mathrm{A\,V} & \text{if } \vec{\mathrm{D}}(f) = \mathrm{A} \\
\vec{\mathrm{D}}\,; (\lambda x.\mathrm{N})\,\mathrm{V} \Rightarrow \vec{\mathrm{D}}\,; \mathrm{N}[x := \mathrm{V}] & \\
\vec{\mathrm{D}}\,; \mathsf{let}\, x \mathsf{=} \mathrm{V}\,; \mathrm{N} \Rightarrow \vec{\mathrm{D}}\,; \mathrm{N}[x := \mathrm{V}] & \\
\vec{\mathrm{D}}\,; \mathsf{let}\, x \mathsf{=} \mathrm{M}\,; \mathrm{N} \Rightarrow \vec{\mathrm{D}}\,; \mathsf{let}\, x \mathsf{=} \mathrm{M}'\,; \mathrm{N} & \text{if } \vec{\mathrm{D}}\,; \mathrm{M} \Rightarrow \vec{\mathrm{D}}\,; \mathrm{M}'
\end{array}$$

### 4.2 The translation

Our translation of lambda into $\mu$ABC is based on the translation of lambda into pi, and thus is parameterized with respect to a continuation $k$. Proceed names must be handled specially, and thus the translation is also parameterized by a list of proceed names, paired with their pointcuts.

$$\vartheta \ ::= \ \cdot \ \mid \ \vartheta, z{:}f$$

In the translation, the value unit and both the names and types of the source language are treated as $\mu$ABC names. We also assume a name $\mathrm{T}^{-1}$ for every lambda type T, which is used for continuations. By convention, we use the names $k$, $j$, $i$ to stand for continuations.

TRANSLATION $((\!|\mathrm{V}|\!)^{\vartheta} = (D)(n))$ $([\![\mathrm{D}]\!]^{\vartheta} = D)$ $([\![\mathrm{M}]\!]^{\vartheta}_{k} = M)$

$$\begin{array}{l}
(\!|n|\!)^{\vartheta} \triangleq (\cdot)(n) \\
(\!|\mathsf{unit}|\!)^{\vartheta} \triangleq (\cdot)(\mathsf{unit}) \\
(\!|\lambda x.\mathrm{M_{T\to S}}|\!)^{\vartheta} \triangleq (\mathsf{new}\, f{:}\mathrm{T{\to}S}\,; \mathsf{adv}\,\{\hat{f}, x{:}\mathrm{T}, k{:}\mathrm{S}^{-1} \to [\![\mathrm{M}]\!]^{\vartheta}_{k}\})(f) \\
\qquad\qquad\qquad\quad \text{where } f \notin \mathit{fn}(\mathrm{M}) \\
[\![\mathsf{fun}\, f \mathsf{=} \lambda x.\mathrm{M_{T\to S}}]\!]^{\vartheta} \triangleq \mathsf{new}\, f{:}\mathrm{T{\to}S}\,; \mathsf{adv}\,\{\hat{f}, x{:}\mathrm{T}, k{:}\mathrm{S}^{-1} \to [\![\mathrm{M}]\!]^{\vartheta}_{k}\} \\
[\![\mathsf{adv}\,\{z.\hat{f} \to \lambda x.\mathrm{M_{T\to S}}\}]\!]^{\vartheta} \triangleq \mathsf{adv}\,\{z.\hat{f}, x{:}\mathrm{T}, k{:}\mathrm{S}^{-1} \to [\![\mathrm{M}]\!]^{\vartheta, z:f}_{k}\} \\
[\![\mathrm{V}]\!]^{\vartheta}_{k} \triangleq D\,; \mathsf{call}\langle \hat{k}, n\rangle \text{ where } (\!|\mathrm{V}|\!)^{\vartheta} = (D)(n)
\end{array}$$

$$\begin{aligned}
&\llbracket \mathrm{V\ U} \rrbracket_k^\vartheta \triangleq D\,;E\,;\mathsf{call}\langle \hat{g}, n, k\rangle \text{ where } g \notin dom(\vartheta)\\
&\qquad\qquad \text{and } (\!|\mathrm{V}|\!)^\vartheta = (D)(g) \text{ and } (\!|\mathrm{U}|\!)^\vartheta = (E)(n)\\
&\llbracket z\ \mathrm{U} \rrbracket_k^\vartheta \triangleq E\,; z\langle \hat{f}, n, k\rangle \text{ where } \vartheta(z) = f \text{ and } (\!|\mathrm{U}|\!)^\vartheta = (E)(n)\\
&\llbracket \mathsf{let}\ x{:}\mathrm{T} = \mathrm{M}\,;\mathrm{N} \rrbracket_k^\vartheta \triangleq \mathsf{new}\ j{:}\mathrm{T}^{-1}\,;\mathsf{adv}\,\{\hat{j}, x{:}\mathrm{T} \rightarrow \llbracket \mathrm{N} \rrbracket_k^\vartheta\}\,;\llbracket \mathrm{M} \rrbracket_j^\vartheta\\
&\qquad\qquad\qquad\qquad \text{where } j \notin fn(\mathrm{M})\\
&\llbracket \mathrm{D}\,;\mathrm{M} \rrbracket_k^\vartheta \triangleq \llbracket \mathrm{D} \rrbracket^\vartheta\,;\llbracket \mathrm{M} \rrbracket_k^\vartheta
\end{aligned}$$

## 4.3 Examples

**Example 10.** Consider the $\lambda$-calculus term $(\lambda x.x^2)\ 5$. It evaluates as: $(\lambda x.x^2)\ 5 \Rightarrow 25$. The translation of the $\lambda$-calculus term into $\mu$ABC with continuation $k$ is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, 25\rangle$.

$$\begin{aligned}
\llbracket (\lambda x.x^2)\ 5 \rrbracket_k = &\mathsf{new} f\,;\\
&\mathsf{adv}\,\{\hat{f}, x, j \rightarrow \mathsf{call}\langle \hat{j}, x^2\rangle\}\,;\\
&\mathsf{call}\langle \hat{f}, 5, k\rangle
\end{aligned}$$

The name $f$ represents the function $\lambda x.x^2$. The function is implemented using by the advice declaration $\mathsf{adv}\,\{\hat{f}, x, j \rightarrow \mathsf{call}\langle \hat{j}, x^2\rangle\}$. The function is applied to the argument 5 by calling function $f$ on argument 5 with continuation $k$.

First, $\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up the advice, which gets enqueued on the current event:

$$\rightarrow \{\hat{f}, x, j \rightarrow \mathsf{call}\langle \hat{j}, x^2\rangle\}\langle f, 5, k\rangle$$

The body of the advice executes with 5 bound to $x$, and $k$ bound to $j$:

$$\rightarrow \mathsf{call}\langle \hat{k}, 5^2\rangle \qquad \square$$

**Example 11.** Consider the $\lambda$-calculus term $\mathsf{let}\ x = 5\,; (\lambda y.y^2)x$. It evaluates as: $\mathsf{let}\ x = 5\,; (\lambda y.y^2)x \Rightarrow (\lambda y.y^2)5 \Rightarrow 25$. The translation of the $\lambda$-calculus term into $\mu$ABC with continuation $k$ is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, 25\rangle$.

$$\begin{aligned}
\llbracket \mathsf{let}\ x = 5\,; (\lambda y.y^2)x \rrbracket_k = &\mathsf{new}\ j\,;\\
&\mathsf{adv}\,\{\hat{j}, x \rightarrow \mathsf{new} f\,;\\
&\qquad\quad \mathsf{adv}\,\{\hat{f}, y, i \rightarrow \mathsf{call}\langle \hat{\imath}, y^2\rangle\}\,;\\
&\qquad\quad \mathsf{call}\langle \hat{f}, x, k\rangle\}\,;\\
&\mathsf{call}\langle \hat{j}, 5\rangle
\end{aligned}$$

First, $\mathsf{call}\langle \hat{j}, 5\rangle$ looks up the advice, which gets enqueued on the current event:

$$\rightarrow \{\hat{j}, x \rightarrow \mathsf{new} f\,; \mathsf{adv}\,\{\hat{f}, y, i \rightarrow \mathsf{call}\langle \hat{\imath}, y^2\rangle\}\,; \mathsf{call}\langle \hat{f}, x, k\rangle\}\langle \hat{j}, 5\rangle$$

The advice body then gets executed with 5 bound to $x$:

$$\rightarrow \mathsf{new} f\,; \mathsf{adv}\,\{\hat{f}, y, i \rightarrow \mathsf{call}\langle \hat{\imath}, y^2\rangle\}\,; \mathsf{call}\langle \hat{f}, 5, k\rangle$$

$\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up the advice, which gets enqueued on the current event:

$$\rightarrow \{\hat{f}, y, i \rightarrow \mathsf{call}\langle \hat{\imath}, y^2\rangle\}\langle \hat{f}, 5, k\rangle$$

The advice body then gets executed with 5 bound to $y$ and $k$ bound to $i$:

$$\rightarrow \mathsf{call}\langle \hat{k}, 5^2\rangle \qquad \square$$

**Example 12.** Consider the $\lambda$-calculus term $\mathsf{fun} f = \lambda y.y^2\,; \mathsf{let}\ x = 5\,; f\ x$. It evaluates as: $\mathsf{fun} f = \lambda y.y^2\,; \mathsf{let}\ x = 5\,; f\ x \Rightarrow \mathsf{let}\ x = 5\,; (\lambda y.y^2)x \Rightarrow (\lambda y.y^2)5 \Rightarrow 25$. The translation of the $\lambda$-calculus term into $\mu$ABC with continuation $k$ is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, 25\rangle$.

$$\begin{aligned}
\llbracket \mathsf{fun} f = \lambda y.y^2\,; \mathsf{let}\ x = 5\,; f\ x \rrbracket_k = &\mathsf{new} f\\
&\mathsf{adv}\,\{\hat{f}, y, j \rightarrow \mathsf{call}\langle \hat{j}, y^2\rangle\}\\
&\mathsf{new}\ i\,;\\
&\mathsf{adv}\,\{\hat{\imath}, x \rightarrow \mathsf{call}\langle \hat{f}, x, k\rangle\}\,;\\
&\mathsf{call}\langle \hat{\imath}, 5\rangle
\end{aligned}$$

First, $\mathsf{call}\langle \hat{\imath}, 5\rangle$ looks up the advice, which gets enqueued on the current event:

$$\rightarrow \{\hat{\imath}, x \rightarrow \mathsf{call}\langle \hat{f}, x, k\rangle\}\langle \hat{\imath}, 5\rangle$$

The advice body is executed with 5 bound to $x$:

$$\rightarrow \mathsf{call}\langle \hat{f}, 5, k\rangle$$

$\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up advice, which gets enqueued on the current event:

$$\rightarrow \{\hat{f}, y, j \rightarrow \mathsf{call}\langle \hat{j}, y^2\rangle\}\langle \hat{f}, 5, k\rangle$$

The advice body is executed with 5 bound to $y$ and $k$ bound to $j$:

$$\rightarrow \mathsf{call}\langle \hat{k}, 5^2\rangle \qquad \square$$

**Example 13.** Consider the $\lambda$-calculus term

$$\begin{aligned}
\mathrm{M} = &\mathsf{fun} f = \lambda y.y^2\,;\\
&\mathsf{adv}\,\{z.\hat{f} \rightarrow \lambda x.z(x+1)\}\,;\\
&f\ 5
\end{aligned}$$

It evaluates as: $\mathrm{M} \Rightarrow (\lambda x.(\lambda y.y^2)(x+1))5 \Rightarrow (\lambda y.y^2)(5+1) \Rightarrow (5+1)^2$. The translation of the $\lambda$-calculus term into $\mu$ABC with continuation $k$ is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, (5+1)^2\rangle$.

$$\begin{aligned}
\llbracket \mathrm{M} \rrbracket_k = &\mathsf{new} f\,;\\
&\mathsf{adv}\,\{\hat{f}, y, k \rightarrow \mathsf{call}\langle \hat{k}, y^2\rangle\}\,;\\
&\mathsf{adv}\,\{z.\hat{f}, x, k \rightarrow z\langle \hat{f}, x+1, k\rangle\}\,;\\
&\mathsf{call}\langle \hat{f}, 5, k\rangle
\end{aligned}$$

First, $\mathsf{call}\langle \hat{f}, 5, k\rangle$ looks up the two pieces of advice and enqueues them on the current event:

$$\rightarrow \{\hat{f}, y, k \rightarrow \mathsf{call}\langle \hat{k}, y^2\rangle\}, \{z.\hat{f}, x, k \rightarrow z\langle \hat{f}, x+1, k\rangle\}\langle \hat{f}, 5, k\rangle$$

The body of the newest advice executes, with 5 bound to $x$, $k$ bound to $k$, and the remaining advice bound to $z$:

$$\rightarrow \{\hat{f}, y, k \rightarrow \mathsf{call}\langle \hat{k}, y^2\rangle\}\langle \hat{f}, 5+1, k\rangle$$

The body of the advice now executes, with $5+1$ bound to $y$ and $k$ bound to $k$:

$$\rightarrow \mathsf{call}\langle \hat{k}, 36\rangle \qquad \square$$

**Example 14.** Consider the $\lambda$-calculus term

$$\begin{aligned}
\mathrm{M} = &\mathsf{fun} f = \lambda y.y^2\,;\\
&\mathsf{adv}\,\{z.\hat{f} \rightarrow \lambda y_1.z(y_1+1)\}\,;\\
&\mathsf{adv}\,\{z.\hat{f} \rightarrow \lambda y_2.\mathsf{let}\ x = z(y_2)\,; z(x)\}\,;\\
&f\ 5
\end{aligned}$$

Let $\mathrm{A} = (\lambda y_1.(\lambda y.y^2)(y_1+1))$. It evaluates as:

$$\begin{aligned}
\mathrm{M} &\Rightarrow (\lambda y_2.\mathsf{let}\ x = \mathrm{A}\ y_2\,; \mathrm{A}\ x)\ 5\\
&\Rightarrow \mathsf{let}\ x = \mathrm{A}\ 5\,; \mathrm{A}\ x\\
&\Rightarrow^* \mathsf{let}\ x = (5+1)^2\,; \mathrm{A}\ x\\
&\Rightarrow \mathrm{A}(5+1)^2\\
&\Rightarrow^* ((5+1)^2+1)^2
\end{aligned}$$

The translation of the $\lambda$-calculus term into $\mu$ABC with continuation $k$ is shown below; we show how it evaluates to $\mathsf{call}\langle \hat{k}, ((5+$

$1)^2+1)^2\rangle$.

$$
\begin{aligned}
&[\![\mathrm{M}]\!]_k = D; \mathsf{call}\langle\hat{f},5,k\rangle\\
&\quad D = \mathsf{new}\, f\,; \mathsf{adv}\, A\,; \mathsf{adv}\, B\,; \mathsf{adv}\, C\,;\\
&\quad A = \{\hat{f},y,k \to \mathsf{call}\langle\hat{k},y^2\rangle\}\\
&\quad B = \{z.\hat{f},y_1,k \to z\langle\hat{f},y_1+1,k\rangle\}\\
&\quad C = \{z.\hat{f},y_2,k \to \mathsf{new}\ j\,; \mathsf{adv}\,\{\hat{j},x \to z\langle\hat{f},x,k\rangle\}\,; z\langle\hat{f},y_2,j\rangle\}
\end{aligned}
$$

The $\mathsf{call}\langle\hat{f},5,k\rangle$ triggers advice lookup, and enqueues the three matching pieces of advice onto the current event:

$$[\![\mathrm{M}]\!]_k \to D;\, (A,B,C)\langle\hat{f},5,k\rangle$$

The newest advice body is executed with 5 bound to $y_2$, $k$ bound to $j$, and the remaining advice bound to $z$:

$$\to E\,;\, (A,B)\langle\hat{f},5,j\rangle$$

where

$$E = D;\, \mathsf{new}\ j\,; \mathsf{adv}\,\{\hat{j},x \to (A,B)\langle\hat{f},x,k\rangle\}$$

The newest advice body is executed with 5 bound to $y_1$, $j$ bound to $k$, and the remaining advice bound to $z$:

$$\to E\,; A\langle\hat{f},5+1,j\rangle$$

The remaining advice body is executed with $5+1$ bound to $y$ and $j$ bound to $k$:

$$\to E\,; \mathsf{call}\langle\hat{j},(5+1)^2\rangle$$

The $\mathsf{call}\langle\hat{j},(5+1)^2\rangle$ triggers another advice lookup, and enqueues the matching advice onto the current event:

$$\to E\,; \{\hat{j},x \to (A,B)\langle f,x,k\rangle\}\langle\hat{j},(5+1)^2\rangle$$

The advice body is executed with $(5+1)^2$ bound to $x$:

$$\to E\,;\, (A,B)\langle\hat{f},(5+1)^2,k\rangle$$

The newest advice body is executed with $(5+1)^2$ bound to $y_1$ and $k$ bound to $k$:

$$\to E\,;\, (A)\langle\hat{f},(5+1)^2+1,k\rangle$$

The remaining advice body is executed with $(5+1)^2+1$ bound to $y$ and $k$ bound to $k$:

$$\to E\,; \mathsf{call}\langle\hat{k},((5+1)^2+1)^2\rangle \qquad \square$$

## 4.4 Correctness

Our correctness proof is stated modulo a "structural congruence" on $\mu$ABC terms. Most of the axioms defining this congruence are innocuous, but the last, *unrolling*, is stated in terms of the translation defined above. As stated in the introduction, there is a further problem in this approach: the congruence is not justified by any semantic reasoning. Nonetheless, our intention is to define this relation such that two structurally equivalent terms in effect "behave the same way".

We provide short examples demonstrating each of the structural equivalence rules, followed by a formal statement of the rules.

**Example 15 (Hoisting).** Hoisting enables us to move declarations out of the body of an advice declaration, provided that none of the variables in the declarations are bound in the advice body. For instance, the $\mu$ABC term

$$
\begin{aligned}
&\mathsf{new}\, f\,;\\
&\mathsf{adv}\,\{\hat{f},y,k \to \mathsf{new}\ g\,; \mathsf{adv}\,\{\hat{g},x,j \to \mathsf{call}\langle\hat{g},y+1,k\rangle\}\,; \mathsf{call}\langle\hat{g},y,k\rangle\}\\
&\mathsf{call}\langle\hat{f},10,k\rangle
\end{aligned}
$$

is structurally equivalent to

$$
\begin{aligned}
&\mathsf{new}\ g\,; \mathsf{adv}\,\{\hat{g},x,j \to \mathsf{call}\langle\hat{g},y+1,k\rangle\}\,;\\
&\mathsf{new}\, f\,; \mathsf{adv}\,\{\hat{f},y,k \to \mathsf{call}\langle\hat{g},y,k\rangle\}\\
&\mathsf{call}\langle\hat{f},10,k\rangle
\end{aligned}
$$

"Hoisting" $g$'s name and advice declaration out of $f$'s advice declaration should have no effect on how the term evaluates. $\square$

**Example 16 (Reordering).** Reordering says that two declarations $D$ and $E$ can be swapped, so long as $fn(E) \notin bn(D)$, and vice versa. For instance:

$$
\begin{array}{lcl}
\mathsf{new}\, f\,; & & \mathsf{new}\ g\,;\\
\mathsf{adv}\,\{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\,; & & \mathsf{adv}\,\{\hat{g},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\,;\\
\mathsf{new}\ g\,; & \equiv & \mathsf{new}\, f\,;\\
\mathsf{adv}\,\{\hat{g},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\,; & & \mathsf{adv}\,\{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\,;\\
\mathsf{call}\langle\hat{f},10,k\rangle & & \mathsf{call}\langle\hat{f},10,k\rangle
\end{array}
$$

Whether $f$'s name and advice is declared before or after $g$'s is irrelevant to how the term evaluates. The following, however, is not allowed, however, since $\mathsf{new}\, f$ must be declared before $f$ can appear in an advice declaration:

$$
\begin{array}{lcl}
\mathsf{new}\, f\,; & & \mathsf{adv}\,\{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\,;\\
\mathsf{adv}\,\{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\,; & \not\equiv & \mathsf{new}\, f\,;\\
\mathsf{call}\langle\hat{f},10,k\rangle & & \mathsf{call}\langle\hat{f},10,k\rangle
\end{array}
\qquad \square
$$

**Example 17 (Garbage Collection).** Garbage collection allows us to eliminate "dead" declarations. For instance, in the following term:

$$
\begin{aligned}
&\mathsf{new}\, f\,; \mathsf{adv}\,\{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x^2\rangle\}\,;\\
&\mathsf{new}\ g\,; \mathsf{adv}\,\{\hat{g},x,k \to \mathsf{call}\langle\hat{k},x+1\rangle\}\,;\\
&\mathsf{call}\langle\hat{f},10,k\rangle
\end{aligned}
$$

The two declarations $\mathsf{new}\ g$ and $\mathsf{adv}\,\{\hat{g},x,k \to \mathsf{call}\langle\hat{k},x+1\rangle\}$ are never used, and as such, can be eliminated without affecting how the term evaluates. Thus the above term is structurally equivalent to

$$\mathsf{new}\, f\,; \mathsf{adv}\,\{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x^2\rangle\}\,; \mathsf{call}\langle\hat{f},10,k\rangle \qquad \square$$

**Example 18 (Unrolling).** Translating function applications from $\lambda$-calculus into $\mu$ABC is extremely intricate. For instance, consider the $\lambda$-calculus term

$$
\begin{aligned}
&\mathsf{fun}\, f = \lambda x.x\,;\\
&\mathsf{adv}\,\{z_1.\hat{f} \to \lambda y.z_1\langle y^2\rangle\}\,;\\
&\mathsf{adv}\,\{z_2.\hat{f} \to \lambda w.z_2\langle w+1\rangle\}\,;\\
&f\ 5
\end{aligned}
$$

The translation of this term, with continuation $k$, is

$$D;\, \mathsf{call}\langle\hat{f},5,k\rangle$$

where

$$
\begin{aligned}
D &= \mathsf{new}\, f\,; \mathsf{adv}\, A\,; \mathsf{adv}\, B\,; \mathsf{adv}\, C\\
A &= \{\hat{f},x,k \to \mathsf{call}\langle\hat{k},x\rangle\}\\
B &= \{z_1.\hat{f},y,k \to z_1\langle f,y^2,k\rangle\}\\
C &= \{z_2.\hat{f},w,k \to z_2\langle f,w+1,k\rangle\}
\end{aligned}
$$

The $\lambda$-calculus term evaluates to

$$(\lambda w.(\lambda y.(\lambda x.x)\ y^2)\ (w+1))\ 5$$

the translation of which is

$$
\begin{aligned}
&\mathsf{new}\ h\,;\\
&\mathsf{adv}\,\{\hat{h},w,i \to \mathsf{new}\, f\,;\\
&\qquad\qquad \mathsf{adv}\,\{\hat{f},y,k \to \mathsf{new}\ g\,;\\
&\qquad\qquad\qquad\qquad \mathsf{adv}\,\{\hat{g},x,j \to \mathsf{call}\langle\hat{j},x\rangle\}\,;\\
&\qquad\qquad\qquad\qquad \mathsf{call}\langle\hat{g},y^2,k\rangle\}\,;\\
&\qquad\qquad \mathsf{call}\langle\hat{f},w+1,i\rangle\}\,;\\
&\mathsf{call}\langle\hat{h},5,k\rangle
\end{aligned}
$$

The fundamental difficulty arises from the fact that the translation of the original $\lambda$-calculus term yields a string of corresponding advice declarations, while the translation of the $\lambda$-calculus term after one step yields a single nested advice declaration. In this vein, unrolling allows us to expand the rolled form into the following structurally equivalent term:

$$D\,;\,(A,B,C)\langle\hat{f},5,k\rangle \qquad \square$$

STRUCTURAL EQUIVALENCE $(M \equiv N)$

*Hoisting:*

$D\,;\,\mathsf{adv}\,\{z.P \to E\,;\,M\} \equiv D\,;\,E\,;\,\mathsf{adv}\,\{z.P \to M\} \quad$ if $z, bn(P) \notin fn(E)$

*Reordering:*

$D\,;\,E\,;\,M \equiv E\,;\,D\,;\,M \quad$ if $fn(E) \notin bn(D)$ and $fn(D) \notin bn(E)$

*Garbage Collection:*

$D\,;\,\mathsf{new}\,f\,;\,\mathsf{adv}\,\{z.P,\hat{f},Q \to M\}\,;\,N \equiv D\,;\,N$ if $f \notin fn(N)$

*Unrolling:*

$$\begin{aligned} D\,;\,\mathsf{call}\langle\hat{f},v,k\rangle \equiv D\,;\,(&\{z_0.\hat{f},x_0,k_0 \to [\![\mathrm{L}_0]\!]_{k_0}\},\\ &\vdots\\ &\{z_m.\hat{f},x_m,k_m \to [\![\mathrm{L}_m]\!]_{k_m}\})\langle\hat{f},v,k)\rangle \end{aligned}$$

where $D = \mathsf{new}\,f\,;$

$\qquad \mathsf{adv}\,\{z_m.\hat{f},x_m,k \to [\![\mathrm{L}_m\sigma_m]\!]_k\}\,;$

and $\sigma_0 = [\,]$ and $\sigma_n = [z_n := \lambda y_{n-1}.\mathrm{L}_{n-1}\sigma_{n-1}]$

The following theorem states that our translation from $\lambda$-calculus into $\mu$ABC is correct up to structural congruence.

**Lemma 19 (Substitution).**

$$[\![\mathrm{M}[x := \mathrm{V}]]\!]_k^{\vartheta} \equiv D;\,[\![\mathrm{M}]\!]_k^{\vartheta}[x := v]$$
$$(\!|\mathrm{U}[x := \mathrm{V}]|\!)^{\vartheta} \equiv (D;\,E[x := v])(u)$$

*where* $(\!|\mathrm{V}|\!)^{\vartheta} = (D)(v)$ *and* $(\!|\mathrm{U}|\!)^{\vartheta} = (E)(u)$ $\square$

**Proposition 20.** *If* $\mathrm{M} \Rightarrow \mathrm{N}$, *then* $[\![\mathrm{M}]\!]_c \to^{*}\equiv [\![\mathrm{N}]\!]_c$.

## 5. A Small Object Language

We give a translation into $\mu$ABC of a small, object-oriented language with advice.

The source language is based roughly on that of [3, 7], but there are a few differences. First, the evaluation strategy is based on the evaluation strategy for lambda calculus presented in the last section; in particular the definition of lookup. Secondly, we ignore fields for simplicity.

As before, $z$ ranges over proceed names. In addition, we use the following conventions for names.

- $k$, $j$, $i$ range over continuations,
- $\ell$, $m$, $n$ range over method names,
- $a$, $b$, $e$ range over class names,
- $p$, $q$, $x$, $y$, $v$, $u$ range over object names,
  - $p$, $q$ range over proper object names,
  - $x$, $y$ range over variable names,
  - $v$, $u$ range over variables or proper object names,

OBJECT CALCULUS

| | | |
|---|---|---|
| $\mathbf{A},\mathbf{B}$ | $::= \lambda\vec{x}.\mathbf{M}_{\mathbf{T}}$ | Abstractions |
| $\mathbf{C}$ | $::= \mathsf{cls}\,a\!:\!b\,\{\bar{\ell}=\bar{\mathbf{A}}\}$ | Class Declarations |
| $\mathbf{D},\mathbf{E}$ | $::= \mathsf{obj}\;p\!:\!a \mid \mathsf{advc}\{z.a.\hat{\ell} \to \mathbf{A}\}$ | |
| $\mathbf{M},\mathbf{N}$ | $::= v \mid v.\ell(\vec{u}) \mid z(\vec{u}) \mid \mathbf{A}(\vec{u}) \mid \mathbf{D}\,;\,\mathbf{M} \mid \mathsf{let}\,x\!:\!a=\mathbf{M}\,;\,\mathbf{N}$ | |
| $\mathbf{T},\mathbf{S}$ | $::= \vec{a}\to b$ | Method Types |
| $\Gamma,\Delta$ | $::= \cdot \mid \Gamma, x\!:\!a$ | Environments |

As usual [6], we fix a class table $\bar{\mathbf{C}}$; we assume that $\mathsf{Object}$ is not declared and that the induced subclass relation is antisymmetric, with greatest element $\mathsf{Object}$. (The subclass relation is the smallest preorder on class names induced by the rule: $a \le b$ if $\bar{\mathbf{C}} \ni \mathsf{cls}\,a\!:\!b\,\{\cdots\}$.) We also assume that every declaration in the class table is well typed (ie, $\forall\mathbf{C} \in \bar{\mathbf{C}}.\ \Vdash \mathbf{C}\ \mathsf{ok}$).

The function *body* is used in both evaluation and typing. We leave out irrelevant bits.

$(body(a.\ell) = \mathbf{A}\!:\!\mathbf{T})$

$$\begin{aligned} body(a.\ell_i) &= \mathbf{A}_i\!:\!\mathbf{T}_i \;\text{ if }\; \bar{\mathbf{C}}(a) = \mathsf{cls}\,a\!:\!b\,\{\bar{\ell}=\bar{\mathbf{A}}\} \;\text{ and }\; \mathbf{A}_i = \lambda\vec{x}.\mathbf{M}_{\mathbf{T}_i}\\ body(a.\ell) &= \mathbf{A}\!:\!\mathbf{T} \;\;\text{ if }\; \bar{\mathbf{C}}(a) = \mathsf{cls}\,a\!:\!b\,\{\bar{m}=\bar{\mathbf{B}}\} \;\text{ and }\; \ell \notin \bar{m}\\ &\qquad\qquad\;\;\text{ and }\; body(b.\ell) = \mathbf{A}\!:\!\mathbf{T} \end{aligned}$$

TYPING $(\Gamma \Vdash \mathbf{A} : \mathbf{T})$ $(\Vdash \mathbf{C}\ \mathsf{ok})$ $(\Gamma \Vdash \mathbf{D} \triangleright \Gamma')$ $(\Gamma \Vdash \mathbf{M} : a)$

$\Gamma \Vdash \lambda\vec{x}.\mathbf{M}_{\vec{a}\to b} : \vec{a}\to b$ if $\Gamma,\vec{x}\!:\!\vec{a} \Vdash \mathbf{M} : b'$ and $b' \le b$

$\Vdash \mathsf{cls}\,a\!:\!b\,\{\bar{\ell}=\bar{\mathbf{A}}\}\ \mathsf{ok}$ if $\forall i.\ \mathsf{self}\!:\!a \Vdash \mathbf{A}_i : \mathbf{T}_i$
and $\forall i.\ body(b.\ell_i) = \mathbf{S}$ implies $\mathbf{T}_i = \mathbf{S}$

$$\begin{aligned} &\Gamma \Vdash \mathsf{obj}\;p\!:\!a \triangleright p\!:\!a && \text{if } \bar{\mathbf{C}}(a) \text{ defined}\\ &\Gamma \Vdash \mathsf{advc}\{z.a.\hat{\ell} \to \mathbf{A}\} \triangleright \cdot && \text{if } body(a.\ell) = \mathbf{T} = \_\overset{\mathsf{c}}{\to}\_\\ & && \text{and } \Gamma,\mathsf{self}\!:\!a,z\!:\!\mathbf{T} \Vdash \mathbf{A} : \mathbf{T} \end{aligned}$$

$$\begin{aligned} &\Gamma \Vdash v : a && \text{if } \Gamma \ni v\!:\!a\\ &\Gamma \Vdash v.\ell(\vec{u}) : b && \text{if } \Gamma \Vdash v : e \text{ and } body(e.\ell) = \vec{a}'\to b\\ & && \text{and } \forall i.\,\Gamma \Vdash u_i : a_i \text{ and } a_i \le a'_i\\ &\Gamma \Vdash z(\vec{u}) : b && \text{if } \Gamma \Vdash z : \vec{a}'\to b\\ & && \text{and } \forall i.\,\Gamma \Vdash u_i : a_i \text{ and } a_i \le a'_i\\ &\Gamma \Vdash \mathbf{A}(\vec{u}) : b && \text{if } \Gamma \Vdash \mathbf{A} : \vec{a}'\to b\\ & && \text{and } \forall i.\,\Gamma \Vdash u_i : a_i \text{ and } a_i \le a'_i\\ &\Gamma \Vdash \mathbf{D}\,;\,\mathbf{M} : a && \text{if } \Gamma \Vdash \mathbf{D} \triangleright \Delta \text{ and } \Gamma,\Delta \Vdash \mathbf{M} : a\\ &\Gamma \Vdash \mathsf{let}\,x\!:\!a=\mathbf{M}\,;\,\mathbf{N} : \mathbf{S} && \text{if } \Gamma \Vdash \mathbf{M} : a' \text{ and } a' \le a\\ & && \text{and } \Gamma,x\!:\!\mathbf{T} \Vdash \mathbf{N} : \mathbf{S} \end{aligned}$$

The functions for advising and lookup now have have the form $advise(\vec{\mathbf{D}})(p\!:\!a.\ell)(\mathbf{A}) = \mathbf{B}$ and $\vec{\mathbf{D}}(p.\ell) = \mathbf{A}$.

EVALUATION $(\vec{\mathbf{D}}\,;\,\mathbf{M} \Rightarrow \vec{\mathbf{E}}\,;\,\mathbf{N})$

$$advise(\cdot)(p\!:\!a.\ell)(\mathbf{A}) = \mathbf{A}$$

$$advise(\mathbf{D}\,;\,\vec{\mathbf{E}})(p\!:\!a.\ell)(\mathbf{A}) = \begin{cases} advise(\vec{\mathbf{E}})(p\!:\!a.\ell)(\mathbf{B}[z := \mathbf{A}]) & \text{if } \mathbf{D} = \mathsf{advc}\{z.a'.\hat{\ell} \to \mathbf{B}\} \text{ and } a \le a'\\ advise(\vec{\mathbf{E}})(p\!:\!a.\ell)(\mathbf{A}) & \text{otherwise} \end{cases}$$

$$\vec{\mathbf{D}}(p.\ell) = advise(\vec{\mathbf{D}})(p\!:\!a.\ell)(body(a.\ell)) \text{ if } \vec{\mathbf{D}} \ni p\!:\!a$$

$$\begin{aligned} \vec{\mathbf{D}}\,;\,p.\ell(\vec{q}) &\Rightarrow \vec{\mathbf{D}}\,;\,(\mathbf{A}[\mathsf{self} := p])(\vec{q}) \;\text{ if }\; \vec{\mathbf{D}}(p.\ell) = \mathbf{A}\\ \vec{\mathbf{D}}\,;\,(\lambda x.\mathbf{N})(\vec{q}) &\Rightarrow \vec{\mathbf{D}}\,;\,\mathbf{N}[x := \vec{q}]\\ \vec{\mathbf{D}}\,;\,\mathsf{let}\,x=p\,;\,\mathbf{N} &\Rightarrow \vec{\mathbf{D}}\,;\,\mathbf{N}[x := p]\\ \vec{\mathbf{D}}\,;\,\mathsf{let}\,x=\mathbf{M}\,;\,\mathbf{N} &\Rightarrow \vec{\mathbf{D}}\,;\,\mathsf{let}\,x=\mathbf{M}'\,;\,\mathbf{N} \quad \text{if } \vec{\mathbf{D}}\,;\,\mathbf{M} \Rightarrow \vec{\mathbf{D}}\,;\,\mathbf{M}' \end{aligned}$$

As before, the translation of terms is parameterized with respect to continuations and bound proceed names. In this case proceed names are bound to pointcuts of a different shape than in the functional case.

$$\vartheta \;::=\; \cdot \;\mid\; \vartheta, z\!:\!\langle p,\ell\rangle$$

TRANSLATION $([\![\mathbf{C}]\!] = D)$ $([\![\ell=\mathbf{A}]\!]^a = D)$

$$[\![\mathsf{cls}\,a\!:\!b\,\{\bar{\ell}=\bar{\mathbf{A}}\}]\!] \triangleq \mathsf{new}\;a\!:\!b\,;\,\mathsf{new}\;\vec{m}\,;\,[\![\bar{\ell}=\bar{\mathbf{A}}]\!]^a$$
$$\text{where } \vec{m} = (\ell_i \mid body(b.\ell_i) \text{ undefined})$$

$$[\![\ell=\lambda\vec{x}.\mathbf{M}_{\vec{a}\to b}]\!]^e \triangleq \mathsf{adv}\,\{\mathsf{self}\!:\!e,\hat{\ell},\vec{x}\!:\!\vec{a},k\!:\!b^{-1} \to [\![\mathbf{M}]\!]_k^{\vartheta}\}$$

TRANSLATION $(\llparenthesis \mathbf{A} \rrparenthesis^{\vartheta} = (D)(f))$ $(\llbracket \mathbf{D} \rrbracket^{\vartheta} = D)$ $(\llbracket \mathbf{M} \rrbracket_k^{\vartheta} = M)$

$$\begin{aligned}
\llparenthesis \lambda\vec{x}.\mathbf{M}_{\vec{a}\to b} \rrparenthesis^{\vartheta} &\triangleq (\mathsf{new}\, f : \vec{a}\to b\,;\ \mathsf{adv}\,\{\hat{f}, \vec{x} : \vec{a}, k : b^{-1} \to \llbracket \mathbf{M} \rrbracket_k^{\vartheta}\})(f) \\
&\quad \text{where}\, f \notin fn(\mathbf{M}) \\
\llbracket \mathsf{obj}\ p : a \rrbracket^{\vartheta} &\triangleq \mathsf{new}\ p : a \\
\llbracket \mathsf{advc}\{z.e.\hat{\ell} \to \lambda\vec{x}.\mathbf{M}_{\vec{a}\to b}\} \rrbracket^{\vartheta} &\triangleq \mathsf{adv}\,\{z.\mathsf{self} : e, \hat{\ell}, \vec{x} : \vec{a}, k : b^{-1} \to \llbracket \mathbf{M} \rrbracket_k^{\vartheta, z:\langle \mathsf{self}, \ell \rangle}\} \\
\llbracket v \rrbracket_k^{\vartheta} &\triangleq \mathsf{call}\langle \hat{k}, v \rangle \\
\llbracket v.\ell(\vec{u}) \rrbracket_k^{\vartheta} &\triangleq \mathsf{call}\langle v, \hat{\ell}, \vec{u}, k \rangle \\
\llbracket z(\vec{u}) \rrbracket_k^{\vartheta} &\triangleq z\langle p, \hat{\ell}, \vec{u}, k \rangle \text{ where } \vartheta(z) = \langle p, \ell \rangle \\
\llbracket \mathbf{A}(\vec{u}) \rrbracket_k^{\vartheta} &\triangleq D\,;\ \mathsf{call}\langle \hat{g}, \vec{u}, k \rangle \text{ where } \llparenthesis \mathbf{A} \rrparenthesis^{\vartheta} = (D)(g) \\
\llbracket \mathbf{D}\,;\mathbf{M} \rrbracket_k^{\vartheta} \triangleq \llbracket \mathbf{D}\,;\mathbf{M} \rrbracket_k^{\vartheta} &\triangleq \llbracket \mathbf{D} \rrbracket^{\vartheta}\,;\ \llbracket \mathbf{M} \rrbracket_k^{\vartheta} \\
\llbracket \mathsf{let}\, x : a = \mathbf{M}\,;\mathbf{N} \rrbracket_k^{\vartheta} &\triangleq \mathsf{new}\ j : a^{-1}\,;\ \mathsf{adv}\,\{\hat{j}, x : a \to \llbracket \mathbf{N} \rrbracket_k^{\vartheta}\}\,;\ \llbracket \mathbf{M} \rrbracket_j^{\vartheta}
\end{aligned}$$

We begin with a simple example.

**Example 21 (Methods).** Consider the following program fragment in the class-based language.

$$\mathsf{cls}\ a\,\{\ell = \lambda x.x^2\}\,;\ \mathsf{obj}\ p : a\,;\ p.\ell(5)$$

The class-based term evaluates as follows:

$$\begin{aligned}
&\Rightarrow \mathsf{cls}\ a\,\{\ell = \lambda x.x^2\}\,;\ \mathsf{obj}\ p : a\,;\ (\lambda x.x^2)\ 5 \\
&\Rightarrow \mathsf{cls}\ a\,\{\ell = \lambda x.x^2\}\,;\ \mathsf{obj}\ p : a\,;\ 25
\end{aligned}$$

The translation of the original term into $\mu$ABC yields:

$$\begin{aligned}
&\mathsf{new}\ a\,;\ \mathsf{adv}\,\{\mathsf{self} : a, \hat{\ell}, x, k \to \mathsf{call}\langle \hat{k}, x^2 \rangle\}\,; \\
&\mathsf{new}\ p : a\,;\ \mathsf{call}\langle p, \hat{\ell}, 5, k \rangle\,;
\end{aligned}$$

Observe that the $\mu$ABC term evaluates to $\mathsf{call}\langle \hat{k}, 25 \rangle$:

$$\begin{aligned}
&\to \mathsf{new}\ a\,;\ \mathsf{adv}\,\{\mathsf{self} : a, \hat{\ell}, x, k \to \mathsf{call}\langle \hat{k}, x^2 \rangle\}\,; \\
&\quad \mathsf{new}\ p : a\,;\ \{\mathsf{self} : a, \hat{\ell}, x, k \to \mathsf{call}\langle \hat{k}, x^2 \rangle\}\langle p, \hat{\ell}, 5, k \rangle \\
&\to \mathsf{new}\ a\,;\ \mathsf{adv}\,\{\mathsf{self} : a, \hat{\ell}, x, k \to \mathsf{call}\langle \hat{k}, x^2 \rangle\}\,; \\
&\quad \mathsf{new}\ p : a\,;\ \mathsf{call}\langle \hat{k}, 5^2 \rangle
\end{aligned}$$ □

**Example 22 (Methods).** Consider the following program fragment in the class-based language.

$$\begin{aligned}
&\mathsf{cls}\ a\,\{\ell = \lambda\,.M\,;\ m = \lambda\,.N\}\,; \\
&\mathsf{cls}\ b : a\,\{m = \lambda\,.P\,;\ n = \lambda\,.Q\}\,; \\
&\mathsf{obj}\ p : b\,; \\
&\mathsf{let}\ x = p.\ell()\,;\ \mathsf{let}\ y = p.m()\,;\ \mathsf{let}\ z = p.n()\,;\ U
\end{aligned}$$

Its $\mu$ABC translation is as follows:

$$\begin{aligned}
&\mathsf{new}\ a\,;\ \mathsf{adv}\,\{\mathsf{self} : a, \hat{\ell}, k \to \llbracket M \rrbracket\}\,;\ \mathsf{adv}\,\{\mathsf{self} : a, \hat{m}, k \to \llbracket N \rrbracket\}\,; \\
&\mathsf{new}\ b : a\,;\ \mathsf{adv}\,\{\mathsf{self} : b, \hat{m}, k \to \llbracket P \rrbracket\}\,;\ \mathsf{adv}\,\{\mathsf{self} : b, \hat{n}, k \to \llbracket Q \rrbracket\}\,; \\
&\mathsf{new}\ p : b\,; \\
&\mathsf{new}\ k_1\,; \\
&\mathsf{adv}\,\{\hat{k}_1, x \to \mathsf{new}\ k_2\,;\ \mathsf{adv}\,\{\hat{k}_2, y \to \mathsf{new}\ k_3\,;\ \mathsf{adv}\,\{\hat{k}_3, z \to \llbracket U \rrbracket\}\,; \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{call}\langle p, \hat{n}, k_3 \rangle\}\,; \\
&\qquad\quad \mathsf{call}\langle p, \hat{m}, k_2 \rangle\}\,; \\
&\mathsf{call}\langle p, \hat{\ell}, k_1 \rangle
\end{aligned}$$ □

**Conjecture 23.** *If* $\mathbf{M} \Rightarrow \mathbf{N}$*, then* $\llbracket \mathbf{M} \rrbracket_k \to^* \equiv \llbracket \mathbf{N} \rrbracket_k$. □

## 6. Conclusions

We have reported some preliminary steps toward attaining a useful type system for $\mu$ABC. Several challenges remain, all discussed in the introduction. First, we face the niggling difference in substitution times for our source and target calculi. Second, and more interestingly, we require a useful notion of semantic equivalence. Third, and most importantly, we must parameterize the sorting system given here with the type of protocols on names described in the introduction. All these problems have solutions, and the solution to the third promises to be very interesting.

## References

[1] M. Abadi, L. Cardelli, P. L. Curien, and J. J. Levy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[2] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. $\mu$ABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer.

[3] C. Clifton and G. T. Leavens. Minimao1: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63(3):321–374, 2006.

[4] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3), 2006.

[5] Peter Hui and James Riely. Temporal aspects as security automata. In *Foundations of Aspect-Oriented Languages (FOAL)*, pages 19–28, 2006.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[7] Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63(3), 2006.

[8] Radha Jagadeesan, Corin Pitcher, and James Riely. Open bisimulation for aspects. In Oege de Moor, editor, *AOSD*. ACM, 2007.

[9] Jay Ligatti, David Walker, and Steve Zdancewic. A type-theoretic interpretation of pointcuts and advice. *Science of Computer Programming*, 63(3), 2006.

[10] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes*. Cambridge Universtity Press, 1991.

[11] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In Colin Runciman and Olin Shivers, editors, *ICFP*, pages 127–139. ACM, 2003.