MiniMAO: Investigating the Semantics of Proceed[*]

Curtis Clifton and Gary T. Leavens
Dept. of Computer Science
Iowa State University
229 Atanasoff Hall
Ames, Iowa 50010-1041
{cclifton,leavens}@cs.iastate.edu

ABSTRACT

This paper describes the semantics of MiniMAO$_1$, a core aspect-oriented calculus. Unlike previous aspect-oriented calculi, it allows `around` advice to change the target object of an advised operation before proceeding. MiniMAO$_1$ accurately models the ways AspectJ allows changing the target object, e.g., at `call` join points. Practical uses for changing the target object using advice include proxies and other wrapper objects.

In addition to accurate modeling of bindings for `around` advice, MiniMAO$_1$ has several other features that make it suitable for the study of aspect-oriented mechanisms, such as those found in AspectJ. Like AspectJ, the calculus consists of an imperative, object-oriented base language plus aspect-oriented extensions. MiniMAO$_1$ has a sound static type system, facilitated by a slightly different form of `proceed` than in AspectJ.

1. INTRODUCTION

This paper describes *MiniMAO$_1$*, a core aspect-oriented calculus. MiniMAO$_1$ is designed to explore two key issues in reasoning about operations in aspect-oriented programs:

— when advice may change the target object of the operation, possibly affecting dynamic method selection, and

— when advice may change or capture the arguments to, or results from, the operation.

MiniMAO$_1$ is sufficiently expressive to encode key aspect-oriented idioms. But by minimizing the set of features, we arrive at a core language that is sufficiently small as to make tractable formal proofs of type soundness and—in planned extensions—proofs of desired modularity properties and verification conditions.

In this paper we describe the dynamic semantics of MiniMAO$_1$ and an interesting portion of its type system. We also state its soundness theorem. Because of space limitations, we refer interested readers to a companion technical report [4] for details that we omit here. We leave the study of reasoning issues to future work. For clarity, we begin with a core object-oriented calculus with classes. We then extend this object-oriented calculus with aspects and advice binding.

2. THE BASE LANGUAGE: MiniMAO$_0$

In this section we introduce *MiniMAO$_0$*, a core imperative object-oriented calculus with classes, derived from Classic Java [8]. Following the lightweight philosophy of Featherweight Java [9], we eliminate interfaces, super calls, and method overloading. We drop `let` expressions and instead use e_1 `;` e_2 to sequentially evaluate e_1 and then e_2. We adopt Featherweight Java's technique of treating the current program and its declarations as global constants. This avoids burdening the formal semantics with excess notation.

To allow later modeling of method call and execution join points, we also separate call and execution in the semantics.

$$
\begin{aligned}
P &::= \mathit{decl}^*\ e \\
\mathit{decl} &::= \texttt{class}\ c\ \texttt{extends}\ c\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
\mathit{field} &::= t\ f \\
\mathit{meth} &::= t\ m(\ \mathit{form}^*\ )\ \{\ e\ \} \\
\mathit{form} &::= t\ \mathit{var},\text{ where } \mathit{var} \neq \texttt{this} \\
e &::= \texttt{new}\ c\texttt{()} \mid \mathit{var} \mid \texttt{null} \mid e.m(\ e^*\ ) \mid \\
&\qquad e.f \mid e.f\ \texttt{=}\ e \mid \texttt{cast}\ t\ e \mid e\texttt{;}\ e
\end{aligned}
$$

$$
\begin{aligned}
c, d &\in \mathscr{C}\text{, the set of a class names} \\
t, s, u &\in \mathscr{T}\text{, the set of types} \\
f &\in \mathscr{F}\text{, the set of field names} \\
m &\in \mathscr{M}\text{, the set of method names} \\
\mathit{var} &\in \{\texttt{this}\} \cup \mathscr{V}\text{, where } \mathscr{V} \text{ is the set of variable names}
\end{aligned}
$$

Figure 1: Syntax of MiniMAO$_0$

2.1 Syntax of MiniMAO$_0$

The syntax for MiniMAO$_0$ is given in Figure 1. A program consists of a sequence of declarations followed by a single expression. Running a program consists of evaluating this expression.

In MiniMAO$_0$ the declarations are all of classes. We omit access modifiers, which would only add gratuitous complexity; hence all methods and fields are globally accessible. MiniMAO$_0$ also omits constructors. All objects are created with their fields set to `null`.

The set of types is denoted by $\mathscr{T}$. MiniMAO$_0$ includes just one built-in type, `Object`, the top of the class hierarchy. `Object` contains no fields or methods. For MiniMAO$_0$, $\mathscr{T} = \mathscr{C}$, the set of valid class names. $\mathscr{C}$ is left unspecified, but we use Java identifier conventions in examples. We follow the same convention for $\mathscr{F}$, $\mathscr{M}$, and $\mathscr{V}$.

Most expressions in MiniMAO$_0$ have a meaning like that in Java, but there are some differences. The expression `new C()` creates an instance of the class named `C`, setting all of its fields to the default `null` value. For syntactic clarity, we follow Classic Java in using a non-Java syntax, `cast` t e, to represent the Java cast `(` t `)` e.

[*]Supported by NSF grants CCF-0428078 and CCF-0429567.

FOAL '05 Chicago, Illinois USA

2.2 Operational Semantics of MiniMAO$_0$

We describe the dynamic semantics of MiniMAO$_0$ using a structured operational semantics [7, 13, 16]. The semantics is quite similar to that for Classic Java. There are two main differences: a stack (used for aspect binding in MiniMAO$_1$) and the separation of method call and execution into separate primitive operations.

For the operational semantics we add two expressions that do not appear in the user-visible syntax.

$$\begin{aligned} e &::= \ldots \mid loc \mid \texttt{(}\ l\ \texttt{(}\ v \ldots \texttt{)}\ \texttt{)} \\ l &::= \texttt{fun}\ m\langle var^* \rangle.e : \tau \\ \tau &::= t \times \ldots \times t \to t \\ v &::= loc \mid \texttt{null} \\ loc &\in \mathscr{L}, \text{ the set of store locations} \end{aligned}$$

One can think of locations, $loc \in \mathscr{L}$, as addresses of object records in a global heap, but we just require that $\mathscr{L}$ is some countable set. The application expression form is used to model method execution independently from method calls. In these expressions, l is a (non-first-class) `fun` term that represents a method and `(` $v\ldots$ `)` is an operand tuple that represents the actual arguments. The application expression thus records information from method dispatch, but before execution of the method body. The `fun` term carries type information—a function type, τ. This type information is not used in evaluation rules, but is helpful in the subject-reduction proof. The use of the application expression form in the operational semantics is described in more detail below.

As is typical in an operational semantics, we consider a subset of the expressions to be irreducible values. The values in MiniMAO$_0$ are the locations and `null`. Evaluation of a well-typed MiniMAO$_0$ program will produce either a value or an exception.

The evaluation context rules, denoted by $\mathbb{E}$, serve as implicit congruence rules and define a left-to-right evaluation order:

$$\begin{aligned} \mathbb{E} ::= &- \mid \mathbb{E}\,.\,f \mid \mathbb{E}\,.\,f\ \texttt{=}\ e \mid v\,.\,f\ \texttt{=}\ \mathbb{E} \mid \texttt{cast}\ t\ \mathbb{E} \mid \mathbb{E}\,\texttt{;}\,e \mid \\ &\mathbb{E}\,.\,m\texttt{(}\,e\ldots\,\texttt{)} \mid v\,.\,m\texttt{(}\,v\ldots\mathbb{E}e\ldots\,\texttt{)} \mid \texttt{(}\ l\ \texttt{(}\ v\ldots\mathbb{E}e\ldots\ \texttt{)}\ \texttt{)} \end{aligned}$$

The evaluation context for the application form only recurses on the arguments and not on the method body expression in the `fun` term of the form. Evaluation of the method body does not take place until the substitution of actuals for formals has been done by the appropriate evaluation rule.

The relation, $\hookrightarrow$, describes evaluation steps:

$$\hookrightarrow : \mathscr{E} \times Stack \times Store \to (\mathscr{E} \cup Excep) \times Stack \times Store$$

This relation takes an expression $e \in \mathscr{E}$, a stack, and a store and maps this to a new expression or an exception, plus a new stack and a new store. Exceptions are elements of

$$Excep = \{\texttt{NullPointerException}, \texttt{ClassCastException}\}.$$

For MiniMAO$_0$, the evaluation relation on the stack is identity, so we leave the set *Stack* undefined for now. The set *Store* contains maps from locations to object records, where an object record has the form $[t\,\texttt{.}\{f \mapsto v \cdot f \in dom(fieldsOf(t))\}]$.

Although suppressed in the evaluation relation, the declarations of the program are used to populate a global *class table*, CT, that maps class names to their declarations.

Evaluation of a MiniMAO$_0$ program begins with the triple consisting of the main expression of the program, a stack, and an empty store. The $\hookrightarrow$ relation is applied repeatedly until the resulting triple is not in the domain of the relation. This terminating condition can arise because the resulting triple contains either an irreducible value or an exception. If the resulting triple contains an irreducible value, then that value, interpreted in the resulting store, is the result of the program. There is no guarantee that this evaluation terminates.

The $\hookrightarrow$ relation is defined by a set of mutually disjoint rules. Except for the CALL and EXEC, these rules are standard and are omitted here. The CALL rule is:

$$\begin{aligned} &\langle \mathbb{E}[loc\,.\,m\texttt{(}\ v_1,\ldots,v_n\ \texttt{)}], J, S\rangle \qquad\qquad \text{CALL} \\ &\quad\hookrightarrow \langle \mathbb{E}[\texttt{(}\ l\ \texttt{(}\ loc, v_1,\ldots,v_n\ \texttt{)}\ \texttt{)}], J, S\rangle \\ &\quad \text{where } S(loc) = [t\,\texttt{.}F] \text{ and } methodBody(t,m) = l \end{aligned}$$

This says that a method call expression, where the target is a location bound in the store, is evaluated by looking up the body of the method and constructing an application form recording the formal parameters, method body, and actual arguments. The interesting part in the definition of the method lookup function is:

$$\frac{\begin{array}{c} CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \exists i \in \{1..p\} \cdot meth_i = t\ m\texttt{(}\ t_1\ var_1,\ldots,t_n\ var_n\ \texttt{)}\ \{\ e\ \} \\ \tau = c \times t_1 \times \ldots \times t_n \to t \end{array}}{methodBody(c,m) = \texttt{fun}\ m\langle \texttt{this}, var_1,\ldots,var_n\rangle.e : \tau}$$

Another part recursively searches in superclasses when a method is not found. This models inheritance of methods.

The application form produced by the CALL rule is evaluated by the EXEC rule:

$$\begin{aligned} &\langle \mathbb{E}[\texttt{(}\ \texttt{fun}\ m\langle var_0,\ldots,var_n\rangle.e : \tau\ \texttt{(}\ v_0,\ldots,v_n\ \texttt{)}\ \texttt{)}], J, S\rangle \quad \text{EXEC} \\ &\quad\hookrightarrow \langle \mathbb{E}[e\{\!|v_0/var_0,\ldots,v_n/var_n|\!\}], J, S\rangle \end{aligned}$$

This rule replaces `this` and the formal parameters in the body with the appropriate values. (The notation $e\{\!|e'/var|\!\}$ denotes the standard capture-avoiding substitution of e' for var in e.)

An example showing the CALL and EXEC rules is given in Section 3.3.6. The companion technical report contains the complete operational semantics. It also contains a separate static semantics and soundness theorem for MiniMAO$_0$.

3. MiniMAO$_1$: ADDING ASPECTS

In this section we add advice binding to MiniMAO$_0$, producing the aspect-oriented core calculus MiniMAO$_1$. Continuing with our minimalist philosophy, the join point model of MiniMAO$_1$ is quite simple. The model only includes `call` and `execution` join points, the parameter binding forms `this`, `target`, and `args`, and the operators for pointcut union, intersection, and negation. We intensionally omit temporal join points, such as `cflow`; the techniques for dealing semantically with such join points are well understood [15], and such temporal join points do not substantially affect the typing rules for aspects.

MiniMAO$_1$ accurately models AspectJ's semantics for around advice, in that it allows advice to change the target object of a method call or execution before proceeding with the operation. Moreover, as in AspectJ, changing the target object at a call join point affects method selection for the call, but changing the target object at an execution join point merely changes the self object of the already selected method. Changing the target object is useful for such idioms as introducing proxy objects. MiniMAO$_1$ does depart from AspectJ's semantics for around advice in two ways: it does not allow changing the `this` (i.e., the caller) object at a `call` join point and it uses a different form of `proceed`, which matches the shape of the advised code rather than the surrounding advice. These differences are discussed more below.

3.1 Related Work

No previous work deals with the actual AspectJ semantics of argument binding for `proceed` expressions and an object-oriented base language. Wand et al. [15] bind all advice parameters at the join point instead of at each subsequent `proceed` expression. Their calculus also is not object-oriented and so does not deal with the effects on method selection of changing the target object. Douence et al. [6] do not formalize advice parameter binding and do not include `proceed` in their language. Jagadeesan et al. [10] only consider primitive `call` and `execution` pointcut descriptors, omitting pointcut operators (like union and intersection) and the ability of advice to change the target object of an invocation. Masuhara and Kiczales [12] do not include around advice in their Scheme-based model for an AspectJ-like language. They do sketch how these features could be added, but do not address the effect on method selection of changing the target object. Aldrich [2] presents a system called "open modules" that includes advice and dynamic join points with a module system that can restrict the set of control flow points to which advice may be attached. The system is not object-oriented, so it does not address the issue of changing the target of a method call, and it does not include state. Dantas and Walker [5] present a simple object-based calculus for "harmless advice". They use a type system with "protection levels" to keep aspects from altering the data of the base program. In keeping with this non-interference property, they do not allow advice to change values when proceeding to the base program. Bruns et al. [3] describe μABC, a name-based calculus in which aspects are the primitive computational entity. Their calculus does not include state. While their calculus does allow modeling of a form of `proceed`, It is difficult to see how it could be used to study the effects of advice on method selection.

Walker et al. [14] use an innovative technique of translating an aspect-oriented language into a labeled core language, where the labels serve as both advice binding sites and targets for `goto` expressions, where they are used to translate around advice that does not proceed. While their work does consider around advice and `proceed` in an object-oriented setting—the object calculus of Abadi and Cardelli [1]—it does not consider changing any arguments to the advised code, let alone the effects on method selection of changing the target object of an invocation.

Our operational semantics for MiniMAO$_1$ draws heavily on the insight of Walker et al. that labeling primitive operations is a useful technique for modeling aspect-oriented languages. However, to handle the run-time changing of the target object and arguments when proceeding from advice, we replace their simple labels with more expressive join point abstractions. Also, rather than introduce these join point abstractions through a static translation from an aspect-oriented language to a core language, we generate them dynamically in the operational semantics. The extra data needed for the join point abstractions (versus the simple static labels) is more readily obtained when they are generated dynamically. (This dynamic generation is also adopted by Dantas and Walker.) Also, directly typing the aspect-oriented language, instead of just showing a type-safe translation to the labeled core language, seems to more clearly illustrate the issues in typing advice, though this is a matter of taste.

3.2 Syntax of MiniMAO$_1$

Figure 2 gives the additional syntax for MiniMAO$_1$. To the declarations of MiniMAO$_0$ we add aspects. For a MiniMAO$_1$ program the set of types, $\mathscr{T}$, is $\mathscr{C} \cup \mathscr{A}$, where $\mathscr{A}$ is the set of aspect names. An aspect declaration includes a sequence of field declarations and a sequence of advice declarations.

We only include `around` advice in MiniMAO$_1$. Operationally, `around` advice can be used to model both `before` and `after` advice. (As noted by Jagadeesan et al. [11], the typing rules necessary for soundness may be less restrictive for `before` or `after` advice.)

$$
\begin{array}{rl}
\mathit{decl} ::= & \ldots \mid \texttt{aspect}\ a\ \{\ \mathit{field}^*\ \mathit{adv}^*\ \} \\
\mathit{adv} ::= & t\ \texttt{around(}\ \mathit{form}^*\ \texttt{)}\ \texttt{:}\ \ \mathit{pcd}\ \{\ e\ \} \\
\mathit{pcd} ::= & \texttt{call(}\ \mathit{pat}\ \texttt{)} \mid \texttt{execution(}\ \mathit{pat}\ \texttt{)} \mid \\
 & \texttt{this(}\ \mathit{form}\ \texttt{)} \mid \texttt{target(}\ \mathit{form}\ \texttt{)} \mid \texttt{args(}\ \mathit{form}^*\ \texttt{)} \mid \\
 & \mathit{pcd}\ \texttt{\&\&}\ \mathit{pcd} \mid \texttt{!}\mathit{pcd} \mid \mathit{pcd}\ \texttt{||}\ \mathit{pcd} \\
\mathit{pat} ::= & t\ \mathit{idPat}\texttt{(..)} \\
e ::= & \ldots \mid e\texttt{.proceed(}\ e^*\ \texttt{)}
\end{array}
$$

$a \in \mathscr{A}$, the set of aspect names

$\mathit{idPat} \in \mathscr{I}$, the set of identifier patterns

Figure 2: Syntax Extensions for MiniMAO$_1$

An advice declaration in MiniMAO$_1$ consists of a return type, followed by the keyword `around` and a sequence of formal parameters. The pointcut descriptor that follows specifies the set of join points—the *pointcut*—where the advice should be executed. A *join point* is any point in the control flow of a program where advice may be triggered. The pointcut descriptor for a piece of advice also specifies how the formal parameters of the advice are to be bound to the information available at a join point. The final part of an advice declaration is an expression that is the advice body.

MiniMAO$_1$ includes a limited vocabulary for pointcut descriptors. The `call` pointcut descriptor matches the invocation of a method whose signature matches the given pattern. The `execution` pointcut descriptor is similar, but matches the join point corresponding to a method execution. In both of these, we restrict method patterns to a concrete return type plus an identifier pattern that is matched against the name of the called method. We choose not to include matching against target or parameter types here because that is essentially syntactic sugar for the `target` and `args` pointcut descriptors.

We leave the set $\mathscr{I}$ of identifier patterns underspecified. Generally, one can think of $\mathscr{I}$ as a class of regular expression languages such that all members of $\mathscr{M}$ are elements of a language in $\mathscr{I}$.

The `this`, `target`, and `args` pointcut descriptors correspond to the parameter-binding forms of these descriptors in AspectJ; they bind the named formal parameters to the corresponding information from the join point. To simplify the operational semantics, the syntax requires a type and a formal parameter. For example, where one could write `this(n)` in AspectJ, one must write `this(Number n)` in MiniMAO (where `Number` is the type of the formal parameter `n` in the advice declaration). While this type could be inferred, including it in the syntax clarifies the formalism. Another simplification versus AspectJ is that the `args` pointcut descriptor in MiniMAO$_1$ binds all arguments available at the join point; such bindings do not allow matching of methods with differing numbers of arguments, unlike those in AspectJ. MiniMAO$_1$ does not include any wildcard or subtype matching for `this`, `target`, or `args` pointcut descriptors.

The final three pointcut descriptor forms represent pointcut negation (`!`*pcd*), union (*pcd* `||` *pcd*), and intersection (*pcd* `&&` *pcd*). These last two are "short circuiting"; for example, if pcd_1 in the form pcd_1 `||` pcd_2 matches a join point, then the bindings defined by pcd_1 are used and pcd_2 is ignored.

MiniMAO$_1$ also includes `proceed` expressions, which are only valid within advice. An expression such as $e_0\texttt{.proceed}(e_1,\ldots,e_n)$ takes a target, e_0, and sequence of arguments, $e_1,\ldots,e_n$, and causes execution to continue with the code at the advised join point—

$$
\begin{aligned}
J &::= j + J \mid \bullet \\
j &::= (\!|k, v_{opt}, m_{opt}, l_{opt}, \tau_{opt}|\!) \\
k &::= \texttt{call} \mid \texttt{exec} \mid \texttt{this} \\
v_{opt} &::= v \mid - \\
m_{opt} &::= m \mid - \\
l_{opt} &::= l \mid - \\
\tau_{opt} &::= \tau \mid -
\end{aligned}
$$

Figure 3: The Join Point Stack

$$
\begin{aligned}
&e ::= \ldots \mid \texttt{joinpt}\ j(\,e^*\,) \mid \texttt{under}\ e \mid \texttt{chain}\ \bar{B}, j(\,e^*\,) \\
&\bar{B} ::= B + \bar{B} \mid \bullet \\
&B ::= [\![b, loc, e, \tau, \tau]\!] \\
&b ::= \langle \alpha, \beta, \beta^* \rangle \\
&\alpha ::= var \mapsto loc \mid - \\
&\beta ::= var \mid - \\
&b \in \mathscr{B}, \text{the set of advice parameter bindings}
\end{aligned}
$$

Figure 4: Expression Forms Added for the Semantics

either the original method or another piece of advice that applies to the same method. As noted above, the `proceed` expression in MiniMAO$_1$ differs from AspectJ. In MiniMAO$_1$, an expression of the form e_0`.proceed`$(e_1,\ldots,e_n)$ must be such that the type of the target, e_0, and the number and types of the arguments, $e_1,\ldots,e_n$, must match those of the *advised methods*. In AspectJ, the arguments to proceed must match the formal parameters of the surrounding *advice*. This design decision matches our intuition for how `proceed` should work; it has little effect on expressiveness in a language with type-safe around advice. Our design also precludes changing the `this` object at `call` join points. Such changes would only be visible from other aspects, not the base program. Precluding these changes eliminates some possibilities for aspect interference, a useful property for our work on aspect-oriented reasoning. We are not aware of any use cases demonstrating a need to allow changing the `this` object.

3.3 Operational Semantics of MiniMAO$_1$

This section gives the changes and additions to the operational semantics for MiniMAO$_1$. We describe the stack, new expression forms introduced for the operational semantics, the new evaluation rules, pointcut descriptor matching, and give evaluation examples.

3.3.1 The Join Point Stack

The stack in MiniMAO$_1$ is a list of *join point abstractions*, which are five-tuples surrounded by half-moon brackets, $(\!|\ldots|\!)$, as shown in Figure 3. A join point abstraction records all the information in a join point that is needed for advice matching and advice parameter bindings, together referred to as *advice binding*. A join point abstraction also includes all the information necessary to proceed from advice to the original code that triggered the join point. A join point abstraction consists of the following parts (most of which are optional and are replaced with "$-$" when omitted):

— a join point kind, k, indicating the primitive operation of the join point, or `this` to record the self object at method or advice execution (for binding the `this` pointcut descriptor);

— an optional value indicating the self object at the join point;

— an optional name indicating the method called or executed at the join point;

— an optional `fun` term recording the body of the method to be executed at an execution join point; and

— an optional a function type indicating the type of the code under the join point (or, equivalently, the type of a `proceed` expression in any advice that binds to the join point).

The code *under* a join point is the program code that would execute at that join point if no advice matched the join point. For example, the code under a method execution join point is the body of the method. The function type includes the type of the target object as the first argument type.

3.3.2 New Expression Forms

The operational semantics relies on three extra expression forms, shown in Figure 4. The first, `joinpt`, reifies join points of a program evaluation into the expression syntax. A `joinpt` expression consists of a join point abstraction followed by a sequence of actual arguments to the code under the join point.

The second expression form that we add for the operational semantics is `under`. An `under` expression serves as a marker that the nested expression is executing under a join point; that is, a join point abstraction was pushed onto the stack before the nested expression was added to the evaluation context. When the nested expression has been evaluated to a value, then the corresponding join point abstraction can be popped from the stack.

The final additional expression form is `chain`. A `chain` expression records a list, $\bar{B}$, of all the advice that matches at a join point, along with the join point abstraction and the original arguments to the code under the join point.

The advice list of a `chain` expression consists of *body tuples*, one per matching piece of advice. For visual clarity, "snake-like" brackets, $[\![\ldots]\!]$, surround each body tuple. A body tuple is comprised of two parts: operational information and type information. The operational information includes: *b*, a parameter binding term described below, *loc*, a location, and *e*, an expression. The location is the self object; it is substituted for `this` when evaluating the advice body. The expression is the advice body.

The *binding term*, *b*, describes how the values of actual arguments should be substituted for formals in the advice body. This substitution is somewhat complex to account for the special binding of the `this` pointcut descriptor, which takes its data from the original join point, and the `target` and `args` pointcut descriptors, which take their data from the invocation or `proceed` expression immediately preceding the evaluation of the advice body.

Structurally, a binding term consists of a variable-location pair, $var \mapsto loc$, which is used for any `this` pointcut descriptors, followed by a non-empty sequence of variables, which represent the formals to be bound to the target object and each argument in order. The "$-$" symbol is used to represent a hole in a binding term. A hole might occur, for example, if a pointcut descriptor did not use `this`. The set of all possible binding terms is $\mathscr{B}$.

The type information in a body tuple is contained in its last two elements. The first of these represents the declared type of the advice, an arrow from formal parameter types to the return type. The second type element, the last element in the body tuple, is the type of any `proceed` expression contained within the advice body. While this type information simplifies the subject-reduction proof, it is not used in the evaluation rules.

3.3.3 *Evaluation Rules for MiniMAO*$_1$

Next we give an intuitive description of the new evaluation rules in MiniMAO$_1$. We add new evaluation context rules to handle the `joinpt`, `under`, and `chain` expressions.

$$\begin{aligned}\mathbb{E} ::= \ldots \mid \texttt{joinpt}\ j\texttt{(}\ v\ldots\mathbb{E}e\ldots\ \texttt{)} \mid \texttt{under}\ \mathbb{E} \mid \\ \texttt{chain}\ \overline{B}, j\texttt{(}\ v\ldots\mathbb{E}e\ldots\ \texttt{)}\end{aligned}$$

The semantics replaces `proceed` expressions with `chain` expressions, so we do not need additional rules for handling `proceed`.

We replace the CALL rule of MiniMAO$_0$ with a pair of rules, CALL$_\text{A}$ and CALL$_\text{B}$ described below, that introduce join points and handle proceeding from advice respectively. We replace the EXEC rule similarly. We introduce three new rules, BODY, ADVISE, and UNDER.

The evaluation of a program in MiniMAO$_1$ does not begin with an empty store as in MiniMAO$_0$. Instead, a single instance of each declared aspect is added to the store. The locations of these instances are recorded in the global *advice table*, *AT*, which is a set of 5-tuples. Each 5-tuple represents one piece of advice. The 5-tuple for the advice t `around(` $t_1\ var_1, \ldots, t_n\ var_n$ `)` `:` pcd `{` e `}`, declared in aspect a, is $\langle loc, pcd, e, (t_1 \times \ldots \times t_n \to t), \tau\rangle$, where loc is such that $S_0(loc) = [a\,\texttt{.}F]$ is the aspect instance for a in the initial store, S_0. The function type τ is the type of `proceed` expressions in e, derived from pcd.

The global class table, *CT*, is extended in MiniMAO$_1$ to also map aspect names to the aspect declarations.

3.3.4 *Splitting the Call Rule*

In MiniMAO$_0$, a method call is evaluated by applying the CALL and EXEC rules in turn. In MiniMAO$_1$, each of these steps is broken into a series of steps. The CALL step becomes:

— CALL$_\text{A}$: creates a `call` join point

— BIND: finds matching advice

— ADVISE: evaluates each piece of advice

— CALL$_\text{B}$: looks up method, creates an application form

A similar division of labor is used for EXEC. We next describe each of these steps in turn.

The CALL$_\text{A}$ rule is as follows:

$$\begin{aligned}&\langle\mathbb{E}[loc\,\texttt{.}\,m\texttt{(}\ v_1,\ldots,v_n\ \texttt{)}], J, S\rangle \qquad\qquad\qquad\qquad \text{CALL}_\text{A}\\ &\quad\hookrightarrow \langle\mathbb{E}[\texttt{joinpt}\ (\!|\texttt{call}, -, m, -, \tau|\!)\texttt{(}\ loc, v_1,\ldots,v_n\ \texttt{)}], J, S\rangle\\ &\quad\text{where } S(loc) = [t\,\texttt{.}F],\\ &\qquad\quad methodType(t,m) = t_1 \times \ldots \times t_n \to t',\\ &\qquad\quad origType(t,m) = t_0, \text{ and } \tau = t_0 \times \ldots \times t_n \to t'\end{aligned}$$

This says that a method call expression with a non-`null` target evaluates to a `joinpt` expression where the join point abstraction carries the information about the call necessary to bind advice and to proceed with the original call. This information is: the `call` kind, the method name, and a function type, τ, for the method that includes a target type in the first argument position. The function type is determined using a pair of auxiliary functions, the interesting bits of which are:

$$\frac{\begin{array}{c}CT(c) = \texttt{class}\ c\ \texttt{extends}\ d\ \texttt{\{}\ field^*\ meth_1\ldots meth_p\ \texttt{\}}\\ \exists i \in \{1..p\} \cdot meth_i = t\ m\texttt{(}\ t_1\ var_1,\ldots,t_n\ var_n\ \texttt{)}\ \texttt{\{}\ e\ \texttt{\}}\end{array}}{methodType(c,m) = t_1 \times \ldots \times t_n \to t}$$

$$\begin{aligned}&origType(t,m) =\\ &\quad \max\{s \in \mathscr{T} \cdot t \preccurlyeq s \wedge methodType(s,m) = methodType(t,m)\}\end{aligned}$$

The first function, *methodType*, searches the class table for the method declaration and returns a function type. The second function, *origType*, finds the type of the “most super” class of the target type that also declares the method m. (The subtyping relation used in *origType* is just the reflexive transitive closure of the `extends` relation on classes, treating aspects as subtypes of `Object`.) The target type included in the `call` join point abstraction generated by CALL$_\text{A}$ is this most super class. Using the most super class allows advice to match a call to any method in a family of overriding methods, by specifying the target type as this most super class. We discuss this a bit more when describing the `target` pointcut descriptor below. Finally, the arguments of the generated `joinpt` expression are the target location—again in the first position—and the arguments of the original call, in order.

The BIND rule is the only place in the calculus where advice binding (lookup) occurs. This rule takes a `joinpt` expression and converts it to a `chain` expression that carries a list of all matching advice for the join point. It also pushes the expression’s join point abstraction onto the join point stack.

$$\begin{aligned}&\langle\mathbb{E}[\texttt{joinpt}\ j\texttt{(}\ v_0,\ldots,v_n\ \texttt{)}], J, S\rangle \qquad\qquad\qquad\qquad \text{BIND}\\ &\quad\hookrightarrow \langle\mathbb{E}[\texttt{under chain}\ \overline{B}, j\texttt{(}\ v_0,\ldots,v_n\ \texttt{)}], j+J, S\rangle\\ &\quad\text{where } adviceBind(j+J,S) = \overline{B}\end{aligned}$$

The rule uses the auxiliary function *adviceBind* to find the (possibly empty) list of advice matching the new join point stack and store.

$$\begin{aligned}&adviceBind(J,S) = \overline{B}, \text{ where } \overline{B} \text{ is a smallest list satisfying}\\ &\forall\langle loc, pcd, e, \tau, \tau'\rangle \in AT \cdot ((matchPCD(J,pcd,S) = b \neq \bot)\\ &\qquad\qquad\qquad\qquad \Longrightarrow [\![\lfloor b, loc, e, \tau, \tau' \rfloor]\!] \in \overline{B})\end{aligned}$$

The *adviceBind* function applies the *matchPCD* function, described in Section 3.3.5, to find the matching advice in the global advice table. (We leave *adviceBind* underspecified. In particular, we don’t give an order for the advice in the list. Any consistent ordering, such as the declaration ordering used in our examples, will suffice.)

Having found the list of matching advice, the BIND rule then constructs a new `chain` expression consisting of this list of advice, the original join point abstraction, and the original arguments. The result expression is wrapped in an `under` expression to record that the join point abstraction must later be popped from the stack.

The ADVISE rule takes a `chain` expression with a non-empty list of advice and evaluates the first piece of advice.

$$\begin{aligned}&\langle\mathbb{E}[\texttt{chain}\ [\![\lfloor b, loc, e, \llcorner, \llcorner \rfloor]\!] + \overline{B}, j\texttt{(}\ v_0,\ldots,v_n\ \texttt{)}], J, S\rangle \qquad \text{ADVISE}\\ &\quad\hookrightarrow \langle\mathbb{E}[\texttt{under}\ e'\{\!|loc/\texttt{this}|\!\}\{\!|(v_0,\ldots,v_n)/b|\!\}], j'+J, S\rangle\\ &\quad\text{where } e' = \langle\!\langle e\rangle\!\rangle_{\overline{B},j} \text{ and } j' = (\!|\texttt{this}, loc, -, -, -|\!)\end{aligned}$$

The general procedure is to substitute for `this` in the advice body with the location, *loc*, of the advice’s aspect and substitute for the advice’s formal parameters according to the binding term, b. But before the substitution occurs, the rule uses the $\langle\!\langle - \rangle\!\rangle_{\overline{B},j}$ auxiliary function to eliminate `proceed` expressions in the advice body.

The “advice chaining” auxiliary function, $\langle\!\langle - \rangle\!\rangle_{\overline{B},j}$, is defined for `proceed` expressions as:

$$\begin{aligned}&\langle\!\langle e_0\,\texttt{.proceed(}\ e_1,\ldots,e_n\ \texttt{)}\rangle\!\rangle_{\overline{B},j}\\ &\qquad\qquad = \texttt{chain}\ \overline{B}, j\texttt{(}\ \langle\!\langle e_0\rangle\!\rangle_{\overline{B},j}, \langle\!\langle e_1\rangle\!\rangle_{\overline{B},j}, \ldots, \langle\!\langle e_n\rangle\!\rangle_{\overline{B},j}\ \texttt{)}\end{aligned}$$

For all other expression forms, the chaining operator is just applied recursively to every subexpression. Thus $\langle\!\langle - \rangle\!\rangle_{\overline{B},j}$ rewrites all `proceed` expressions, replacing them with `chain` expressions carrying the remainder of the advice list $\overline{B}$, along with the join point abstraction, j, needed to proceed to the original operation once the advice list has been exhausted. This rewriting is like that used by

$$e\{\!\!\{\langle v_0,\ldots,v_n\rangle / \langle var \mapsto loc, \beta_0,\ldots,\beta_p\rangle\}\!\!\} =$$
$$e\{\!\!\{loc/var\}\!\!\}\{\!\!\{v_i/var_i\}\!\!\}_{i\in\{0..n\}\cdot\beta_i=var_i} \text{ where } n \le p$$

$$e\{\!\!\{\langle v_0,\ldots,v_n\rangle / \langle -, \beta_0,\ldots,\beta_p\rangle\}\!\!\} =$$
$$e\{\!\!\{v_i/var_i\}\!\!\}_{i\in\{0..n\}\cdot\beta_i=var_i} \text{ where } n \le p$$

In all other cases, binding substitution is undefined.

Figure 5: Binding Substitution

Jagadeesan et al. [10], though they do not consider the target object to be one of the arguments to `proceed`. Advice chaining is illustrated with an example in Section 3.3.6.

After using the advice chaining function to rewrite the advice body, the ADVISE rule uses variable substitution to bind the formal parameters of the advice to the actual arguments. It substitutes the aspect location, *loc*, for `this` and substitutes the actuals for the formals according to *b*. We overload notation to define this substitution for binding terms. Figure 5 gives this definition. The definition says that the variable in the $var \mapsto loc$ pair is replaced with the location, unless there is a hole,"$-$", in this position of the binding term. Each element, β_i, in the binding term that is not a hole must be a variable. Each such variable is replaced with the corresponding argument, v_i. For example:

$$(\texttt{x.f = y})\{\!\!\{\langle\texttt{loc0,loc1}\rangle/\langle\texttt{x} \mapsto \texttt{loc2, -, y}\rangle\}\!\!\} = (\texttt{loc2.f = loc1})$$

The $\texttt{x} \mapsto \texttt{loc2}$ in the binding term does not use data from the arguments $\langle\texttt{loc0,loc1}\rangle$; the value `loc0` is not used because of the hole in the binding term; and `y` is replaced with `loc1`. The type system rules out repeated use of a variable in a binding term.

After substitution, the ADVISE rule pushes a `this` join point abstraction onto the stack and wraps the result expression in an `under` expression.

Once the list of advice has been exhausted, the result is a `chain` expression with an empty advice list, the original join point abstraction, and a sequence of arguments. If the BIND rule had found no advice, then the arguments will be the target and arguments from the original call. Otherwise, the arguments will be whatever was provided by the last piece of advice. This `chain` expression is used by the CALL_B rule to evaluate the original call.

$$\begin{array}{l}\langle\mathbb{E}[\texttt{chain}\ \bullet, (\!|\texttt{call}, -, m, -, \tau|\!)\,(\,loc, v_1,\ldots,v_n\,)], J, S\rangle \quad \text{CALL}_\text{B}\\ \quad\hookrightarrow \langle\mathbb{E}[(\,l\ (\,loc, v_1,\ldots,v_n\,)\,)], J, S\rangle\\ \quad\quad \text{where } S(loc) = [t\,.F] \text{ and } methodBody(t,m) = l\end{array}$$

The CALL_B rule looks up the type of the (possibly changed) target object in the store and finds the method body in the global class table. The rule takes the method name from the join point abstraction. The result of the rule is an application expression, just like the result of the CALL rule in MiniMAO_0.

Because both the CALL_A and CALL_B rules use a target location for method lookup, there are corresponding rules for `null` targets. These rules just map to a triple with a `NullPointerException` and are omitted here.

A General Technique. The technique used to convert the CALL rule from the MiniMAO_0 calculus into a pair of rules, with intervening advice binding and execution, is general. The first rule in the new pair replaces the original expression with a `joinpt` expression, ready for advice binding. The second rule in the pair takes a `chain` expression, exhausted of advice, and maps it to a new expression like the result expression of the rule from MiniMAO_0. This is how the two new EXEC rules are generated:

$$\begin{array}{l}\langle\mathbb{E}[(\,l\ (\,v_0,\ldots,v_n\,)\,)], J, S\rangle \quad\quad \text{EXEC}_\text{A}\\ \quad\hookrightarrow \langle\mathbb{E}[\texttt{joinpt}\ (\!|\texttt{exec}, v_0, m, l, \tau|\!)\,(\,v_0,\ldots,v_n\,)], J, S\rangle\\ \quad\quad \text{where } l = \texttt{fun}\ m\langle var_0,\ldots,var_n\rangle.e : \tau\\ \langle\mathbb{E}[\texttt{chain}\ \bullet, (\!|\texttt{exec}, v, m, l, \tau|\!)\,(\,v_0,\ldots,v_n\,)], J, S\rangle \quad \text{EXEC}_\text{B}\\ \quad\hookrightarrow \langle\mathbb{E}[\texttt{under}\ e\{\!\!\{v_0/var_0,\ldots,v_n/var_n\}\!\!\}], j + J, S\rangle\\ \quad\quad \text{where } l = \texttt{fun}\ m\langle var_0,\ldots,var_n\rangle.e : \tau \text{ and}\\ \quad\quad\quad j = (\!|\texttt{this}, v_0, -, -, -|\!)\end{array}$$

The EXEC_A rule replaces the application expression with a `joinpt` expression. The join point abstraction of this expression includes the `exec` kind, the method name, the `fun` term of the application, and the type of the `fun` term. The abstraction also includes, in the position reserved for `this` objects, the value of the target object from the argument tuple, because `target` and `this` objects are the same at an `execution` join point. The arguments to the `joinpt` expression are the arguments to the original application expression.

The EXEC_B rule takes a `chain` expression that has been exhausted of its advice. It applies the `fun` term from the `chain`'s join point abstraction to the argument sequence, substituting the arguments for the variables in the body of the `fun` term. Like ADVISE, the EXEC_B rule pushes a `this` join point abstraction onto the stack and wraps its result expression in an `under` expression.

It would be straightforward to add pointcut descriptors and join points for any of the primitive operations in the original calculus. We would have to generalize the data carried in the join point abstractions to accommodate additional information, but the BIND and ADVISE rules would remain unchanged. Because the `call` and `exec` join points are sufficient for our study, we choose not to include join points for the other primitive operations. To do so would just introduce additional notation and bookkeeping.

The Under Rule. The UNDER rule is the simplest of the new evaluation rules.

$$\begin{array}{l}\langle\mathbb{E}[\texttt{under}\ v], J, S\rangle \hookrightarrow \langle\mathbb{E}[v], J', S\rangle \quad \text{UNDER}\\ \quad \text{where } J = j + J', \text{ for some j}\end{array}$$

It just extracts the value from the `under` expression and pops one join point abstraction from the stack.

3.3.5 Pointcut Matching

Following Wand et al. [15], we define the *matchPCD* function for matching pointcut descriptors to join points using a boolean algebra over binding terms. Our binding terms, as described in Section 3.3.2 above, are somewhat more complex than theirs, since we model `this`, `target`, and `args` pointcut descriptors and faithfully model the semantics of `proceed` from AspectJ with regard to changing target objects in advice. Nevertheless, the basic technique is the same.

The boolean algebra is:

$$\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\} \qquad b \in \mathscr{B} \qquad r \in \mathscr{B}_\perp \qquad b \vee r = b$$

$$\perp \vee r = r \qquad \perp \wedge r = \perp \qquad b \wedge \perp = \perp \qquad b \wedge b' = b \sqcup b'$$

$$\neg\perp = \langle -, -\rangle \qquad \neg b = \perp$$

The terms of the algebra are drawn from the set $\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\}$, where binding terms can be thought of as "true" and $\perp$ as "false". The operators in the algebra are conjunction ($\wedge$), disjunction ($\vee$), and complement ($\neg$). The double complement of an element is not necessarily the original element, unless we consider all binding terms to be isomorphic; the effect of this detail on advice binding is discussed below. The binary operators are short circuiting; for example, $b \vee r = b$, ignoring the value of r. One difference in our algebra, versus Wand et al. [15], is in the conjunction of two non-$\perp$ terms. Our calculus must consider the bindings from both terms, because we have more than one pointcut descriptor that can bind formals. Sometimes these bindings must be combined, for example when both a `target` and `args` pointcut descriptor are used. The bindings are combined using a pointwise join:

$$\begin{aligned}
&\langle \alpha, \beta_0, \ldots, \beta_n \rangle \sqcup \langle \alpha', \beta'_0, \ldots, \beta'_p \rangle \\
&\qquad = \langle \alpha \sqcup \alpha', \beta_0 \sqcup \beta'_0, \ldots, \beta_q \sqcup \beta'_q \rangle \\
&\qquad\qquad \text{where } q = \max(n, p), \\
&\qquad\qquad\qquad \forall i \in \{(n+1)..q\} \cdot (\beta_i = -), \text{ and} \\
&\qquad\qquad\qquad \forall i \in \{(p+1)..q\} \cdot (\beta'_i = -)
\end{aligned}$$

The pointwise join operator extends the shorter binding term if the two terms do not have the same number of elements. The join operator, $\sqcup$, on pairs of α or β terms resolves to the term that is not a hole. Collisions in the join operator, where neither binding has a hole at a given position, are resolved in favor of the left-hand term; however, the typing rules for pointcut descriptors ensure that such collisions do not occur in well-typed programs.

The rules defining *matchPCD* are straightforward. If the pointcut descriptor matches the join point stack, then the rules construct the appropriate binding term; otherwise they evaluate to $\perp$.

The `call` rule only matches if the most recent join point is of the corresponding kind and the return type and name of the method under the join point are matched by the pattern:

$$\begin{aligned}
&\mathit{matchPCD}(\llparenthesis k, \_, m, \_, t_0 \times \ldots \times t_p \to t \rrparenthesis + J, \\
&\qquad \texttt{call(}\ u\ \mathit{idPat}\texttt{(..)}\ \texttt{)}, S) \\
&\qquad = \begin{cases} \langle -, - \rangle & \text{if } k = \texttt{call},\ t = u,\ m \in \mathit{idPat} \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

Because this pointcut descriptor does not bind formal parameters, a match is indicated by an empty binding term. The `execution` rule is similar.

Two rules are used to handle `this` pointcut descriptors:

$$\begin{aligned}
&\mathit{matchPCD}(\llparenthesis \_, v, \_, \_, \_ \rrparenthesis + J, \texttt{this(}\ t\ \mathit{var}\ \texttt{)}, S) \\
&\qquad = \begin{cases} \langle \mathit{var} \mapsto v, - \rangle & \text{if } v \neq \texttt{null},\ S(v) = [s.F],\ s \preccurlyeq t \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
&\mathit{matchPCD}(\llparenthesis \_, -, \_, \_, \_ \rrparenthesis + J, \texttt{this(}\ t\ \mathit{var}\ \texttt{)}, S) \\
&\qquad = \mathit{matchPCD}(J, \texttt{this(}\ t\ \mathit{var}\ \texttt{)}, S)
\end{aligned}$$

Together, these rules find the most recent join point where the optional self object location is provided in the join point abstraction. Once found, if the object record in that location is a subtype of the formal parameter type, then the formal named by the pointcut descriptor is mapped to the location; otherwise the result is $\perp$.

The `target` pointcut descriptor is handled similarly, but uses the target type from the join point instead:

$$\begin{aligned}
&\mathit{matchPCD}(\llparenthesis \_, \_, \_, \_, s_0 \times \ldots \times s_n \to s \rrparenthesis + J, \\
&\qquad \texttt{target(}\ t\ \mathit{var}\ \texttt{)}, S) \\
&\qquad = \begin{cases} \langle -, \mathit{var} \rangle & \text{if } s_0 = t \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

A rule for searching through the join point stack is elided. Unlike the `this` pointcut descriptor, the location to be bound to the formals is not available from the join point abstraction. The location may come from a `proceed` expression to be evaluated later. Also unlike `this`, `target` requires an exact type match. This is necessary for type soundness, as noted by Jagadeesan et al. [11]. If the descriptor were to match when the target type was a supertype of the parameter type, then the advice could call a method on the object bound to the formal that did not exist in the object's class. On the other hand, if the descriptor were to match when the target type was a subtype of the parameter type, then the advice could replace the target object with a supertype before proceeding to a method call. If this supertype did not declare the method, then a runtime type error would result.[1] Thus, for soundness the `target` pointcut descriptor must use exact type matching.

This restriction to exact type matching is not as severe as it may seem at first. This is because when the CALL$_A$ rule generates the target type for its join point abstraction, it uses the type of the class declaring the top-most method in the method overriding hierarchy. Thus, the actual target object for a matched call may be a subtype of the target type that was matched exactly. Using the declaring class of this top-most method also means that advice can be written to match a call to any method in a family of overriding methods. Unlike the CALL$_A$ rule, the EXEC$_A$ rule creates a join point abstraction using the actual target type. Again, this is necessary for soundness. At an `exec` join point method selection has already occurred and advice cannot be allowed to change the target object to a superclass even if that superclass declared an overridden method.

The rule for the `args` pointcut descriptor is similar to the one for `target` above. It matches if the argument types of the most recent join point match those of the pointcut descriptor. The resulting binding includes all formals named in the pointcut descriptor in the corresponding positions. As with the `target` pointcut descriptor, only the relative position to be bound, not the actual value, is available until the advice is executed.

The rules for pointcut descriptor operators (which we elide) simply appeal to the corresponding operators in the binding algebra: union to disjunction, intersection to conjunction, and negation to complement. The definition of complement implies that $\neg\neg \mathit{pcd} \neq \mathit{pcd}$. Both would match the same pointcut, but the former would not bind any formals while the later might. (This is slightly different than AspectJ, which simply disallows binding pointcut descriptors under negation operators.)

A final rule says that any cases not covered by the other rules evaluates to $\perp$. This just serves to make *matchPCD* a total function, handling cases that do not occur in the evaluation of a well-typed program (such as matching against an empty join point stack).

3.3.6 *Example Evaluations in MiniMAO$_1$*

This section gives examples of several evaluations.

Calls in MiniMAO$_0$ vs. MiniMAO$_1$. Suppose we have the program declared in Figure 6. This program does not include any aspects and the result of evaluating it is the same in MiniMAO$_0$ and MiniMAO$_1$, though the difference in the steps taken is illustrative. In both cases there is an evaluation step with left hand side:

$\langle \texttt{L0.m(L1)}, \bullet, S \rangle$

where the store S maps both `L0` and `L1` to `C1` objects. In MiniMAO$_0$ this evolves by the CALL and EXEC rules:

$$\hookrightarrow \langle (\texttt{fun m}\langle\texttt{this, a}\rangle\texttt{.(this;a):}\tau\ \texttt{(L0,L1)}), \bullet, S \rangle \qquad \text{(CALL)}$$

[1]Indeed, in AspectJ 1.2, which includes subtype matching for its `target` pointcut descriptor, one can generate a run-time type error in just this way.

```
class Cl extends Object {
    Object m(Cl a) { this; a }
}

new Cl().m(new Cl);
```

Figure 6: A Sample Program Without Aspects

$$\hookrightarrow \langle \texttt{L0; L1}, \bullet, S\rangle \qquad (\textsc{Exec})$$

where we leave τ as an exercise for the reader. On the other hand, the evaluation in MiniMAO$_1$ is:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\langle \texttt{L0.m(L1)}, \bullet, S\rangle} \\
\hookrightarrow \langle \texttt{joinpt}\ \llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis\ \texttt{(L0, L1)}, \bullet, S\rangle & (\textsc{Call}_\textsc{A}) \\
\hookrightarrow \langle \texttt{under chain}\ \bullet, \llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis\ \texttt{(L0, L1)}, J, S\rangle & \\
 & (\textsc{Bind}) \\
\hookrightarrow \langle \texttt{under} & \\
\quad \texttt{(fun m}\langle \texttt{this, a}\rangle\texttt{.(this;a):}\tau\ \texttt{(L0,L1))}, J, S\rangle & \\
 & (\textsc{Call}_\textsc{B}) \\
\hookrightarrow \langle \texttt{under} & (\textsc{Exec}_\textsc{A}) \\
\quad \texttt{joinpt}\ \llparenthesis \texttt{exec}, \texttt{L0}, \texttt{m}, l, \tau \rrparenthesis\ \texttt{(L0, L1)}, J, S\rangle & \\
\hookrightarrow \langle \texttt{under under} & (\textsc{Bind}) \\
\quad \texttt{chain}\ \bullet, \llparenthesis \texttt{exec}, \texttt{L0}, \texttt{m}, l, \tau \rrparenthesis\ \texttt{(L0, L1)}, J', S\rangle & \\
\hookrightarrow \langle \texttt{under under (L0; L1)}, J', S\rangle & (\textsc{Exec}_\textsc{B})
\end{array}
$$

where l is $\texttt{fun m}\langle \texttt{this, a}\rangle\texttt{.(this;a):}\tau$, and τ', J, and J' are left to the reader. Each step in the original evaluation is split into two parts, with intervening advice lookup.

Advice Binding. Suppose we add the aspect declaration of Figure 7 to the program in Figure 6. The presence of this advice changes the result of the first BIND step above (i.e., the one for the `call` pointcut descriptor). BIND's call to *adviceBind* uses the following application of *matchPCD*:[2]

$$
\begin{array}{l}
\mathit{matchPCD}(\llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis, \mathit{pcd}, S) \\
\qquad\qquad\qquad \text{where} \quad \tau' = \texttt{Cl}\times\texttt{Cl}\rightarrow\texttt{Object}, \text{ and} \\
\qquad\qquad\qquad\qquad\quad \mathit{pcd} \text{ is from Figure 7} \\
= \mathit{matchPCD}(\llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis, \texttt{call(Object m(..))}, S) \\
\qquad \wedge\, \mathit{matchPCD}(\llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis, \texttt{target(Cl t)}, S) \\
\qquad \wedge\, \mathit{matchPCD}(\llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis, \texttt{args(Cl s)}, S) \\
= (\langle -, - \rangle \sqcup \langle -, \texttt{t} \rangle) \sqcup \langle -, -, \texttt{s} \rangle \\
= \langle -, \texttt{t} \rangle \sqcup \langle -, -, \texttt{s} \rangle \\
= \langle -, \texttt{t}, \texttt{s} \rangle
\end{array}
$$

Using this matching derivation, the result of the BIND step is:

$$
\begin{array}{l}
\langle \texttt{under chain}\ \llbracket \langle -, \texttt{t}, \texttt{s}\rangle\texttt{, L2, this, }\tau'\texttt{, }\tau' \rrbracket, \\
\quad \llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis\ \texttt{(L0, L1)}, J, S\rangle
\end{array}
$$

where `L2` is the location of the aspect instance in the initial store. This triple evolves by the ADVISE rule. Because the body of the advice does not proceed to the advised code, the result of this step is the final result of the program, after using UNDER to pop the join point stack:

$$
\begin{array}{ll}
\hookrightarrow \langle \texttt{under under L2}, J'', S\rangle & (\textsc{Advise}) \\
\hookrightarrow \langle \texttt{under L2}, J, S\rangle & (\textsc{Under}) \\
\hookrightarrow \langle \texttt{L2}, \bullet, S\rangle & (\textsc{Under})
\end{array}
$$

```
aspect A {
    Object around(Cl t, Cl s) :
        call(Object m(..))
            && target(Cl t) && args(Cl s)
    { this }
}
```

Figure 7: Aspect Added to Program of Figure 6

```
aspect A {
    Object around(Cl t, Cl s) :
        call(Object m(..))
            && target(Cl t) && args(Cl s)
    {
        s.proceed(t) // swaps target, argument
    }
}

class Cl extends Object {
    Object m(Cl a) { this; a }
}

class SCl extends Cl {
    Object m(Cl a) { new Object() }
}

new Cl().m(new SCl);
```

Figure 8: A Sample Program Demonstrating Proceed

Advice Chaining. A final example considers advice that proceeds to the advised code and changes the target object. Consider the program in Figure 8. Unlike our previous examples, the advice proceeds and there is a subclass, `SCl`, which is used for the argument to the method call. Evaluation of this program reaches a stage where the result of the BIND rule is:

$$
\begin{array}{l}
\langle \texttt{under chain} \\
\qquad \llbracket \langle -, \texttt{t}, \texttt{s}\rangle\texttt{, L2, s.proceed(t), }\tau'\texttt{, }\tau' \rrbracket, \\
\qquad\quad \llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis\ \texttt{(L0, L1)}, J, S\rangle
\end{array}
$$

where, as before, `L2` is the location of `A`'s instance and `L0` is the location of a `Cl` instance, but now `L1` is the location of a `SCl` instance. This triple evolves by the ADVISE rule, which calculates

$$\langle\!\langle \texttt{s.proceed(t)} \rangle\!\rangle_{\bullet, j} = \texttt{chain}\ \bullet, j\ \texttt{(s, t)}$$

where $j = \llparenthesis \texttt{call}, -, \texttt{m}, -, \tau' \rrparenthesis$. The rule then substitutes into this expression according to the binding term $\langle -, \texttt{t}, \texttt{s}\rangle$ to form its result, with the order of the two locations swapped as compared to the original, advice-free example above:

$$
\begin{array}{ll}
\hookrightarrow \langle \texttt{under under chain}\ \bullet, j\ \texttt{(L1, L0)}, J'', S\rangle & (\textsc{Advise}) \\
\hookrightarrow \langle \texttt{under} & \\
\quad \texttt{(fun m}\langle \texttt{Cl this, Cl a}\rangle\texttt{.(new Object()):}\tau\ \texttt{(L1,L0))}, & \\
\quad\ J'', S\rangle & \\
 & (\textsc{Call}_\textsc{B})
\end{array}
$$

[2]Technically the store must be different than before, due to the aspect instance in the initial store. However, because S is underspecified, we use the same meta-variable here to facilitate comparisons.

The method body found by the CALL$_B$ rule is declared in `SCl`, instead of in `Cl`.

We invite the reader to consider the same example, but replace the advice's `call` pointcut descriptor with a similar `execution` one. This will demonstrate that changing the target object when proceeding at an `exec` join point does not affect method selection.

3.4 Static Semantics of MiniMAO$_1$

We next sketch some of the static semantics of MiniMAO$_1$. We focus on the typing of pointcuts and advice, since they are the most interesting deviations from past work.

The rules for typing pointcut descriptors make use of a simple algebra over $\mathscr{T} \cup \{\bot\}$, whose only operator, $\sqcup$, is used to combine type information when pointcuts are intersected:

$$t \sqcup \bot = t \qquad \bot \sqcup t = t \qquad \bot \sqcup \bot = \bot$$

The operation is undefined for $t \sqcup s$, because in the type judgment for pointcuts such a combination would indicate a collision and is disallowed. This operation is also lifted to type sequences.

The type of a pointcut descriptor, *pcd*, has six parts, $\hat{u}\,\mathtt{.}\,\hat{u}'\,\mathtt{.}\,U\,\mathtt{.}\,\hat{u}''\,\mathtt{.}\,V_1\,\mathtt{.}\,V_2$, where:

— $\hat{u}$ is the `this` type matched by *pcd*;

— $\hat{u}'$ is the target type;

— U is the tuple of argument types;

— $\hat{u}''$ is the return type;

— V_1 is the set of variables that would definitely be bound by *pcd* at a matched join point; and

— V_2 is the set of variables that might be bound there.

Each of the type parts may also be $\bot$ to indicate that the information cannot be determined from the pointcut descriptor. The two sets of variables, V_1 and V_2, represent "must-bind" and "may-bind" sets respectively, which are useful in reasoning about variable bindings in pointcut unions and intersections. Well-typed advice requires that the must-bind and may-bind sets are identical (see the first hypothesis of T-ADV below).

The pointcut descriptor typing rules are mostly straightforward. We discuss a couple of them here. The T-TARGPCD rule gives the type for a `target` pointcut descriptor:

T-TARGPCD
$$\frac{\Gamma(var) = t}{\Gamma \vdash \mathtt{target(}\; t\; var\; \mathtt{)} : \bot\,\mathtt{.}\,t\,\mathtt{.}\,\bot\,\mathtt{.}\,\bot\,\mathtt{.}\,\{var\}\,\mathtt{.}\,\{var\}}$$

The hypothesis of the above rule looks up the type of *var* in the type environment Γ. (Γ is a partial map from $\mathscr{V} \cup \{\mathtt{this}, \mathtt{proceed}\}$ to $\mathscr{T}$.) The conclusion of the rule records the target type, t, of the pointcut descriptor and records that the must- and may-bind sets are both $\{var\}$. The rules for the other base cases (`call`, `execution`, `this`, and `args`) are similar.

The most interesting of the typing rules for recursive pointcut descriptors is the one for intersection:

T-INTPCD
$$\frac{\begin{array}{c}\Gamma \vdash pcd_1 : \hat{u}_1\,\mathtt{.}\,\hat{u}'_1\,\mathtt{.}\,U_1\,\mathtt{.}\,\hat{u}''_1\,\mathtt{.}\,V_1\,\mathtt{.}\,V'_1 \\ \Gamma \vdash pcd_2 : \hat{u}_2\,\mathtt{.}\,\hat{u}'_2\,\mathtt{.}\,U_2\,\mathtt{.}\,\hat{u}''_2\,\mathtt{.}\,V_2\,\mathtt{.}\,V'_2 \\ \hat{u} = \hat{u}_1 \sqcup \hat{u}_2 \qquad \hat{u}' = \hat{u}'_1 \sqcup \hat{u}'_2 \qquad U = U_1 \sqcup U_2 \qquad \hat{u}'' = \hat{u}''_1 \sqcup \hat{u}''_2 \\ V'_1 \cap V'_2 = \emptyset \qquad V = V_1 \cup V_2 \qquad V' = V'_1 \cup V'_2\end{array}}{\Gamma \vdash pcd_1\;\mathtt{\&\&}\;pcd_2 : \hat{u}\,\mathtt{.}\,\hat{u}'\,\mathtt{.}\,U\,\mathtt{.}\,\hat{u}''\,\mathtt{.}\,V\,\mathtt{.}\,V'}$$

This rule allows for the combination of the various binding forms in pointcut descriptors like `target(T t) && args(S s)`. The first two hypotheses obtain the types of pcd_1 and pcd_2. The next four hypotheses combine these types using the $\sqcup$ operator described above. These hypotheses select the non-$\bot$ entries from the types and prevent duplicate bindings. For example, if both pcd_1 and pcd_2 have a non-$\bot$ target type, $\hat{u}'_1 \sqcup \hat{u}'_2$ is undefined and pcd_1 `&&` pcd_2 has no type. Finally the last three hypotheses deal with the must- and may-bind sets. $V'_1 \cap V'_2 = \emptyset$ requires no overlap in the sets variables that may be bound by the two pointcut descriptors. The last two hypotheses calculate the combined must- and may-bind sets.

Advice is well typed if its pointcut descriptor matches a join point where the code under the join point has target type u_0, argument types $u_1, \ldots, u_p$ and return type u.

T-ADV
$$\frac{\begin{array}{c} var_1 : t_1, \ldots, var_n : t_n \vdash pcd : \_\,\mathtt{.}\,u_0\,\mathtt{.}\,\langle u_1, \ldots, u_p\rangle\,\mathtt{.}\,u\,\mathtt{.}\,V\,\mathtt{.}\,V \\ V = \{var_1, \ldots, var_n\} \\ var_1 : t_1, \ldots, var_n : t_n, \mathtt{this} : a, \mathtt{proceed} : (u_0 \times \ldots \times u_p \rightarrow u) \vdash e : s \\ s \preccurlyeq t \preccurlyeq u \end{array}}{\vdash t\;\mathtt{around(}\;t_1\;var_1, \ldots, t_n\;var_n\;\mathtt{)}\;\;\mathtt{:}\;\;pcd\;\;\mathtt{\{}\;\;e\;\;\mathtt{\}}\;\;\text{OK in } a}$$

The "$\_$" in the first hypothesis indicates that the type bound by a `this` pointcut descriptor does not affect the advice type. The pointcut descriptor must also specify bindings for all of the formal parameters of the advice; the use of $\{var_1, \ldots, var_n\}$ for both the must- and may-bind sets ensures this. Finally, the body of the advice is typed in an environment that gives each formal its declared type; gives `this` the aspect type, a; and gives `proceed` the type derived from *pcd*. In this environment, the advice body must have a type that is a subtype of the declared return type of the advice. In turn, this declared return type must be a subtype of the return type of the original code under the join point. This allows the result of the advice to be substituted for the result of the original code.

Rule T-ADV permits advice to declare a return type that is a subtype of that of the advised method. This means that advice like:

```
A around(C t) :
    call(B m(..)) && target(C t) && args()
{ t.proceed() }
```

is not well typed if A is a proper subtype of B: the `proceed` expression has type `B`, which is not a subtype of the declared return type of the advice. Wand et al. [15, §5.3] argue that this advice should be typable, but we disagree. This case is really no different than a super call in a language with covariant return-type specialization. In such a language, an overriding method that specializes the return type cannot merely return the result of a super call as its result. The overriding method must ensure that the result is appropriately specialized.

3.5 Meta-theory of MiniMAO$_1$

The key property of MiniMAO$_1$ is that it is type sound: a well-typed program either converges to a value or exception, or else it diverges. We prove this using the usual subject reduction and progress theorems. For MiniMAO$_0$, the proofs closely follow those of Flatt et al. [8]. The soundness proof for MiniMAO$_1$ relies on a pair of key lemmas that we sketch here. The companion technical report [4] gives the full details.

The first key lemma is used in the BIND case of the subject reduction proof. The lemma relates advice binding to advice typing. It is used to argue that the list of advice that matches at a `joinpt` expression can be used by the BIND rule to generate a well typed `chain` expression. We prove the lemma using a structural induction on the type derivation for the pointcut of the matching advice.

The second key lemma states that advice chaining, replacing `proceed` expressions with `chain` expressions, does not affect typing judgments given the appropriate assumptions. This lemma is used for the ADVISE case in the subject reduction proof.

The subject reduction and progress theorems are standard and are elided. Finally, we have the soundness theorem.

THEOREM 1 (SOUNDNESS). *Given a program*

$$P = decl_1 \ldots decl_n\ e, \text{ with } \vdash P\ OK,$$

and a valid store S_0, *then either the evaluation of e diverges or else* $\langle e, \bullet, S_0 \rangle \overset{*}{\hookrightarrow} \langle v, J, S \rangle$ *and one of the following hold for v:*

— $v = loc$ *and* $loc \in dom(S)$,

— $v = \texttt{null}$, *or*

— $v \in \{\texttt{NullPointerException}, \texttt{ClassCastException}\}$

4. CONCLUSION

In many respects MiniMAO$_1$ faithfully explains the semantics of AspectJ's around advice on method call and execution join points. In particular, MiniMAO$_1$ faithfully models the binding of arguments and the ability of `proceed` to change the target object in a call join point. The semantics supports this ability by breaking the processing of method calls into several steps: (i) creating the join point for the call, (ii) finding matching advice, (iii) evaluating each piece of advice, and (iv) finally creating an application form. Since the target object is only used to determine the method called in step (iv) (the CALL$_B$ rule), the advice can change the target by using a different target in the `proceed` expression. Such a change affects the application form created, which affects the join point created for the method's execution.

In addition to the necessary simplifications, MiniMAO$_1$, also has a few interesting differences from AspectJ. In particular the typing of `proceed` and the various pointcut descriptions has a different philosophy from AspectJ. Its typing in MiniMAO$_1$ corresponds to the type of the method being advised, instead of being related to the type of the advice's formal parameters. This contributes to a simpler and more understandable semantics for `proceed`.

Future work involves using MiniMAO$_1$ to study the reasoning problems indicated in the introduction.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[2] Jonathan Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL 2004 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004*, pages 7–18, Lancaster, UK, 2004. URL `http://www.cs.iastate.edu/~leavens/FOAL/papers-2004/proceedings.pdf`.

[3] Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μabc: A minimal aspect calculus. In *Proceedings of the 2004 International Conference on Concurrency Theory*, pages 209–224. Springer-Verlag, 2004.

[4] Curtis Clifton and Gary T. Leavens. MiniMAO: Investigating the semantics of proceed. Technical Report TR05-01, Iowa State University, 2005. Available from `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-08/TR.ps.gz`.

[5] Daniel S. Dantas and David Walker. Harmless advice. In *The 12th International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, Long Beach, California, 2005. ACM.

[6] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Reflection 2001*, number 2192 in LNCS. Spring-Verlag, November 2001.

[7] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, chapter 7, pages 241–269. Springer-Verlag, 1999. URL `http://citeseer.ist.psu.edu/flatt99programmers.html`.

[9] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[10] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In Luca Cardelli, editor, *ECOOP 2003, European Conference on Object-Oriented Programming, Darmstadt, Germany*, volume 2743, pages 54–73. Springer-Verlag, 2003.

[11] Radha Jagadeesan, Alan Jeffrey, and James Riely. A typed calculus for aspect oriented programs. Available from `ftp://fpl.cs.depaul.edu/pub/rjagadeesan/typedABL.pdf`, Feb 2004.

[12] Hidehiko Masuhara and Gregar Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP 2003 - Object-Oriented Programming European Conference*, pages 2–28. Springer-Verlag, 2003.

[13] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[14] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139, Uppsala, Sweden, 2003. ACM Press.

[15] Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *Trans. on Prog. Lang. and Sys.*, 26(5):890–910, 2004.

[16] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115 (1):38–94, 1994.