MINIMAO$_1$: An Imperative Core Language for Studying Aspect-Oriented Reasoning

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

December 8, 2005

Abstract

This paper describes MiniMAO$_1$, a core aspect-oriented language. Unlike previous aspect-oriented calculi and core languages, MiniMAO$_1$ 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.

MiniMAO$_1$ was designed to serve as a core language for studying modular specification and verification in the aspect-oriented paradigm. To this end MiniMAO$_1$

— has an imperative, reference-based semantics,

— models the control-flow effects of changing target object bindings with advice, and

— has a safe static type system.

The first two features make MiniMAO$_1$ suitable for the study of aspect-oriented mechanisms, such as those found in AspectJ. These features are important for studying the interaction of aspect-oriented language features with modular specification and verification. A statically type-safe language is also import for such research. AspectJ does not have a safe static type system. To achieve static type safety MiniMAO$_1$ uses a slightly different form of proceed and advice binding than in AspectJ. These changes are sufficient for static type safety, but we do not claim that they are necessary; a less restrictive type system might suffice.

This paper gives an operational semantics, type system, and proof of soundness for MiniMAO$_1$.

1 Introduction

This paper describes a core aspect-oriented [14] language, *MiniMAO$_1$*. MiniMAO$_1$ is designed to serve as a core language for studying modular specification and verification in the aspect-oriented paradigm. In particular, MiniMAO$_1$ explores 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. Clifton's dissertation [5][1] applies MiniMAO$_1$ to demonstrate the soundness of modular reasoning in an AspectJ-like language with some simple extensions ("aspect maps" and "spectator aspects").

[1] This paper is based on chapter 3 of Clifton's dissertation. An earlier version of this paper, without as many formal details and without proofs, appeared in the proceedings of the FOAL workshop in 2005 [6].

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

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

Figure 1: Syntax of MiniMAO$_0$

For clarity, we begin with a core object-oriented language with classes. We then extend this object-oriented language with aspects and advice binding. We assume that the reader is familiar with the basic concepts of aspect-oriented programming as embodied in the AspectJ programming language [15].

2 MiniMAO$_0$: A Core Object-Oriented Calculus with Classes

In this section we introduce *MiniMAO$_0$*, an imperative, object-oriented core language with classes. Because features like mutation and variable capture are central to studying the reasoning issues in aspect-oriented programming, MiniMAO$_0$ is derived from Classic Java [10]. In particular, MiniMAO$_0$ uses Classic Java's imperative, reference-based semantics. Without these imperative features, it is extremely difficult to write aspects that do typical tasks, such as logging; so simplifying the language and proofs by taking out imperative features is not sensible. Following the lightweight philosophy of Featherweight Java [11], we eliminate interfaces, super calls, method overloading, and let expressions from Classic Java. Since eliminating let expressions eliminates implicit sequencing [1], we introduce explicit expression sequencing. We adopt Featherweight Java's technique of treating the current program and its declarations as global constants. This reduces the notational burden of the formal semantics.

One innovation of MiniMAO$_0$ is the separation of method call and method execution into two primitive operations. This simplifies the modeling of method call and method execution join points in the aspect-oriented version of the language.

2.1 Syntax of MiniMAO$_0$

The syntax for MiniMAO$_0$ is given in Figure 1. A MiniMAO$_0$ program consists of a sequence of declarations followed by a single expression. The expression represents the entry point for the program, like the execution of a program's main method in Java.

In MiniMAO$_0$ the declarations are all of classes; later MiniMAO$_1$ will add aspect declarations. A class declaration gives the name of the class, the name of its superclass, and a sequence of fields and methods. MiniMAO$_0$ does not include access modifiers; all methods and fields are globally accessible. For our purposes, access modifiers would be gratuitous complexity. MiniMAO$_0$ also omits constructors. All objects are instantiated with their fields set to null. Constructors can be modeled by defining methods that

initialize the fields.

The set of types in MiniMAO$_0$ is denoted by $\mathscr{T}$. MiniMAO$_0$ includes just one built-in type, that of Object, the top-most class in all class hierarchies. In MiniMAO$_0$, 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 for examples we will take it to be the set of all valid Java identifiers. We use a similar convention for the sets $\mathscr{F}$ of valid field names, $\mathscr{M}$ of valid method names, and $\mathscr{V}$ of valid variable names.

The field declarations within a class declaration just give a type and a field name. We omit field initializers from the language.

Method declarations in MiniMAO$_0$ consist of a return type, the method name, a sequence of formal parameters (which are similar in form to field declarations), and a method body expression. For simplicity we do not include return statements in MiniMAO$_0$; instead, the result of the method is just the result of evaluating the body expression, with proper substitution for formal parameters and this.

MiniMAO$_0$ includes just a few different kinds of expressions. The expression new C() creates an instance of the class named C, setting all of its fields to the default null value. Variable references and null expressions have the usual meaning. Method invocations are written as in Java, as are field access and update. For syntactic clarity, we follow Classic Java in using the syntax cast t e to represent the Java cast (t) e. Finally, we include an expression for sequencing: e; e. One could simulate sequencing through a baroque combination of classes and method calls, but the additional complexity of including an actual sequencing expression is small, so we choose the direct approach.

2.2 Operational Semantics of MiniMAO$_0$

We describe the dynamic semantics of MiniMAO$_0$ using a structured operational semantics [9, 17, 20]. The semantics is given in Figure 2 on the next page and is quite similar to that for Classic Java. There are three main differences: a stack (which will be used for aspect binding in MiniMAO$_1$), a primitive operation for expression sequencing, and the separation of method call and execution into separate primitive operations. This latter change is helpful in MiniMAO$_1$ for distinguishing the effect on dynamic dispatch of method call advice versus that of method execution advice.

2.2.1 The Abstract Machine

The operational semantics of MiniMAO$_0$ use an abstract machine that tracks the current expression, a global store, and a "join point stack". The stack is only included for use by the subsequent aspect-oriented extension; it remains empty in MiniMAO$_0$.

We add two expressions for the operational semantics of MiniMAO$_0$ that do not appear in the user-visible syntax. To model state, we extend the set of expressions to include locations, $loc \in \mathscr{L}$. One can think of locations as addresses of object records in a global heap, but for the purposes of the core language we just require that $\mathscr{L}$ is some countable set.

To model method execution independently from method calls, we add an application expression form, where a (non-first-class) fun term represents a method and an operand tuple represents the actual arguments after method dispatch but before substitution of actual arguments for formal parameters. The fun term carries type information: a function type, τ, from a tuple of target and argument types to the return type of the method. 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 in the following subsection.

As is typical in an operational semantics, we consider a subset of the expressions, denoted by v, to be irreducible values. The values in MiniMAO$_0$ are the locations and null. Evaluation of a well-typed MiniMAO$_0$ program will produce a value or an exception; this soundness property is proven later.

Evaluation contexts are denoted by $\mathbb{E}$. The definition of evaluation contexts serves both to define implicit congruence rules and to define a left-to-right evaluation order. The first rule, "$-$", is the base case. The next two rules require that the target of a method call be evaluated before the arguments and that the arguments are evaluated in left-to-right order. The rule for the application form only recurses on the

Syntax extensions:

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

Objects:

$$
\begin{aligned}
o &::= [t\,.\,F] \\
F &: \mathscr{F} \to \mathscr{V}
\end{aligned}
$$

Evaluation contexts:

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

Evaluation relation:

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

$\langle \mathbb{E}[\mathsf{new}\ c()], J, S\rangle \hookrightarrow \langle \mathbb{E}[loc], J, S \oplus (loc \mapsto [c\,.\,\{f \mapsto \mathsf{null} \cdot f \in dom(fieldsOf(c))\}])\rangle$ NEW
where $loc \notin dom(S)$

$\langle \mathbb{E}[loc.m(\ v_1, \ldots, v_n\ )], J, S\rangle \hookrightarrow \langle \mathbb{E}[(\ l\ (\ loc, v_1, \ldots, v_n\ )\ )], J, S\rangle$ CALL
where $S(loc) = [t\,.\,F]$ and $methodBody(t, m) = l$

$\langle \mathbb{E}[(\ \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle . e : \tau\ (\ v_0, \ldots, v_n\ )\ )], J, S\rangle \hookrightarrow \langle \mathbb{E}[e\{\!|\, v_0/\, var_0, \ldots, v_n/\, var_n \,|\!\}], J, S\rangle$ EXEC

$\langle \mathbb{E}[loc.f], J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J, S\rangle$ GET
where $S(loc) = [t\,.\,F]$ and $F(f) = v$

$\langle \mathbb{E}[loc.f = v], J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J, S \oplus (loc \mapsto [t\,.\,F \oplus (f \mapsto v)])\rangle$ SET
where $S(loc) = [t\,.\,F]$

$\langle \mathbb{E}[\mathsf{cast}\ t\ loc], J, S\rangle \hookrightarrow \langle \mathbb{E}[loc], J, S\rangle$ CAST
where $S(loc) = [s\,.\,F]$ and $s \preccurlyeq t$

$\langle \mathbb{E}[\mathsf{cast}\ t\ \mathsf{null}], J, S\rangle \hookrightarrow \langle \mathbb{E}[\mathsf{null}], J, S\rangle$ NCAST

$\langle \mathbb{E}[v;\ e], J, S\rangle \hookrightarrow \langle \mathbb{E}[e], J, S\rangle$ SKIP

$\langle \mathbb{E}[\mathsf{null}.m(\ v_1, \ldots, v_n\ )], J, S\rangle \hookrightarrow \langle \mathsf{NullPointerException}, J, S\rangle$ NCALL

$\langle \mathbb{E}[\mathsf{null}.f], J, S\rangle \hookrightarrow \langle \mathsf{NullPointerException}, J, S\rangle$ NGET

$\langle \mathbb{E}[\mathsf{null}.f = v], J, S\rangle \hookrightarrow \langle \mathsf{NullPointerException}, J, S\rangle$ NSET

$\langle \mathbb{E}[\mathsf{cast}\ t\ loc], J, S\rangle \hookrightarrow \langle \mathsf{ClassCastException}, J, S\rangle$ XCAST
where $S(loc) = [s\,.\,F]$ and $s \not\preccurlyeq t$

Figure 2: Operational Semantics of MiniMAO$_0$

arguments and not on the method body expression in the fun term. 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 rules cast $t\ \mathbb{E}$ and $\mathbb{E}.f$ are simple congruence rules. The rule for sequencing requires that the left expression in a pair be evaluated first. The last two rules require that the target object for a field update be evaluated before the new value for the field is evaluated.

The relation, $\hookrightarrow$, describes the steps in the evaluation of a MiniMAO$_0$ program. The relation takes an expression $e \in \mathscr{E}$ (the set of all expressions), a stack, and a store and maps this to a new expression or an exception, plus a new stack and a new store. For MiniMAO$_0$, the evaluation relation on the stack is identity, so we leave the set *Stack* undefined for now; the aspect-oriented language will manipulate the stack for advice binding. The set *Store* consists of a map from locations to object records, where an object record has the form $[t\,.\,\{f \mapsto v \cdot f \in dom(fieldsOf(t))\}]$. That is, an object record consists of a type and a map from the fields of that type to their values. The exceptions in MiniMAO$_0$ are elements of the set $Excep = \{\mathsf{NullPointerException}, \mathsf{ClassCastException}\}$.

Evaluation of a MiniMAO$_0$ program begins with the triple consisting of the main expression of the program, an empty 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 either because the resulting triple contains an irreducible value or it contains 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.

We write $\hookrightarrow^*$ for the reflexive, transitive closure of the $\hookrightarrow$ relation. (Because of exceptions, the range of $\hookrightarrow$ does not equal its domain. So to be precise, $\hookrightarrow^*$ is actually the $\hookrightarrow$ relation unioned with the reflexive, transitive closure of the $\hookrightarrow$ relation restricted to the range $\mathscr{E} \times Stack \times Store$.)

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.

The $\hookrightarrow$ relation is defined by a set of mutually disjoint rules. In the subsequent subsections, we briefly describe the intuition behind each of the evaluation rules, and we give a small example program and trace its evaluation.

2.2.2 Intuition for Evaluation Rules

The NEW rule says that an expression new $c()$ evaluates to a fresh location, where that location maps to an object record of the appropriate type with all of its fields initialized to null. This rule also uses two auxiliary functions, which are formally defined in Figure 3 on the following page. The $\oplus$ operator represents map update; the *fieldsOf*(c) function returns a map from all the fields defined in c (and its supertypes) to the types of those fields.

The CALL rule 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 (using the *methodBody* auxiliary function) and constructing an application form with a function term, l, recording the formal parameters and method body and an argument tuple recording the actual arguments. The separate EXEC rule evaluates this application form by replacing 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.)

To illustrate these rules, we use the program in Figure 4 on page 7. The evaluation steps below start with the store $S_0 = \{\mathsf{loc0} \mapsto [\mathsf{Zero}\,.\,\{\mathsf{pred} \mapsto \mathsf{null}\}]\}$. In the following, the initial call of the succ method evolves in two evaluation steps, first using the CALL rule and then the EXEC rule.

$\langle$*loc0.succ()*.add(**new** Zero().succ().succ()), $J, S_0\rangle$

$\hookrightarrow \langle$(***fun*** *succ*$\langle$***this***$\rangle$.***new*** *Natural().setPred(**this**) (loc0))*.add(**new** Zero().succ().succ()), $J, S_0\rangle$ (CALL)

$\hookrightarrow \langle$***new*** *Natural()*.setPred(loc0).add(**new** Zero().succ().succ()), $J, S_0\rangle$ (EXEC)

(A fully worked out version of this example is present in Clifton's dissertation [5].)

The rule, NCALL, says that if the target value of a method call expression is null, then the result of evaluation is a NullPointerException. (The evaluation rules which result in exceptions are grouped together

Map update:

$$\oplus : \mathscr{P}(T \rightarrowtail U) \times (T \rightarrowtail U) \rightarrow \mathscr{P}(T \rightarrowtail U), \text{ polymorphic in } T \text{ and } U$$

$$A \oplus (t \rightarrowtail u) = \{t' \rightarrowtail u' \cdot (t' \neq t \wedge A(t') = u') \vee (t = t' \wedge u = t')\}$$

Field lookup:

$$\frac{CT(c) = \text{class } c \text{ extends } d \ \{\ t_1\ f_1 \ldots t_n\ f_n\ meth^*\ \} \qquad fieldsOf(d) = F'}{fieldsOf(c) = \{f_i \rightarrowtail t_i \cdot i \in \{1..n\}\} \cup F'} \qquad fieldsOf(\text{Object}) = \emptyset$$

Method lookup:

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

$$\frac{\begin{array}{c} CT(c) = \text{class } c \text{ extends } d \ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \nexists i \in \{1..p\} \cdot meth_i = t\ m(\ t_1\ var_1, \ldots, t_n\ var_n\ )\ \{\ e\ \} \qquad methodBody(d, m) = l \end{array}}{methodBody(c, m) = l}$$

Method type lookup:

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

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

Valid method overriding:

$$\frac{methodType(d, m) = t_1 \times \ldots \times t_n \rightarrow t}{override(m, d, t_1 \times \ldots \times t_n \rightarrow t)}$$

$$\frac{\begin{array}{c} CT(d) = \text{class } d \text{ extends } d' \ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \nexists i \in \{1..p\} \cdot meth_i = t\ m(\ t_1\ var_1, \ldots, t_n\ var_n\ )\ \{\ e\ \} \qquad override(m, d', \tau) \end{array}}{override(m, d, \tau)}$$

$$\frac{}{override(m, \text{Object}, t_1 \times \ldots \times t_n \rightarrow t)}$$

Valid class:

$$\frac{CT(c) = \text{class } c \text{ extends } d \ \{\ \ldots\ \}}{isClass(c)} \qquad \frac{}{isClass(\text{Object})}$$

Figure 3: Auxiliary Functions for MiniMAO$_0$

```
class Natural extends Object {
   /** Stores the predecessor of this. */
   Natural pred;

   /** Initializes the predecessor of this. */
   Natural setPred(Natural pred) {
      this.pred = pred;
      this
   }

   /** Returns the predecessor of this. */
   Natural pred() {
      this.pred
   }

   /** Returns the successor of this. */
   Natural succ() {
      new Natural().setPred(this)
   }

   /** Returns the sum of this and n. */
   Natural add(Natural n) {
      this.pred().add(n.succ())
   }
}

class Zero extends Natural {
   Natural pred() {
      this
   }

   Natural add(Natural n) {
      n
   }
}

new Zero().succ().add(new Zero().succ().succ())        // 1 + 2
```

Figure 4: A Sample MiniMAO$_0$ Program

$$t \preccurlyeq t \qquad \frac{t \preccurlyeq s \qquad s \preccurlyeq u}{t \preccurlyeq u} \qquad \frac{CT(c) = \text{class } c \text{ extends } d \ \{ \ \ldots \ \}}{c \preccurlyeq d}$$

Figure 5: Subtyping in MiniMAO$_0$

at the bottom of Figure 2 on page 4.)

The GET and SET rules both look up the object record for the target location in the store. The GET rule then looks up the value of the named field. The SET rule, on the other hand, updates the store with a new object record that is identical to the original object record except that the value of the named field is replaced with the new value. The NGET and NSET rules handle the cases where the target value is null.

Three different rules deal with type casts. The CAST rule handles valid casts of non-null values. A cast is valid at evaluation time if the target type of the cast is a supertype of the actual type of the value. Figure 5 gives the subtyping relation for MiniMAO$_0$. The relation is just the reflexive, transitive closure of the syntactic extends relation. The NCAST rule handles casts of null. For both CAST and NCAST, the result of evaluation is just the value within the cast expression. The XCAST rule handles invalid casts of non-null values; in this case, the result of evaluation is a ClassCastException.

Finally, the SKIP rule says that a sequence expression, where the first expression is already reduced to a value, is evaluated to just the second expression.

2.3 Static Semantics of MiniMAO$_0$

Figure 6 on the following page gives the static semantics for MiniMAO$_0$. To avoid overburdening the typing rules, we make the following simplifying assumptions (implicit side conditions):

— All declared classes in a program have unique names.

— The extends relation on classes, generated by the declarations in a program, is acyclic. (Formally, $t \preccurlyeq u \wedge u \preccurlyeq t \implies t = u$.)

— Field and method names are unique within a single declaration.

The typing rules for expressions use a simple type environment, Γ. The type environment Γ is a finite partial map from $\mathcal{V}_{\text{this}}$ to $\mathcal{T}$, where $\mathcal{V}_{\text{this}} = \mathcal{V} \cup \{\text{this}\}$ and $\mathcal{T}$ is the set of all types. Unlike the expression typing rules, the typing rules for programs, classes, and methods do not rely on a type environment.

The static semantics is standard, but a brief explanation of the typing rules is warranted.

The program typing rule, T-PROG, says that a program is well typed if all of its declarations are well typed and if its main expression is well typed in the empty type environment. (The effect of the declarations is implicit in the expression's typing through the global class table, for example see rule T-NEW.)

A class declaration is well typed, according to T-CLASS, if the declaration does not shadow any of its superclass fields; if its declared superclass is, in fact, a class; and if its methods are all well typed.

Rule T-MET says that a method declaration is well typed within a class c if the method body can be shown to have a subtype of the declared return type by assuming that the formal parameters have their declared types and this has type c. The last hypothesis of T-MET uses the auxiliary function *override* (defined in Figure 3 on page 6) to require that either the method is fresh (i.e., no method of the same name exists in a superclass) or the method is a valid override—it has the same type as the overridden superclass method. This definition precludes overloading.

The expression typing rules are mostly straightforward. Instead of a separate subsumption rule as is sometimes used, subtyping is handled directly in the appropriate rules (T-CALL, T-EXEC, and T-SET). The T-NEW, T-OBJ, and T-VAR rules are obvious. The T-LOC rule is used in the meta-theory, where the domain of the type environment is extended to include locations. The T-NULL rule says that null can be treated as having any type.

Program typing:

$$\text{T-PROG}\quad \frac{\forall i \in \{1..n\} \cdot \vdash decl_i \text{ OK} \qquad \emptyset \vdash e : t}{\vdash decl_1 \ldots decl_n\ e \text{ OK}}$$

Class typing:

$$\text{T-CLASS}\quad \frac{\forall i \in \{1..n\} \cdot f_i \notin dom(fieldsOf(d)) \qquad isClass(d) \qquad \forall j \in \{1..p\} \cdot \vdash meth_j \text{ OK in } c}{\vdash \text{class } c \text{ extends } d \ \{\ t_1\ f_1 \ldots t_n\ f_n\ meth_1 \ldots meth_p\ \}\ \text{OK}}$$

Method typing:

$$\text{T-MET}\quad \frac{var_1 : t_1, \ldots, var_n : t_n, \text{this} : c \vdash e : u \qquad u \preccurlyeq t \qquad CT(c) = \text{class } c \text{ extends } d\ \{\ \ldots\ \} \qquad override(m, d, t_1 \times \ldots \times t_n \to t)}{\vdash t\ m(\ t_1\ var_1, \ldots,\ t_n\ var_n\ )\ \{\ e\ \}\ \text{OK in } c}$$

Expression typing:

$$\text{T-NEW}\quad \frac{c \in dom(CT)}{\Gamma \vdash \text{new } c() : c} \qquad \text{T-OBJ}\quad \frac{}{\Gamma \vdash \text{new Object}() : \text{Object}} \qquad \text{T-VAR}\quad \frac{\Gamma(var) = t}{\Gamma \vdash var : t} \qquad \text{T-LOC}\quad \frac{\Gamma(loc) = t}{\Gamma \vdash loc : t} \qquad \text{T-NULL}\quad \frac{t \in \mathcal{T}}{\Gamma \vdash \text{null} : t}$$

$$\text{T-CALL}\quad \frac{\Gamma \vdash e_0 : t_0 \qquad \forall i \in \{1..n\} \cdot \Gamma \vdash e_i : u_i \qquad methodType(t_0, m) = t_1 \times \ldots \times t_n \to t \qquad \forall i \in \{1..n\} \cdot u_i \preccurlyeq t_i}{\Gamma \vdash e_0.m(\ e_1, \ldots, e_n\ ) : t}$$

$$\text{T-EXEC}\quad \frac{\Gamma, var_0 : t_0, \ldots, var_n : t_n \vdash e : s \qquad s \preccurlyeq t \qquad \forall i \in \{0..n\} \cdot \Gamma \vdash e_i : u_i \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i \qquad \tau = t_0 \times \ldots \times t_n \to t}{\Gamma \vdash (\ \text{fun } m\langle var_0, \ldots, var_n \rangle . e : \tau\ (\ e_0, \ldots, e_n\ )\ ) : t}$$

$$\text{T-GET}\quad \frac{\Gamma \vdash e : s \qquad fieldsOf(s)(f) = t}{\Gamma \vdash e.f : t}$$

$$\text{T-SET}\quad \frac{\Gamma \vdash e_1 : u \qquad fieldsOf(u)(f) = t \qquad \Gamma \vdash e_2 : s \qquad s \preccurlyeq t}{\Gamma \vdash e_1.f\ = e_2 : s} \qquad \text{T-CAST}\quad \frac{\Gamma \vdash e : s}{\Gamma \vdash \text{cast } t\ e : t} \qquad \text{T-SEQ}\quad \frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1;\ e_2 : t}$$

Figure 6: Static Semantics of MiniMAO$_0$

The T-CALL rule uses the type of the target object expression to look up the method type. The rule checks that all argument expressions are subtypes of the formal parameter types. The type of the entire call expression is the declared return type of the method.

The T-EXEC rule is only necessary for the subject-reduction proof. The lambda application form, which T-EXEC types, can only appear during evaluation; it cannot be used statically. The rule uses the formal parameter types to type the body expression. It also ensures that the actual arguments are subtypes of the formal parameter types.

The T-GET and T-SET rules use the type of the target object expression to look up the field type. For T-GET, the field type is the type of the whole expression. For field update, T-SET requires that the right-hand expression, giving the new value of the field, be a subtype of the field type. The type of the right-hand expression is also the type of the whole update expression.

We choose to use a single rule, T-CAST, for typing casts in MiniMAO$_0$. This is more permissive than Java, which disallows casting an expression to an unrelated type. As pointed out by Igarashi et al. [11], we need to allow such "stupid casts" between unrelated types to achieve a proof of subject reduction for a small-step semantics. This is because an upcast followed by a downcast can reduce to a stupid cast. Igarashi et al. [11] introduce a technique of splitting the casting rule into three rules: one for downcasts, one for upcasts, and one for stupid casts. The stupid cast rule allows for a subject reduction proof while still matching the typing rules of Java: a Featherweight Java program is a well-typed Java program if its typing derivation does not include a stupid cast. The three cast typing rules of Featherweight Java also allow a strong safety property: for a program that can be typed without downcasts or stupid casts, progress is always possible. In our terminology, they show that evaluation cannot result in a ClassCastException. (Featherweight Java is a functional calculus and does not include a null value. Hence, NullPointerExceptions are not an issue there.) We choose to use the simpler single cast rule, since the precise correspondence to Java's cast typing rules is not needed for our work and a soundness theorem that admits exceptions is sufficiently strong.

Finally, the T-SEQ rule simply requires both expressions in a sequence to be well typed and gives the sequence the type of the second expression.

2.4 Meta-theory of MiniMAO$_0$

The key property of MiniMAO$_0$ is that it is type sound: a well-typed MiniMAO$_0$ program either converges to a value or exception, or else it diverges. We prove this using the usual subject reduction and progress theorems. The proofs closely follow those of Flatt et al. [10].

Before stating and proving a subject reduction theorem, we first need a notion of consistency between a type environment and a store [9, 10]. For the meta-theory, the type environment maps variables and store locations to types, $\Gamma : (\mathcal{V}_{\mathsf{this}} \cup \mathscr{L}) \to \mathscr{T}$.

Definition 1 (Environment-Store Consistency)**.** A type environment Γ and a store S are *consistent*, and we write $\Gamma \approx S$, if all of the following are satisfied:[2]

1. $\forall\, loc \in \mathscr{L} \cdot S(loc) = [t\,.\,F] \implies$
   (a) $\Gamma(loc) = t$ and
   (b) $dom(F) = dom(fieldsOf(t))$ and
   (c) $rng(F) \subseteq dom(S) \cup \{\mathsf{null}\}$ and
   (d) $\forall\, f \in dom(F) \cdot (F(f) = loc'$ and $fieldsOf(t)(f) = u$ and $S(loc') = [t'\,.\,F'] \implies t' \preccurlyeq u)$
2. $\forall\, loc \in \mathscr{L} \cdot (loc \in dom(\Gamma) \implies loc \in dom(S))$
3. $dom(S) \subseteq dom(\Gamma)$

[2] Using an implication in part 2 of this definition allows the type environment to give types to global constants should we wish to add basic types to the language.

The following standard substitution lemma will also be useful. (Proofs of this and other interesting lemmas are given in the appendix.)

Lemma 2 (Substitution). *If* $\Gamma, var_1 : t_1, \ldots, var_n : t_n \vdash e : t$ *and* $\forall i \in \{1..n\} \cdot \Gamma \vdash e_i : s_i$ *where* $s_i \preccurlyeq t_i$ *then* $\Gamma \vdash e\{e_1 / var_1, \ldots, e_n / var_n\} : s$ *for some* $s \preccurlyeq t$.

We will also need four other standard lemmas: the first pair let us introduce fresh references into, and remove unused references from, the domain of the type environment; the second pair of lemmas let us replace subderivations within typing derivations, with or without subtyping. These lemmas are useful when handling reductions within evaluation contexts.

Lemma 3 (Environment Extension). *If* $\Gamma \vdash e : t$ *and* $a \notin dom(\Gamma)$, *then* $\Gamma, a : t' \vdash e : t$.

Lemma 4 (Environment Contraction). *If* $\Gamma, a : t' \vdash e : t$ *and* a *is not free in* e, *then* $\Gamma \vdash e : t$.

Lemma 5 (Replacement). *If* $\Gamma \vdash \mathbb{E}[e] : t$, $\Gamma \vdash e : t'$, *and* $\Gamma \vdash e' : t'$, *then* $\Gamma \vdash \mathbb{E}[e'] : t$.

Lemma 6 (Replacement with Subtyping). *If* $\Gamma \vdash \mathbb{E}[e] : t$, $\Gamma \vdash e : u$, *and* $\Gamma \vdash e' : u'$ *where* $u' \preccurlyeq u$, *then* $\Gamma \vdash \mathbb{E}[e'] : t'$ *where* $t' \preccurlyeq t$.

Theorem 7 (Subject Reduction). *Given a well typed MiniMAO$_0$ program, for an expression* e, *a stack* J, *a store* S, *and a type environment* Γ *consistent with* S, *if* $\Gamma \vdash e : t$ *and* $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$, *then there exist* Γ' *and* t' *such that* $\Gamma' \approx S'$, $\Gamma' \vdash e' : t'$, *and* $t' \preccurlyeq t$.

Theorem 8 (Progress). *For an expression* e, *a stack* J, *a store* S, *and a type environment* Γ *consistent with* S, *if* $\Gamma \vdash e : t$ *then either:*

— $e = loc$ *and* $loc \in dom(S)$,

— $e = null$, *or*

— *one of the following hold:*

- $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$
- $\langle e, J, S\rangle \hookrightarrow \langle NullPointerException, J', S'\rangle$
- $\langle e, J, S\rangle \hookrightarrow \langle ClassCastException, J', S'\rangle$.

The soundness property of MiniMAO$_0$ follows from subject reduction and progress.

Theorem 9 (Soundness). *Given a program* $P = decl_1 \ldots decl_n$ e, *if* $\vdash P$ *OK then either the evaluation of* e *diverges or else* $\langle e, \bullet, \emptyset\rangle \overset{*}{\hookrightarrow} \langle v, J, S\rangle$ *where one of the following holds for* v*:*

— $v = loc$ *and* $loc \in dom(S)$,

— $v = null$,

— $v = NullPointerException$, *or*

— $v = ClassCastException$

Proof. If e diverges then the claim holds. If e converges, then note that the empty environment is consistent with the empty store. The proof (by induction on the number of evaluation steps) is immediate from Theorem 7 (Subject Reduction) and Theorem 8 (Progress). □

3 MiniMAO$_1$: Adding Aspects

In this section we add advice binding to MiniMAO$_0$. The result is an aspect-oriented core language, MiniMAO$_1$. Continuing with the 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. The omission of the temporal join points, such as cflow, is an intentional decision. The techniques for dealing semantically with such join points are well understood [19], and such temporal join points do not substantially affect the modular reasoning properties of aspects. The join points that we do include are sufficient for studying the effects of advice on method dispatch (with call, execution, and target) and on program state (this, args, and target again).

MiniMAO$_1$ accurately models AspectJ's semantics for around advice [15], 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. Such proxy objects can be used in aspect-oriented implementations of persistence or for redirecting method calls to remote machines.

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, which could be bound using this pointcut descriptors, and it uses a different form of proceed, which syntactically looks like the advised method call rather than the surrounding advice declaration as in AspectJ. Both of these changes reduce the expressiveness of the language. Limiting the power of this pointcut descriptors reduces the possibility of interaction between distinct pieces of advice applied to the same join point, though we know of no idioms that use this interaction. Changing the form of proceed helps us to create a safe static type system for MiniMAO$_1$. Our restricted proceed is perhaps not expressive enough for general use, however it is sufficiently expressive for our purposes. Future work will evaluate how much of the expressiveness of AspectJ's proceed can be maintained while adding static type safety. The two differences in around advice between AspectJ and MiniMAO$_1$ are discussed in more detail below.

One motivation for the design of MiniMAO$_1$ is to keep pointcut matching, advice execution, and primitive operations in the base language as separate as possible. This goal causes us to use more evaluation rules that are strictly necessary. One way to think of MiniMAO$_1$ is as an operational semantics for an aspect-oriented virtual machine, where each primitive operation may generate a join point that may trigger other rules for advice matching. Our approach increases the syntactic complexity of the language, but we find that it actually simplifies reasoning. The approach keeps separate concepts in separate rules that can be analyzed with separate lemmas.

No previous work on formalizing the semantics of an aspect-oriented language deals with the actual AspectJ semantics of argument binding for proceed expressions and an object-oriented base language. Our language is motivated by the insight of Walker et al. [18] 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 [7].) 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. Our type system is motivated by that of Jagadeesan et al. [13]. We discuss this and other related work in more detail in Section 4.

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

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

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

Figure 7: Syntax Extensions for MiniMAO$_1$

3.1 Syntax of MiniMAO$_1$

Figure 7 gives the additional syntax for MiniMAO$_1$. To the declarations of MiniMAO$_0$ we add aspects, with a ranging over the set, $\mathscr{A}$, of aspect names. As for identifiers in MiniMAO$_0$, we leave $\mathscr{A}$ unspecified, but for examples will draw names from the set of legal Java identifiers. For a MiniMAO$_1$ program the set of types is $\mathscr{T} = \mathscr{C} \cup \mathscr{A}$. 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. [13], there are some interesting differences for typing around advice versus before or after advice. We discuss these in more detail later.)

An advice declaration in MiniMAO$_1$ consists of a return type, followed by the keyword around and a sequence of formal parameters. A pointcut descriptor comes next. The pointcut descriptor 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. 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 just 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 $\mathscr{I}$. For examples, we will treat $\mathscr{I}$ as the set of all legal Java identifiers, but allowing the wildcard character, *, as a legal identifier character.

The execution pointcut descriptor is like the one for call, except that it matches the join point corresponding to a method execution. There are two key differences between method call and method execution join points:

— at a method call join point the this object is the caller, while at a method execution join point the this object is the callee, and

— a method call join point is reached before method dispatch is performed, but the corresponding method execution join point is reached after method dispatch.

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). This type elaboration

could easily be performed automatically; 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; that is, MiniMAO$_1$ does not include AspectJ's mechanism for binding arguments when matching methods with differing numbers of arguments. 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*). Pointcut negation only reverses the boolean (match or mismatch) value of the negated pointcut. Any parameters bound by the negated pointcut are dropped. Pointcut union and intersection 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. This is the same semantics as implemented in recent versions of AspectJ.

MiniMAO$_1$ also includes proceed expressions, which are only valid within advice. An expression such as e_0.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—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, $e_1,\ldots,e_n$, 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. More importantly, this design reflects the fact that static type safety depends on a proceed expression having the type of the code that it will execute rather than the type of the advice in which it appears.

Our design also precludes changing the this object bound by a this pointcut descriptor. For execution join points, the this and target objects are the same, so the ability to change the target binding suffices. For call join points changing the this binding might affect the values bound to parameters of subsequent advice, but the change would not be observable by the advised method. Thus, 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 binding.

3.2 Operational Semantics of MiniMAO$_1$

This section gives the changes and additions to the operational semantics for MiniMAO$_1$. Subsections describe the stack in MiniMAO$_1$, new expression forms introduced for the operational semantics, the new evaluation rules, and pointcut descriptor matching. Another subsection gives several example evaluations.

3.2.1 The Join Point Stack

As in AspectJ, advice selection in MiniMAO$_1$ is based on matching pointcut descriptors against a stack representing the join points encountered during the execution of the program. Although MiniMAO$_1$ does not include temporal pointcut descriptors, like cflowbelow, a join point stack is still useful for advice selection. In particular, the join point stack lets us track the object that should be bound by this and target pointcut descriptors independent of call and execution join points. This is necessary because although the this binding would be known when a method or piece of advice starts executing, it would not be known when a subsequent method call occurs. The stack allows the abstract machine to retrieve the this object when matching advice to the method call.

The stack in MiniMAO$_1$, as shown in Figure 8 on the next page, is represented by a list of *join point abstractions*. A join point abstraction records all the information in a join point that is needed for *advice binding*, i.e., advice matching plus advice parameter 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 is represented by a five-tuple surrounded by half-moon brackets, ⦇…⦈. It consists of the following parts (most of which are optional and are replaced with "–" when omitted):

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

Figure 8: The Join Point Stack

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

Figure 9: Additional Expression Forms for the Operational Semantics of MiniMAO$_1$

— 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, used for parameter binding by this pointcut descriptors;

— an optional name indicating the name of the method called or executed at the join point, used for pattern matching in call and execution pointcut descriptors;

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

— an optional 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.

3.2.2 New Expression Forms

As with MiniMAO$_0$, the operational semantics of MiniMAO$_1$ includes additional expression forms to represent the state of the abstract machine. Figure 9 shows the three additional expression forms used.

The first additional expression form, joinpt, reifies join points of a program evaluation into the expression syntax. This reification allows advice binding to be handled by a single rule in the operational semantics instead of using different rules for each sort of join point. A joinpt expression consists of a join point abstraction followed by a sequence of expressions representing the 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. (In a language that included after advice, a term under v (where v is a value) could also serve as an indication that any after advice matching the stack should be triggered.)

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. A chain expression can be viewed as the dynamic equivalent of a static proceed expression. Unlike proceed, a chain expression records all of the subsequent advice that matched the original 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 aspect's 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. (No previous formalization of AspectJ has faithfully modeled this binding semantics for target and args.) We give examples of binding terms in Section 3.2.5 on page 24.

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.

3.2.3 Evaluation Rules for MiniMAO$_1$

Next we give an intuitive description of the new evaluation rules in MiniMAO$_1$. These rules are given in Figure 10 on the following page. The example evaluations in Section 3.2.5 on page 24 illustrate the rules.

We add new evaluation context rules to handle the joinpt, under, and chain expressions. 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$_A$ and CALL$_B$ described below, that introduce join points and handle proceeding from advice respectively. We replace the EXEC rule similarly. This division exposes join points for call and execution to the evaluation rules. Just as virtual dispatch is a primitive operation in a Java virtual machine, our semantics models advice binding as a primitive operation on these exposed join points. This advice binding is done by the new BIND rule. The new ADVISE rule models advice execution, and an UNDER rule helps maintain the join point stack by recording when join point abstractions should be popped.

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, has

Evaluation contexts:

$$\mathbb{E} ::= \ldots \mid \text{joinpt } j(\, v \ldots \mathbb{E}\, e \ldots\,) \mid \text{under } \mathbb{E} \mid \text{chain } \bar{B}, j(\, v \ldots \mathbb{E}\, e \ldots\,)$$

Evaluation relation (additional and replacement rules):

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

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

$$\langle \mathbb{E}[(\, l\, (\, v_0, \ldots, v_n\,)\,)], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{joinpt } (\!|\text{exec}, v_0, m, l, \tau|\!)(\, v_0, \ldots, v_n\,)], J, S\rangle \quad \textsc{Exec}_\text{A}$$
where $l = \text{fun } m\langle var_0, \ldots, var_n\rangle.e : \tau$

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

$$\langle \mathbb{E}[\text{null}.m(\, v_1, \ldots, v_n\,)], J, S\rangle \hookrightarrow \langle \text{NullPointerException}, J, S\rangle \quad \textsc{NCall}_\text{A}$$

$$\langle \mathbb{E}[\text{chain } \bullet, (\!|\text{call}, -, m, -, \tau|\!)(\, \text{null}, v_1, \ldots, v_n\,)], J, S\rangle \hookrightarrow \langle \text{NullPointerException}, J, S\rangle \quad \textsc{NCall}_\text{B}$$

$$\langle \mathbb{E}[\text{joinpt } j(\, v_0, \ldots, v_n\,)], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{under chain } \bar{B}, j(\, v_0, \ldots, v_n\,)], j + J, S\rangle \quad \textsc{Bind}$$
where $adviceBind(j + J, S) = \bar{B}$

$$\langle \mathbb{E}[\text{chain } [\![b, loc, e, \llcorner, \lrcorner]\!] + \bar{B}, j(\, v_0, \ldots, v_n\,)], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{under } e'\{\!|\, loc / \, \text{this} \,|\!\}\{\!|(v_0, \ldots, v_n) / \, b \,|\!\}], j' + J, S\rangle \quad \textsc{Advise}$$
where $e' = \langle\!\langle e \rangle\!\rangle_{\bar{B}, j}$ and $j' = (\!|\text{this}, loc, -, -, -|\!)$

$$\langle \mathbb{E}[\text{under } v], J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J', S\rangle \quad \textsc{Under}$$
where $J = j + J'$, for some j

Figure 10: Changes to the Operational Semantics for MiniMAO$_1$

$$\frac{CT(a) = \text{aspect } a \{ \ldots \}}{a \preccurlyeq \text{Object}}$$

Figure 11: Additional Subtyping Rule for MiniMAO$_1$

the form $\langle loc, pcd, e, (t_1 \times \ldots \times t_n \to t), \tau \rangle$; in this 5-tuple $S(loc) = [a.F]$ is the aspect instance for a in the initial store. For a given aspect a, every 5-tuple in AT representing advice from a has the same location. The function type τ is the type of proceed expressions in e, derived from *pcd*. (In AspectJ, τ would be redundant, because the type of proceed expressions in AspectJ advice is derived from the advice signature. That is, $\tau = (t_1 \times \ldots \times t_n \to t)$. In MiniMAO$_1$ the type of proceed expressions is derived from the pointcut descriptor.)

The global class table, CT, is extended in MiniMAO$_1$ to also map aspect names to the aspect declarations. We extend the subtyping rules with a rule that all aspects are subtypes of Object, as shown in Figure 11. Treating aspect instances as regular objects allows the rules for field access to be applied uniformly for aspect and class instances. This treatment also matches the situation in AspectJ. We also extend the field lookup function, *fieldsOf*, with an additional rule for aspects as shown in Figure 12 on the next page.

Next we describe the new evaluation rules in more detail.

Splitting the Call Rule In object-oriented MiniMAO$_0$, a method call is evaluated by applying the CALL and EXEC rules in turn. In aspect-oriented MiniMAO$_1$, each of these steps is broken into a series of steps. This division lets use reason about (and prove properties about) advice binding and execution independent of the sort of join point that is advise. With this division the CALL step becomes:

— CALL$_A$: creates a call join point

— BIND: finds matching advice

— ADVISE: evaluates each piece of advice

— CALL$_B$: looks up a method and creates an application form

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

Create a Join Point The CALL$_A$ rule says that a method call expression with a non-null target evaluates to a joinpt expression. 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. The function type includes a target type in the first argument position. The function type is determined using a pair of auxiliary functions, *methodType* and *origType*, shown in Figure 12 on the following page.

The *methodType* function is similar to *methodBody* discussed above; it searches the class table for the method declaration and returns a function type. The *origType* function finds the type of the "most super" class of the target type that also declares the method m. The target type included in the call join point abstraction generated by CALL$_A$ 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.

Field lookup (additional rule):

$$\frac{CT(a) = \text{aspect } a \ \{ \ t_1 \ f_1 \ldots t_n \ f_n \ adv^* \ \}}{fieldsOf(a) = \{f_i \mapsto t_i \cdot i \in \{1..n\}\}}$$

Original declaration lookup:

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

Advice binding:

$$adviceBind : Stack \times Store \to \langle \mathscr{B} \times \mathscr{L} \times \mathscr{E} \times (\mathscr{T}^* \to \mathscr{T}) \times (\mathscr{T}^* \to \mathscr{T}) \rangle$$

$adviceBind(J, S) = \bar{B}$, where $\bar{B}$ is a smallest list satisfying

$$\forall \langle loc, pcd, e, \tau, \tau' \rangle \in AT \cdot ((matchPCD(J, pcd, S) = b \neq \bot) \implies [\![b, loc, e, \tau, \tau']\!] \in \bar{B})$$

Advice chaining:

$$\langle\!\langle - \rangle\!\rangle_{\bar{B}, j} : \mathscr{E} \to \mathscr{E}$$

$$\langle\!\langle e_0.\text{proceed}(\ e_1, \ldots, e_n\ ) \rangle\!\rangle_{\bar{B}, j} = \text{chain } \bar{B}, j(\ \langle\!\langle e_0 \rangle\!\rangle_{\bar{B}, j}, \langle\!\langle e_1 \rangle\!\rangle_{\bar{B}, j}, \ldots, \langle\!\langle e_n \rangle\!\rangle_{\bar{B}, j}\ )$$

For all other expression forms, the chaining operator is just applied recursively to every subexpression. For example, the definition of the chaining operator for field set is:

$$\langle\!\langle e.f{=}e' \rangle\!\rangle_{\bar{B}, j} = \langle\!\langle e \rangle\!\rangle_{\bar{B}, j}.f{=}\langle\!\langle e' \rangle\!\rangle_{\bar{B}, j}$$

Binding substitution:

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

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

In all other cases, binding substitution is undefined.

Figure 12: Auxiliary Functions for MiniMAO$_1$ Operational Semantics

Find Matching Advice The BIND rule is the only place in the language 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.

The rule uses the auxiliary function *adviceBind* to find the (possibly empty) list of advice matching the new join point stack and store. The *adviceBind* function applies the *matchPCD* function, described in Section 3.2.4, 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. For practical purposes some well-defined ordering is needed, but 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.

Evaluate Advice The ADVISE rule takes a chain expression with a non-empty list of advice and evaluates the first piece of advice. Before evaluating the advice the rule uses the $\langle\!\langle - \rangle\!\rangle_{\bar{B},j}$ auxiliary function to eliminate proceed expressions in the advice body. This "advice chaining" function rewrites all proceed expressions, replacing them with chain expressions carrying the remainder of the advice list $\bar{B}$, 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 Jagadeesan et al. [12], 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 3.2.5.

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 (see Figure 12 on the previous page). 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:

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

The $\mathsf{x} \mapsto \mathsf{loc2}$ in the binding term does not use data from the arguments $\langle$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—equivalent to the self reference stored on the call stack in a Java virtual machine—and wraps the result expression in an under expression, which records that the join point abstraction should be popped from the stack later.

Finish the Original Operation 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.

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.

A General Technique The technique used to convert the CALL rule from the MiniMAO$_0$ language 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.

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 language, such as field assignment. 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.

The Under Rule The UNDER rule is the simplest of the new evaluation rules. It just extracts the value from the under expression and pops one join point abstraction from the stack.

3.2.4 Pointcut Matching

This section describes the semantics for matching advice to join points in the execution of a program. We use an auxiliary function, *matchPCD*, that takes a join point stack, pointcut descriptor, and the current store (used to find object types). The function returns a binding term describing how the parameters of the advice should be bound, or else $\bot$ if the pointcut descriptor does not match. The most interesting typing issues in MiniMAO$_1$ arise in conjunction with the semantics of pointcut matching. These issues are discussed in this section and in Section 3.3.2 below.

Following Wand et al. [19], we define the *matchPCD* function using a boolean algebra over binding terms. Our binding terms, as described in Section 3.2.2 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 and the definition of *matchPCD* are given in Figure 13 on the following page. The terms of the algebra are drawn from the set $\mathscr{B}_\bot = \mathscr{B} \cup \{\bot\}$, where binding terms can be thought of as "true" and $\bot$ as "false". The operators in the algebra are conjunction ($\wedge$), disjunction ($\vee$), and complement ($\neg$). The complement of the 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, if $b \neq \bot$, then $b \vee r = b$, ignoring the value of r. One difference in our algebra, versus Wand et al. [19], is in the conjunction of two non-$\bot$ terms. Our language must consider the bindings from both terms, because we have more than one pointcut descriptor that can bind formal parameters. 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 (denoted $\sqcup$) that extends the shorter binding term if the two terms do not have the same number of elements. 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 $\bot$. The only

Boolean algebra of bindings (adapted from Wand et al. [19]):

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

$$b \wedge b' = b \sqcup b' \qquad \neg\bot = \langle -, - \rangle \qquad \neg b = \bot$$

Join of bindings:

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

$$(var \mapsto loc) \sqcup (var' \mapsto loc') = var \mapsto loc \qquad (var \mapsto loc) \sqcup - = var \mapsto loc \qquad - \sqcup (var' \mapsto loc') = var' \mapsto loc'$$

$$var \sqcup var' = var \qquad var \sqcup - = var \qquad - \sqcup var' = var' \qquad - \sqcup - = -$$

Pointcut descriptor matching:

$$matchPCD(\langle\!| k, \_, m, \_, t_0 \times \ldots \times t_p \to t |\!\rangle + J, \text{call}(\ u\ idPat(..)\ ), S) = \begin{cases} \langle -, - \rangle & \text{if } k = \text{call},\ t = u, \text{ and } m \in idPat \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD(\langle\!| k, \_, m, \_, t_0 \times \ldots \times t_p \to t |\!\rangle + J, \text{execution}(\ u\ idPat(..)\ ), S) = \begin{cases} \langle -, - \rangle & \text{if } k = \text{exec},\ t = u, \text{ and } m \in idPat \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD(\langle\!| \_, v, \_, \_, \_ |\!\rangle + J, \text{this}(\ t\ var\ ), S) = \begin{cases} \langle var \mapsto v, - \rangle & \text{if } v \neq \text{null},\ S(v) = [s\,.\,F], \text{ and } s \preccurlyeq t \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD(\langle\!| \_, -, \_, \_, \_ |\!\rangle + J, \text{this}(\ t\ var\ ), S) = matchPCD(J, \text{this}(\ t\ var\ ), S)$$

$$matchPCD(\langle\!| \_, \_, \_, \_, s_0 \times \ldots \times s_n \to s |\!\rangle + J, \text{target}(\ t\ var\ ), S) = \begin{cases} \langle -, var \rangle & \text{if } s_0 = t \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD(\langle\!| \_, \_, \_, \_, - |\!\rangle + J, \text{target}(\ t\ var\ ), S) = matchPCD(J, \text{target}(\ t\ var\ ), S)$$

$$matchPCD(\langle\!| \_, \_, \_, \_, t_0 \times \ldots \times t_p \to t |\!\rangle + J, \text{args}(\ u_1\ var_1, \ldots, u_n\ var_n\ ), S) = \begin{cases} \langle -, -, var_1, \ldots, var_n \rangle & \text{if } p = n \text{ and } \forall i \in \{1..n\} \cdot (t_i = u_i) \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD(J, pcd\ ||\ pcd', S) = matchPCD(J, pcd, S) \vee matchPCD(J, pcd', S)$$

$$matchPCD(J, pcd\ \&\&\ pcd', S) = matchPCD(J, pcd, S) \wedge matchPCD(J, pcd', S)$$

$$matchPCD(J, !\ pcd, S) = \neg\, matchPCD(J, pcd, S)$$

$$matchPCD(J, pcd, S) = \bot \text{ for any case not matched by the preceding rules}$$

Figure 13: Pointcut Descriptor Matching for MiniMAO$_1$

complications are to accommodate the multiple parameter binding forms. For example, this and target matching must be done without information on how many additional arguments might be bound by an args pointcut descriptor. Thus, the length of binding terms must be allowed to vary.

Call and Execution The call and execution rules only match 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. Because these pointcut descriptors do not bind formal parameters, a match is indicated by an empty binding term.

This Two rules are used to handle this pointcut descriptors. Together, these rules find the most recent join point abstraction that includes the optional self object location. Once found, the type of the object record in that location is checked. If it 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 $\bot$.

Target The target pointcut descriptor is handled similarly to this, but uses the target type from the join point instead. 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. [13]. 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.[3] Thus, for soundness the target pointcut descriptor must use exact type matching. If advice were not allowed to change the target object, then less restrictive target type matching could be used.

This restriction to exact type matching is not as severe as it may seem at first. This is because when the $\textsc{Call}_\textsc{A}$ rule generates the target type for its join point abstraction, it uses the type of the class declaring the "most super" 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 most super method also means that advice can be written to match a call to any method in a family of overriding methods. Unlike the $\textsc{Call}_\textsc{A}$ rule, the $\textsc{Exec}_\textsc{A}$ 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.

We are interested in investigating whether a more elaborate type system might permit more expressive pointcut matching while maintaining soundness. However, this is orthogonal to our concerns with modular reasoning and so we leave it for future work.

Args The args pointcut descriptor 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 simply appeal to the corresponding operators in the binding algebra. The definition of complement implies that $\neg\neg pcd \neq pcd$. 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 preceding rules evaluates to $\bot$. 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] 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.

3.2.5 Example Evaluations in MiniMAO$_1$

This section gives several example MiniMAO$_1$ programs and their evaluations. These examples help to illustrate several points mentioned above but do not break new ground. Thus, the reader who understands our formalism my skip the remainder of this section.

Calls in MiniMAO$_0$ vs. Unadvised Calls in MiniMAO$_1$ The first example compares the evaluation of method calls in MiniMAO$_0$ and MiniMAO$_1$. Consider the following program:

```
class Simple extends Object {
   Object f;
   Object m(Object arg) {
      this.f = arg
   }
}
new Simple().m(new Object())
```

Figure 14 on the next page shows the evaluation of this program in both MiniMAO$_0$ and MiniMAO$_1$. The evaluation on the left uses the operational semantics of MiniMAO$_0$. The one on the right uses that of MiniMAO$_1$. This illustrates the splitting of the CALL and EXEC rules into pairs with advice look up, by the BIND rule, on the inserted join points. Because this program includes no advice, the BIND rule creates chain expressions with empty advice lists and the ADVISE rule is never used. At the end of the MiniMAO$_1$ evaluation, the UNDER rules pop the join point stack.

Advice Binding The next example illustrates advice binding. The example code is given in Figure 15 on page 26. Below is the evaluation in MiniMAO$_1$. In the evaluation, the initial store is $S_0 = \{\text{locA} \mapsto [\text{Asp}.\{\text{f1} \mapsto \text{null}, \text{f2} \mapsto \text{null}\}]\}$. The illustrative part of this example is in the application of the BIND and ADVISE rules—the last two steps shown. In the BIND rule the binding term, b is $\langle -, \text{s}, \text{arg1}\rangle$, indicating that the target object will be bound to the formal parameter s and the argument to arg1. Figure 16 on page 26 shows the matching operation that yields this binding term. In the ADVISE rule the argument to the original method call, loc1, is substituted for arg1 in the advice body. The formal parameter s does not appear in the advice body and so the target object of the original call, loc0, is not bound. The advice never proceeds to the original method, as evidenced by the dropping of the chain expression in the application of the ADVISE rule.

$\langle$***new*** *Simple()*.m(**new** Object()), $\bullet, S_0\rangle$

$\hookrightarrow \langle \text{loc0.m}(\textbf{\textit{new}}\ \textit{Object()}), \bullet, S_1\rangle$ (NEW)

where $S_1 = \{\text{locA} \mapsto [\text{Asp}.\{\text{f1} \mapsto \text{null}, \text{f2} \mapsto \text{null}\}],$
$\text{loc0} \mapsto [\text{Simple}.\{\text{f} \mapsto \text{null}\}]\}$

$\hookrightarrow \langle \textit{loc0.m(loc1)}, \bullet, S_2\rangle$ (NEW)

where $S_2 = \{\text{locA} \mapsto [\text{Asp}.\{\text{f1} \mapsto \text{null}, \text{f2} \mapsto \text{null}\}],$
$\text{loc0} \mapsto [\text{Simple}.\{\text{f} \mapsto \text{null}\}],$
$\text{loc1} \mapsto [\text{Object}.\emptyset]\}$

$\hookrightarrow \langle$***joinpt*** $(\!|$***call***, −, *m*, −, *Simple* × *Object* → *Object*$|\!)$ *(loc0,loc1)*, $\bullet, S_2\rangle$ (CALL$_A$)

$\hookrightarrow \langle$**under** ***chain*** (BIND)

$[\![b,$ *locA,* ***this****.f1=arg1, Object* × *Simple* → *Object, Simple* × *Object* → *Object*$]\!]$,
$(\!|$***call***, −, *m*, −, *Simple* × *Object* → *Object*$|\!)$ *(loc0,loc1)*, $J_1, S_2\rangle$

where $b = \langle -, \text{s}, \text{arg1}\rangle$

$J_1 = (\!|\textbf{call}, -, \text{m}, -, -, \text{Simple} \times \text{Object} \to \text{Object}|\!)$

Evaluation in MiniMAO$_0$		Evaluation in MiniMAO$_1$	
$\langle$***new*** *Simple()*.m(**new** Object()), $\bullet$, $\emptyset\rangle$		$\langle$***new*** *Simple()*.m(**new** Object()), $\bullet$, $\emptyset\rangle$	
$\hookrightarrow \langle$loc0.m(***new*** *Object()*), $\bullet$, $S_0\rangle$	(NEW)	$\hookrightarrow \langle$loc0.m(***new*** *Object()*), $\bullet$, $S_0\rangle$	(NEW)
$\hookrightarrow \langle$*loc0.m(loc1)*, $\bullet$, $S_1\rangle$	(NEW)	$\hookrightarrow \langle$*loc0.m(loc1)*, $\bullet$, $S_1\rangle$	(NEW)
$\hookrightarrow \langle$***fun*** *m*$\langle$***this***,*arg*$\rangle$.***this***.*f=arg:*τ *(loc0,loc1)*, $\bullet$, $S_1\rangle$	(CALL)	$\hookrightarrow \langle$***joinpt*** j_1 *(loc0,loc1)*, $\bullet$, $S_1\rangle$	(CALL$_A$)
.		$\hookrightarrow \langle$**under** ***chain*** $\bullet$, j_1 *(loc0,loc1)*, j_1, $S_1\rangle$	(BIND)
.		$\hookrightarrow \langle$**under** ***fun*** *m*$\langle$***this***,*arg*$\rangle$.***this***.*f=arg:*τ *(loc0,loc1)*, j_1, $S_1\rangle$	(CALL$_B$)
$\hookrightarrow \langle$*loc0.f = loc1*, $\bullet$, $S_1\rangle$	(EXEC)	$\hookrightarrow \langle$**under** ***joinpt*** j_2 *(loc0,loc1)*, j_1, $S_1\rangle$	(EXEC$_A$)
.		$\hookrightarrow \langle$**under under** ***chain*** $\bullet$, j_2 *(loc0,loc1)*, $j_2 + j_1$, $S_1\rangle$	(BIND)
.		$\hookrightarrow \langle$**under under under** *loc0.f = loc1*, $j_3 + j_2 + j_1$, $S_1\rangle$	(EXEC$_B$)
$\hookrightarrow \langle$loc1, $\bullet$, $S_2\rangle$	(SET)	$\hookrightarrow \langle$**under under** ***under*** *loc1*, $j_3 + j_2 + j_1$, $S_2\rangle$	(SET)
		$\hookrightarrow \langle$**under** ***under*** *loc1*, $j_2 + j_1$, $S_2\rangle$	(UNDER)
		$\hookrightarrow \langle$***under*** *loc1*, j_1, $S_2\rangle$	(UNDER)
		$\hookrightarrow \langle$loc1, $\bullet$, $S_2\rangle$	(UNDER)

where $S_0 = \{\text{loc0} \mapsto [\text{Simple}\centerdot\{\text{f} \mapsto \text{null}\}]\}$,

$S_1 = \{\text{loc0} \mapsto [\text{Simple}\centerdot\{\text{f} \mapsto \text{null}\}], \text{loc1} \mapsto [\text{Object}\centerdot\emptyset]\}$,

$\tau = \text{Simple} \times \text{Object} \to \text{Object}$,

$S_2 = \{\text{loc0} \mapsto [\text{Simple}\centerdot\{\text{f} \mapsto \text{loc1}\}], \text{loc1} \mapsto [\text{Object}\centerdot\emptyset]\}$,

$j_1 = (\!|\textbf{call}, -, \text{m}, -, \tau|\!)$,

$j_2 = (\!|\textbf{exec}, -, \text{m}, \textbf{fun}\ \text{m}\langle\textbf{this},\text{arg}\rangle.\textbf{this}.\text{f=arg:}\tau, \tau|\!)$, and

$j_3 = (\!|\textbf{this}, \text{loc0}, -, -, -|\!)$.

Figure 14: Comparison of Evaluation in MiniMAO$_0$ and MiniMAO$_1$

```
aspect Asp {
   Object f1;
   Object around(Object arg1, Simple s) :
      call(Object m(..)) && args(Object arg1) && target(Simple s)
   {
      this.f1 = arg1;
   }
}

class Simple extends Object {class Simple extends Object {
   Object f;
   Object m(Object arg) {
      this.f = arg
   }
}
new Simple().m(new Object())
```

Figure 15: A Sample Program Showing Advice Binding

$$
\begin{aligned}
&\mathit{matchPCD}(\langle\!|\mathbf{call}, -, \mathrm{m}, -, -, \mathrm{Simple} \times \mathrm{Object} \to \mathrm{Object}|\!\rangle,\\
&\qquad \mathbf{call}(\mathrm{Object\ m(..)})\ \&\&\ \mathbf{args}(\mathrm{Object\ arg1})\ \&\&\ \mathbf{target}(\mathrm{Simple\ s}), S_2)\\
=\ & \mathit{matchPCD}(\langle\!|\mathbf{call}, -, \mathrm{m}, -, -, \mathrm{Simple} \times \mathrm{Object} \to \mathrm{Object}|\!\rangle, \mathbf{call}(\mathrm{Object\ m(..)}), S_2)\\
&\wedge \mathit{matchPCD}(\langle\!|\mathbf{call}, -, \mathrm{m}, -, -, \mathrm{Simple} \times \mathrm{Object} \to \mathrm{Object}|\!\rangle, \mathbf{args}(\mathrm{Object\ arg1}), S_2)\\
&\wedge \mathit{matchPCD}(\langle\!|\mathbf{call}, -, \mathrm{m}, -, -, \mathrm{Simple} \times \mathrm{Object} \to \mathrm{Object}|\!\rangle, \mathbf{target}(\mathrm{Simple\ s}), S_2)\\
=\ & \langle -, -\rangle \sqcup \langle -, -, \mathrm{arg1}\rangle \sqcup \langle -, \mathrm{s}\rangle\\
=\ & \langle -, -, \mathrm{arg1}\rangle \sqcup \langle -, \mathrm{s}\rangle\\
=\ & \langle -, \mathrm{s}, \mathrm{arg1}\rangle
\end{aligned}
$$

Figure 16: Sample Derivation of Pointcut Descriptor Matching

$$
\begin{aligned}
&\hookrightarrow \langle \mathbf{under\ under}\ \mathit{locA.f1{=}loc1}, J_2, S_2\rangle && \text{(ADVISE)}\\
&\qquad\qquad \text{where } J_2 = \langle\!|\mathbf{this}, \mathrm{locA}, -, -, -|\!\rangle + J_1\\
&\hookrightarrow \ldots
\end{aligned}
$$

We omit the remaining steps of the evaluation because similar steps have been shown above.

Advice Chaining The next example illustrates how multiple pieces of advice may bind to a single join point. It also shows how proceed expressions are converted by the $\langle\!\langle - \rangle\!\rangle_{\bar{B},j}$ auxiliary function. We give the full program listing in Figure 17 on the following page, but only describe the advice chaining part of the evaluation in detail.

After looking up advice for the method call in this program, the BIND rule produces an expression that contains a subexpression like the following:

$$
\begin{aligned}
&\mathbf{chain}\ [\![\langle -, \mathrm{s1}, \mathrm{arg1}\rangle, \mathrm{locA}, \mathbf{this}.\mathrm{f1{=}s1}.\mathbf{proceed}(\mathrm{arg1}), \tau, \tau 2]\!]\\
&\qquad + [\![\langle -, \mathrm{s2}, \mathrm{arg2}\rangle, \mathrm{locA}, \mathbf{this}.\mathrm{f2{=}s2}.\mathbf{proceed}(\mathrm{arg2}), \tau, \tau 2]\!],\\
&\qquad\quad \langle\!|\mathbf{call}, -, \mathrm{m}, -, \tau 2|\!\rangle\ \ (\mathrm{loc0}, \mathrm{loc1})
\end{aligned}
$$

where we assume appropriate values for the store and the type meta-variables, τ and $\tau 2$, but omit those details. This expression is evaluated by the ADVISE rule, which applies the advice chaining function to the

```
aspect Asp {
   Object f1;
   Object f2;

   Object around(Simple s1, Object arg1) :
         call(Object m(..)) && target(Simple s1) && args(Object arg1)
   {
      this.f1 = s1.proceed(arg1);
   }

   Object around(Simple s2, Object arg2) :
         call(Object m(..)) && target(Simple s2) && args(Object arg2)
   {
      this.f2 = s2.proceed(arg2);
   }
}

class Simple extends Object {class Simple extends Object {
   Object f;
   Object m(Object arg) {
      this.f = arg
   }
}

new Simple().m(new Object())
```

Figure 17: A Sample Program Showing Advice Chaining

body of the first advice in the chain's advice list:

$$\langle\!\langle \textbf{this}.\text{f1=s1}.\textbf{proceed}(\text{arg1})\rangle\!\rangle_{[\![\langle -,\text{s2},\text{arg2}\rangle,\ \text{locA},\ \textbf{this}.\text{f2=s2}.\textbf{proceed}(\text{arg2}),\ \tau,\ \tau 2]\!],\ (\textbf{call},\ -,\ \text{m},\ -,\ \tau 2)}$$

The function replaces the proceed expression with a chain expression, yielding:

$$\textbf{this}.\text{f1=}\textbf{chain}\ [\![\langle -,\text{s2},\text{arg2}\rangle,\ \text{locA},\ \textbf{this}.\text{f2=s2}.\textbf{proceed}(\text{arg2}),\ \tau,\ \tau 2]\!],\ (\!|\textbf{call},\ -,\ \text{m},\ -,\ \tau 2|\!)\ \ (\text{s1},\ \text{arg1})$$

Finally, the ADVISE rule substitutes for this and the formal parameters, and adds an under expression yielding:

$$\begin{array}{l}\textbf{under}\ \text{locA.f1} = \\ \quad \textbf{chain}\ [\![\langle -,\text{s2},\text{arg2}\rangle,\ \text{locA},\ \textbf{this}.\text{f2=s2}.\textbf{proceed}(\text{arg2}),\ \tau,\ \tau 2]\!],\ (\!|\textbf{call},\ -,\ \text{m},\ -,\ \tau 2|\!)\ \ (\text{loc0},\ \text{loc1})\end{array}$$

The next evaluation step is also by ADVISE and reduces the chain expression, exhausting the advice list, and yielding the expression:

$$\begin{array}{l}\textbf{under}\ \text{locA.f1} = \\ \quad (\textbf{under}\ \text{locA.f2} = \textbf{chain}\ \bullet,\ (\!|\textbf{call},\ -,\ \text{m},\ -,\ \tau 2|\!)\ \ (\text{loc0},\ \text{loc1}))\end{array}$$

The last chain expression has an empty advice list. It will be evaluated by the CALL_B rule, causing evaluation to proceed to the originally called method. Although the target object was not changed in this example, either piece of advice could have used a different first argument for its proceed call. The effect of this would be to replace loc0 in the above expression with the location of the new target object. Because the CALL_B rule uses that argument position for method lookup, changing the target object at a call join point will affect method lookup.

This Binding vs. Target Binding Our final example illustrates the differences between parameter binding for this and target pointcut descriptors in MiniMAO_1. Recall that our semantics for proceed with respect to the this pointcut descriptor differs from AspectJ's. AspectJ treats both this- and target-bound arguments like target-bound arguments in MiniMAO_1. That is, AspectJ allows advice to change the value bound by the this pointcut descriptor in subsequent advice. As discussed in above, our treatment of this is intended to reduce the interaction of aspects.

Besides contrasting the this and target pointcut descriptors, the example also uses both call and execution advice. Figure 18 on the next page gives the sample program.

Below is the evaluation in MiniMAO_1. In the evaluation, the initial store is $S_0 = \{\text{locA} \mapsto [\text{Asp}\,.\,\emptyset]\}$. For conciseness, the values of the stores and the derivation of the binding terms are left as exercises for the reader. We write under^n to indicate n instances of the keyword under. Interesting parts of the evaluation are noted along the way.

$$\begin{array}{ll}
\langle \textbf{\textit{new}}\ \textit{Super()}.\text{run}(), \bullet, S_0\rangle & \\
\hookrightarrow \langle \textit{loc0.run()}, \bullet, S_1\rangle & (\text{NEW}) \\
\hookrightarrow \langle \textbf{\textit{joinpt}}\ (\!|\textbf{\textit{call}},-,\textit{run},-,\tau 0|\!)\ \textit{(loc0)}, \bullet, S_1\rangle & (\text{CALL}_\text{A}) \\
\qquad \text{where } \tau 0 = \text{Super}\rightarrow\text{Object} & \\
\hookrightarrow \langle \textbf{under}\ \textbf{\textit{chain}}\ \bullet,\ (\!|\textbf{\textit{call}},-,\textit{run},-,\tau 0|\!)\ \textit{(loc0)}, J_0, S_1\rangle & (\text{BIND}) \\
\qquad \text{where } J_0 = (\!|\textbf{call},-,\text{run},-,\tau 0|\!) & \\
\hookrightarrow \langle \textbf{under}\ \textit{(}\textbf{\textit{fun}}\ \textit{run}\langle\textbf{\textit{this}}\rangle.\textbf{\textit{this}}.\textit{m(}\textbf{\textit{new}}\ \textit{Super()):}\tau 0\ \textit{(loc0))}, J_0, S_1\rangle & (\text{CALL}_\text{B}) \\
\hookrightarrow \langle \textbf{under}\ \textbf{\textit{joinpt}}\ (\!|\textbf{\textit{exec}},\textit{loc0},\textit{run},\textbf{\textit{fun}}\ \textit{run}\langle\textbf{\textit{this}}\rangle.\textbf{\textit{this}}.\textit{m(}\textbf{\textit{new}}\ \textit{Super()):}\tau 0,\tau 0|\!)\ \textit{(loc0)}, J_0, S_1\rangle & (\text{EXEC}_\text{A}) \\
\hookrightarrow \langle \textbf{under}^2\ \textbf{\textit{chain}}\ \bullet,\ (\!|\textbf{\textit{exec}},\textit{loc0},\textit{run},\textbf{\textit{fun}}\ \textit{run}\langle\textbf{\textit{this}}\rangle.\textbf{\textit{this}}.\textit{m(}\textbf{\textit{new}}\ \textit{Super()):}\tau 0,\tau 0|\!)\ \textit{(loc0)}, J_1, S_1\rangle & (\text{BIND}) \\
\qquad \text{where } J_1 = (\!|\textbf{exec},\text{loc0},\text{run},\textbf{fun}\ \text{run}\langle\textbf{this}\rangle.\textbf{this}.\text{m}(\textbf{new}\ \text{Super}()):\tau 0,\tau 0|\!) + J_0 &
\end{array}$$

```
aspect Asp {
   // call advice
   Object around(Super caller, Super callee, Super arg) : call(Object m(..)) &&
      this(Super caller) && target(Super callee) && args(Super arg)
   {
      caller;          // these variable references just help illustrate the substitution behavior
      callee;
      new Sub().proceed(arg)          // changes target to subtype, affects method selection
   }

   // execution advice
   Object around(Super caller, Sub callee, Super arg) : execution(Object m(..)) &&
      this(Super caller) && target(Sub callee) && args(Super arg)
   {
      caller;          // these variable references just help illustrate the substitution behavior
      callee;
      new SubSub().proceed(arg)    // changes target to subtype, no effect on method selection
   }
}

class Super extends Object {
   Object run() {
      this.m(new Super())
   }

   Object m(Super arg) {
      arg
   }
}

class Sub extends Super {
   Object m(Super arg) {
      arg;
      this
   }
}

class SubSub extends Sub {
   Object m(Super arg) {
      this
   }
}

new Super().run();
```

Figure 18: A Sample Program Contrasting this vs. target Binding and call vs. execution Advice

$\hookrightarrow \langle \textbf{under}^3\ \text{loc0.m}(\textbf{\textit{new}}\ \textit{Super()}), J_2, S_1\rangle$ (EXEC$_B$)

where $J_2 = (\!|\textbf{this}, \text{loc0}, -, -, -|\!) + J_1$

$\hookrightarrow \langle \textbf{under}^3\ \textit{loc0.m(loc1)}, J_2, S_2\rangle$ (NEW)

$\hookrightarrow \langle \textbf{under}^3\ \textbf{\textit{joinpt}}\ (\!|\textbf{\textit{call}}, -, m, -, \tau 1|\!)\ \textit{(loc0,loc1)}, J_2, S_2\rangle$ (CALL$_A$)

where $\tau 1 = \text{Super}\times\text{Super}\rightarrow\text{Object}$

$\hookrightarrow \langle \textbf{under}^4$
$\textbf{\textit{chain}}\ [\![\langle \textit{caller}\mapsto \textit{loc0,callee,arg}\rangle, \textit{locA, (caller;callee;}\textbf{\textit{new}}\ \textit{Sub().}\textbf{\textit{proceed}}\textit{(arg)), } \tau 2, \tau 1]\!]$
$(\!|\textbf{\textit{call}}, -, m, -, \tau 1|\!)\ \textit{(loc0,loc1)}, J_3, S_2\rangle$

(BIND)

where $\tau 2 = \text{Super}\times\text{Super}\times\text{Super}\rightarrow\text{Object}$
$J_3 = (\!|\textbf{call}, -, \text{m}, -, \tau 1|\!) + J_2$

The binding term above maps caller to the calling object's location, loc0, and records that callee and arg should be bound to the target and argument of the chain expression.

$\hookrightarrow \langle \textbf{under}^5\ \textit{(loc0;loc0;}\textbf{\textit{chain}}\ \bullet\ (\!|\textbf{\textit{call}}, -, m, -, \tau 1|\!)\ (\textbf{\textit{new}}\ \textit{Sub(), loc1))}, J_4, S_2\rangle$ (ADVISE)

where $J_4 = (\!|\textbf{this}, \text{locA}, -, -, -|\!) + J_3$

Now the proceed expression in the advice body has been replaced with a chain expression. The target argument to the chain is **new** Sub(), not the original target.

$\hookrightarrow \langle \textbf{under}^5\ \textbf{chain}\ \bullet\ (\!|\textbf{call}, -, \text{m}, -, \tau 1|\!)\ (\textbf{\textit{new}}\ \textit{Sub()}, \text{loc1}), J_4, S_2\rangle$ (SKIP×2)

$\hookrightarrow \langle \textbf{under}^5\ \textbf{\textit{chain}}\ \bullet\ (\!|\textbf{\textit{call}}, -, m, -, \tau 1|\!)\ \textit{(loc2, loc1)}, J_4, S_3\rangle$ (NEW)

$\hookrightarrow \langle \textbf{under}^5\ \textit{(}\textbf{\textit{fun}}\ m\langle\textbf{\textit{this}}, arg\rangle\textit{.(arg;}\textbf{\textit{this}}\textit{):}\tau 3\ \textit{(loc2, loc1))}, J_4, S_3\rangle$ (CALL$_B$)

where $\tau 3 = \text{Sub}\times\text{Super}\rightarrow\text{Object}$

Because the advice changed the target of the call to loc2, the fun term above came from Sub, not Super.

$\hookrightarrow \langle \textbf{under}^5\ \textbf{\textit{joinpt}}\ (\!|\textbf{\textit{exec}}, loc2, m, \textbf{\textit{fun}}\ m\langle\textbf{\textit{this}}, arg\rangle\textit{.(arg;}\textbf{\textit{this}}\textit{):}\tau 3, \tau 3|\!)\ \textit{(loc2, loc1)}, J_4, S_3\rangle$ (EXEC$_A$)

$\hookrightarrow \langle \textbf{under}^6$
$\textbf{\textit{chain}}\ [\![\langle \textit{caller}\mapsto \textit{loc2,callee,arg}\rangle, \textit{locA, (caller; callee; }\textbf{\textit{new}}\ \textit{SubSub().}\textbf{\textit{proceed}}\textit{(arg)), } \tau 4, \tau 3]\!]$,
$(\!|\textbf{\textit{exec}}, loc2, m, \textbf{\textit{fun}}\ m\langle\textbf{\textit{this}}, arg\rangle\textit{.(arg;}\textbf{\textit{this}}\textit{):}\tau 3, \tau 3|\!)\ \textit{(loc2, loc1)}, J_5, S_3\rangle$

(BIND)

where $\tau 4 = \text{Super}\times\text{Sub}\times\text{Super}\rightarrow\text{Object}$
$J_5 = (\!|\textbf{exec}, \text{loc2}, \text{m}, \textbf{fun}\ \text{m}\langle\textbf{this}, \text{arg}\rangle.(\text{arg};\textbf{this}):\tau 3, \tau 3|\!) + J_4$

$\hookrightarrow \langle \textbf{under}^7$
$\textit{(loc2; loc2; }\textbf{\textit{chain}}\ \bullet\textit{, (}\textbf{\textit{exec}}\textit{,loc2,m,}\textbf{\textit{fun}}\ m\langle\textbf{\textit{this}}, arg\rangle\textit{.(arg;}\textbf{\textit{this}}\textit{):}\tau 3, \tau 3|\!)\ \textit{(}\textbf{\textit{new}}\ \textit{SubSub(),loc1))}, J_6, S_3\rangle$

(ADVISE)

where $J_6 = (\!|\textbf{this}, \text{locA}, -, -, -|\!) + J_5$

Again the proceed expression in the new advice body—new SubSub().proceed(arg)—was replaced with a chain expression that has a new target object, new SubSub() instead of loc2.

$\hookrightarrow \langle \textbf{under}^7\ \textbf{chain}\ \bullet, (\!|\textbf{exec}, \text{loc2}, \text{m}, \textbf{fun}\ \text{m}\langle\textbf{this}, \text{arg}\rangle.(\text{arg};\textbf{this}):\tau 3, \tau 3|\!)\ (\textbf{\textit{new}}\ \textit{SubSub()}, \text{loc1}), J_6, S_3\rangle$

(SKIP×2)

$\hookrightarrow \langle \textbf{under}^7\ \textbf{\textit{chain}}\ \bullet\textit{, (}\textbf{\textit{exec}}\textit{,loc2,m,}\textbf{\textit{fun}}\ m\langle\textbf{\textit{this}}, arg\rangle\textit{.(arg;}\textbf{\textit{this}}\textit{):}\tau 3, \tau 3|\!)\ \textit{(loc3,loc1)}, J_6, S_4\rangle$ (NEW)

$\hookrightarrow \langle \textbf{under}^8\ \textit{(loc1;loc3)}, J_7, S_4\rangle$ (EXEC$_B$)

where $J_7 = (\!|\textbf{this}, \text{loc3}, -, -, -|\!) + J_6$

Unlike for the call advice above, even though the target object was changed to an instance of SubSub, the already selected method body was used when proceeding to the code under the exec join point.

$\hookrightarrow \langle \textbf{under}^8\ \text{loc3}, J_7, S_4\rangle$ (SKIP)

$\hookrightarrow \langle \text{loc3}, \bullet, S_4\rangle$ (UNDER×8)

3.3 Static Semantics of MiniMAO$_1$

We conclude our description of the MiniMAO$_1$ formalism by giving its static semantics. As stated in the introduction, for our purposes it is sufficient that MiniMAO$_1$–unlike AspectJ–have a safe static type system. Our design is based on finding a simple static type system that is safe. To do this, we have sacrificed more expressiveness than might be strictly necessary.

Figure 19 on the following page and Figure 21 on page 34 give the additional rules for the static semantics of MiniMAO$_1$. All of the rules from MiniMAO$_0$ are used unchanged.

For typing MiniMAO$_1$, we extend the domain of Γ to include the keyword proceed, and its range to include function types. That is, for the static semantics:

$$\Gamma : (\mathcal{V} \cup \{\mathsf{this}, \mathsf{proceed}\}) \to (\mathcal{T} \cup (\mathcal{T}^* \to \mathcal{T}))$$

This lets us use the type environment to record the type of an advised method so that proceed expressions in the body of advice may be assigned the appropriate type.

3.3.1 Declaration and Expression Typing Rules

The T-ASP rule says that an aspect declaration is well typed if all of its advice declarations are well typed.

The T-ADV rule says that 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. The "$\sqcup$" in the hypothesis indicates that we do not care about the type bound by a this pointcut descriptor here. The pointcut descriptor must also specify bindings for all of the formal parameters of the advice. These requirements are embodied in the pointcut descriptor typing, $pcd : \sqcup \,.\, u_0 \,.\, \langle u_1, \ldots, u_p \rangle \,.\, u \,.\, V \,.\, V$, which is discussed in Section 3.3.2 below. The body of the advice is typed in an environment that gives each formal its declared type, gives this the aspect type, and gives proceed the type of the code under the join point matched by the advice. 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. However, this means that advice that both specializes the return type and returns the result of calling proceed, like:

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

is not well typed if A is a proper subtype of B. In the above example, the proceed expression has type B, which is not a subtype of the declared return type of the advice. Wand et al. [19, §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.

There are four new typing rules for expressions in MiniMAO$_1$. Only the first, T-PROC, is used in the static typing of programs. The other three arise in the subject reduction proof to handle expression forms that are only introduced by the evaluation rules.

The T-PROC rule types proceed expressions. A proceed expression is well typed if its argument expressions are subtypes of the required types as recorded in the type environment. The type of the proceed expression is also taken from the type environment.

The T-UNDER rule says that an under expression is well typed if its contained expression is well typed. The type of the under expression is just that of the contained expression.

Aspect typing:

$$\textsc{T-Asp}\quad \frac{\forall i \in \{1..p\} \cdot \vdash adv_i \text{ OK in } a}{\vdash \text{aspect } a \{ \mathit{field}_1 \ldots \mathit{field}_n \; \mathit{adv}_1 \ldots \mathit{adv}_p \} \text{ OK}}$$

Advice typing:

$$\textsc{T-Adv}\quad \frac{\begin{array}{c} \mathit{var}_1 : t_1, \ldots, \mathit{var}_n : t_n \vdash \mathit{pcd} : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_p \rangle \,.\, u \,.\, V \,.\, V \\ V = \{\mathit{var}_1, \ldots, \mathit{var}_n\} \qquad \mathit{var}_1 : t_1, \ldots, \mathit{var}_n : t_n, \text{this} : a, \text{proceed} : (u_0 \times \ldots \times u_p \to u) \vdash e : s \qquad s \preccurlyeq t \preccurlyeq u \end{array}}{\vdash t \text{ around}(\; t_1 \; \mathit{var}_1, \ldots, t_n \; \mathit{var}_n \;) : \; \mathit{pcd} \; \{ \; e \; \} \text{ OK in a}}$$

Expression typing:

$$\textsc{T-Proc}\quad \frac{\begin{array}{c} \forall i \in \{0..n\} \cdot \Gamma \vdash e_i : u_i \\ \Gamma(\text{proceed}) = t_0 \times \ldots \times t_n \to t \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i \end{array}}{\Gamma \vdash e_0.\text{proceed}(\; e_1, \ldots, e_n \;) : t} \qquad \textsc{T-Under}\quad \frac{\Gamma \vdash e : t}{\Gamma \vdash \text{under } e : t}$$

$$\textsc{T-Chain}\quad \frac{\begin{array}{c} \forall i \in \{0..n\} \cdot \Gamma \vdash e'_i : u'_i \qquad \forall i \in \{0..n\} \cdot u'_i \preccurlyeq t_i \\ \forall i \in \{1..p\} \cdot \Gamma, \text{this} : \Gamma(\mathit{loc}), \text{proceed} : \tau, \mathit{typeBind}(\Gamma, b_i, (t_0, \ldots, t_n)) \vdash e_i : s'_i \\ \forall i \in \{1..p\} \cdot \Gamma \vdash b_i \text{ OK} \qquad \forall i \in \{1..p\} \cdot s'_i \preccurlyeq t \qquad \tau = t_0 \times \ldots \times t_n \to t \end{array}}{\Gamma \vdash \text{chain } [\![b_i, \mathit{loc}_i, e_i, \tau', \tau]\!]_{i \in \{1..p\}}, (\!| \_, \_, \_, \_, \tau |\!)(\; e'_0, \ldots, e'_n \;) : t}$$

$$\textsc{T-Join}\quad \frac{\forall i \in \{0..n\} \cdot \Gamma \vdash e_i : u_i \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i \qquad (v_{opt} = \mathit{loc}) \implies (\mathit{loc} \in \mathit{dom}(\Gamma))}{\Gamma \vdash \text{joinpt } (\!| \_, v_{opt}, \_, \_, (t_0 \times \ldots \times t_n \to t) |\!)(\; e_0, \ldots, e_n \;) : t}$$

Binding typing:

$$\textsc{T-Bind}\quad \frac{\begin{array}{c} (\alpha = \mathit{var} \mapsto v) \implies (\mathit{var} \notin V \setminus \mathit{var}) \qquad \forall i \in \{0..n\} \cdot (\beta_i = \mathit{var}) \implies (\mathit{var} \notin V \setminus \{\beta_i\}) \\ \forall \mathit{var} \in V \cdot (V \notin \mathit{dom}(\Gamma)) \qquad V = \mathit{var}(b) \qquad b = \langle \alpha, \beta_0, \ldots, \beta_n \rangle \end{array}}{\Gamma \vdash b \text{ OK}}$$

$$\text{where } \mathit{var}(\langle \alpha, \beta_0, \ldots, \beta_n \rangle) = \begin{cases} \{\mathit{var}\} \cup \{\beta_i \cdot i \in \{0..n\}, \beta_i \neq -\} & \text{if } \alpha = \mathit{var} \mapsto v \\ \{\beta_i \cdot i \in \{0..n\}, \beta_i \neq -\} & \text{otherwise} \end{cases}$$

Figure 19: Additions to the Static Semantics for MiniMAO$_1$

$$\mathit{typeBind}(\Gamma, \langle \mathit{var} \mapsto \mathit{loc}, \beta_0, \ldots, \beta_n \rangle, \langle t_0, \ldots, t_p \rangle) = \mathit{var} : \Gamma(\mathit{loc}), (\mathit{var}_i : t_i)_{i \in \{0..n\} \cdot \beta_i = \mathit{var}_i} \text{ if } n \leq p$$
$$\mathit{typeBind}(\Gamma, \langle -, \beta_0, \ldots, \beta_n \rangle, \langle t_0, \ldots, t_p \rangle) = (\mathit{var}_i : t_i)_{i \in \{0..n\} \cdot \beta_i = \mathit{var}_i} \text{ if } n \leq p$$
$$\mathit{typeBind}((\Gamma, \langle \alpha, \beta_0, \ldots, \beta_n \rangle, \langle t_0, \ldots, t_p \rangle) \text{ is undefined if } n > p$$

Figure 20: Binding for Type Environments

The most complex of the typing rules is T-CHAIN. This rule is not used in the static typing of programs, but arises in the subject reduction proof to handle chain expressions introduced by the evaluation rules. Our use of chain and joinpt expressions in the semantics of MiniMAO$_1$ allows advice binding to be localized in a single evaluation rule, and to be separated from advice execution.. The necessary trade-off is the complexity of the T-CHAIN rule, which ensures the advice bound to a join point is well-behaved.

The first two hypotheses of T-CHAIN require that the argument expressions are subtypes of the types expected for the code under the join point. The last hypothesis is just a side condition on τ. The remaining hypotheses ensure the each piece of advice in the advice list satisfies the following conditions:

— The advice's binding term is well formed according to the T-BIND rule, which ensures that only fresh variables are bound and no variable is bound more than once.

— The advice's body expression is a subtype of the return type of the join point abstraction. This is also the type given to the entire chain expression. The typing of the body expression uses an auxiliary function, *typeBind*, defined in Figure 20, that converts the type environment, the binding term, and the argument types into a type environment. This type environment corresponds to the substitution defined by the binding term (see Figure 12 on page 19).

Finally, the T-JOIN rule types joinpt expressions. It simply ensures that all of the arguments are subtypes of the argument types in the join point abstraction. It also checks that any location given in the join point abstraction is valid in the type environment.

3.3.2 Pointcut Descriptor Typing Rules

The rules for typing pointcut descriptors are shown in Figure 21 on the following page. These rules 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. This is also lifted to type sequences. The pointcut descriptor typing judgment, $\Gamma \vdash \mathit{pcd} : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, \hat{u}'' \,.\, V \,.\, V'$, gives:

— $\hat{u}$, the this type for any code under a join point matched by this pointcut descriptor, or $\bot$ if the information cannot be determined from the pointcut descriptor;

— $\hat{u}'$, the target type for any code under a join point matched by this pointcut descriptor, or $\bot$ if the information cannot be determined from the pointcut descriptor;

— U, the argument types for any code under a join point matched by this pointcut descriptor, or $\bot$ if the information cannot be determined from the pointcut descriptor;

— $\hat{u}''$, the return type for any code under a join point matched by this pointcut descriptor, or $\bot$ if the information cannot be determined from the pointcut descriptor;

— V, the set of variables that would definitely be bound by the pointcut descriptor at a matched join point; and

— V', the set of variables that might be bound by the pointcut descriptor at a matched join point.

Pointcut typing:

$$U \in \mathcal{T}^* \cup \{\bot\} \qquad \hat{u} \in \mathcal{T} \cup \{\bot\} \qquad V \in \mathcal{P}(\mathcal{V}) \qquad \Gamma \vdash pcd : \hat{u}.\hat{u}.U.\hat{u}.V.V$$

T-CallPCD

$$\frac{}{\Gamma \vdash \mathsf{call}(\ t\ idPat(..)\ ) : \bot.\bot.\bot.t.\emptyset.\emptyset}$$

T-ExecPCD

$$\frac{}{\Gamma \vdash \mathsf{execution}(\ t\ idPat(..)\ ) : \bot.\bot.\bot.t.\emptyset.\emptyset}$$

T-ThisPCD

$$\frac{\Gamma(var) = t}{\Gamma \vdash \mathsf{this}(\ t\ var\ ) : t.\bot.\bot.\bot.\{var\}.\{var\}}$$

T-TargPCD

$$\frac{\Gamma(var) = t}{\Gamma \vdash \mathsf{target}(\ t\ var\ ) : \bot.t.\bot.\bot.\{var\}.\{var\}}$$

T-ArgsPCD

$$\frac{\forall i \in \{1..n\} \cdot (\Gamma(var_i) = t_i) \qquad \forall i \in \{1..n\} \cdot (\forall j \in \{1..n\} \setminus \{i\} \cdot (var_i \neq var_j))}{\Gamma \vdash \mathsf{args}(\ t_1\ var_1, \ldots, t_n\ var_n\ ) : \bot.\bot.\langle t_1, \ldots, t_n \rangle.\bot.\{var_1, \ldots, var_n\}.\{var_1, \ldots, var_n\}}$$

T-UnionPCD

$$\frac{\Gamma \vdash pcd_1 : \hat{u}.\hat{u}'.U.\hat{u}''.V_1.V_1' \qquad \Gamma \vdash pcd_2 : \hat{u}.\hat{u}'.U.\hat{u}''.V_2.V_2' \qquad V = V_1 \cap V_2 \qquad V' = V_1' \cup V_2'}{\Gamma \vdash pcd_1 \ ||\ pcd_2 : \hat{u}.\hat{u}'.U.\hat{u}''.V.V'}$$

T-NegPCD

$$\frac{\Gamma \vdash pcd : \hat{u}.\hat{u}'.U.\hat{u}''.V.V'}{\Gamma \vdash !\ pcd : \bot.\bot.\bot.\bot.\emptyset.\emptyset}$$

T-IntPCD

$$\frac{\Gamma \vdash pcd_1 : \hat{u}_1.\hat{u}_1'.U_1.\hat{u}_1''.V_1.V_1' \qquad \Gamma \vdash pcd_2 : \hat{u}_2.\hat{u}_2'.U_2.\hat{u}_2''.V_2.V_2' \qquad \hat{u} = \hat{u}_1 \sqcup \hat{u}_2 \qquad \hat{u}' = \hat{u}_1' \sqcup \hat{u}_2' \qquad U = U_1 \sqcup U_2 \qquad \hat{u}'' = \hat{u}_1'' \sqcup \hat{u}_2'' \qquad V_1' \cap V_2' = \emptyset \qquad V = V_1 \cup V_2 \qquad V' = V_1' \cup V_2'}{\Gamma \vdash pcd_1\ \&\&\ pcd_2 : \hat{u}.\hat{u}'.U.\hat{u}''.V.V'}$$

$$\hat{u} \sqcup \bot = \hat{u} \qquad \bot \sqcup \hat{u} = \hat{u} \qquad U \sqcup \bot = U \qquad \bot \sqcup U = U$$

Figure 21: Static Semantics of Pointcuts in MiniMAO$_1$

The two sets of variables 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).[4]

Given this form for the typing judgment, the rules for the primitive pointcut descriptors are mostly obvious. The only interesting bits are:

— the T-ThisPCD, T-TargPCD, and T-ArgsPCD rules verify that the type annotations for the bound parameters match the type of the formals as recorded in the type environment; and

— the second hypothesis of T-ArgsPCD ensures that no formal parameter is bound twice.

The typing rules for pointcut descriptor operators are more interesting. The T-UnionPCD rule requires that the two combined pointcut descriptors match join points where the type of the code under the join points is the same. This allows typing of any proceed expressions within the advice regardless of which pointcut in the disjunction was matched. The T-IntPCD rule requires that the combined pointcut descriptors specify types in disjoint positions. For example, if one of the combined pointcut descriptors specifies the argument types, then the other must not. This helps to ensure that no actual argument may be bound to multiple formal parameters. The T-IntPCD rule also requires that the sets of variables that may be bound by the two pointcut descriptors be disjoint; this helps to ensure that no formal is bound twice.

[4] This identification of the must- and may-bind sets only holds for the whole pointcut descriptor of well-typed advice. The sets may (and typically will) differ within subderivations of a pointcut descriptor typing judgment.

3.4 Meta-theory of MiniMAO$_1$

The meta-theory of MiniMAO$_1$ is essentially the same as for MiniMAO$_0$. The key difference in the theorems and lemmas is that we must deal with a non-empty initial store that contains aspect instances. Some complications arise in the proofs, which must be extended to deal with the new typing and evaluation rules. A few additional lemmas are needed to deal with advice binding and join points.

The statements of the supporting lemmas from MiniMAO$_0$ are unchanged. An updated proof of the substitution lemma is given in the appendix. Before stating the Subject Reduction theorem for MiniMAO$_1$, we give a few additional, necessary definitions and lemmas.

We define notions of a consistent stack and a valid store for a given MiniMAO$_1$ program. These definitions are used to ensure that all locations listed in the stack are bound in the store, and that the store contains an instance of every aspect declared in the program.

Definition 10 (Stack-Store Consistency)**.** A stack J and a store S are *consistent*, and we write $J \approx S$, if

$$\forall (\!|\_, loc, \_, \_, \_|\!) \in J \cdot loc \in dom(S).$$

Definition 11 (Store Validity)**.** Given a program P, we say that a store S is *valid* if both of the following hold:

1. $\forall \text{aspect } a \{ \ldots \} \in CT \cdot (\exists loc \in \mathscr{L} \cdot S(loc) = [a.F])$
2. $\exists \Gamma \cdot \Gamma \approx S$

We will need a lemma that relates advice binding to advice typing. This lemma is used in the subject reduction proof 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.

Lemma 12 (Binding Soundness)**.** *Let S be a valid store and $J = (\!|\ldots, t_0 \times \ldots \times t_n \to t|\!) + J'$ be a stack consistent with S. If $\bar{B} = adviceBind(J, S)$, then $\forall [\![b, loc, e, \tau, \tau']\!] \in \bar{B}$ the following conditions hold:*

1. *$\tau' = t_0 \times \ldots \times t_n \to t$,*
2. *$\emptyset \vdash b$ OK, and*
3. *for $\Gamma \approx S$ the judgment $\Gamma, this : \Gamma(loc), proceed : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e : t'$ holds for some $t' \preccurlyeq t$.*

Proof Sketch. Because a well-typed pointcut descriptor in MiniMAO$_1$ must consist of multiple primitive pointcut descriptors, it is difficult to prove the consequents of the lemma using a single inductive argument. Instead, we propose and prove a series of simpler subclaims. Each subclaim is proven via a structural induction on the pointcut type derivation. A well-typed pointcut descriptor that matches J satisfies the antecedents of all the subclaims, and the consequents of the subclaims imply the consequents of the lemma. Page 44 in the appendix gives the full proof. □

The following lemma states that advice chaining, replacing proceed expressions with chain expressions, does not affect typing judgments given the appropriate assumptions. These assumptions are essentially the hypotheses of the T-CHAIN rule, since advice chaining is performed by the ADVISE evaluation rule on chain expressions. This lemma is used for the ADVISE case in the subject reduction proof.

Lemma 13 (Advice Chaining)**.** *Let $\Gamma, proceed : \tau \vdash e : t$, $j = (\!|\_, \_, \_, \_, \tau|\!)$, $\tau = t_0 \times \ldots \times t_n \to t$, and for all $B = [\![b, loc, e', \tau', \tau]\!] \in \bar{B}$ let*

— $\Gamma, this : \Gamma(loc), proceed : \tau, typeBind(\Gamma, b, (t_0, \ldots, t_n)) \vdash e' : s'$,

— $\Gamma \vdash b$ *OK, and*

— $s' \preccurlyeq t$.

Then $\Gamma \vdash \langle\!\langle e \rangle\!\rangle_{\bar{B},j} : t$.

Finally, a simple lemma regarding join point abstractions is useful in the subject reduction and progress proofs.

Lemma 14 (Join Point Abstractions)**.** *In a MiniMAO$_1$ program evaluation, if a join point abstraction, j, appears in the expression of an evaluation triple, then one of the following hold:*

1. *Either $j = (\!|\mathsf{exec}, v, m, l, \tau|\!)$ and $l = \mathit{fun}\ m\langle var_0, \ldots, var_n\rangle.e : \tau$, or else*
2. *$j = (\!|\mathsf{call}, -, m, -, (t_0 \times \ldots \times t_n \to t)|\!)$ and $\mathit{methodType}(t_0, m) = t_1 \times \ldots \times t_n \to t$.*

Proof. Join point abstractions are not part of the user syntax of MiniMAO$_1$. By inspection, the only evaluation rules that can introduce new join point abstractions in the expression of an evaluation triple are EXEC$_\text{A}$ and CALL$_\text{A}$. Only EXEC$_\text{A}$ introduces exec join point abstractions, and these abstractions satisfy part 1 of the lemma. Only CALL$_\text{A}$ introduces call join point abstractions. By the definition of *origType*, these call join point abstractions satisfy the part 2 of the lemma. □

The subject reduction theorem for MiniMAO$_1$ is essentially the same as for MiniMAO$_0$, except that it requires and maintains stack-store consistency and stack validity.

Theorem 15 (Subject Reduction)**.** *Given a well typed MiniMAO$_1$ program, for an expression e, a valid store S, a stack J consistent with S, and a type environment Γ consistent with S, if $\Gamma \vdash e : t$ and $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$, then $J' \approx S'$, S' is valid, and there exist Γ' and t' such that $\Gamma' \approx S'$, $\Gamma' \vdash e' : t'$, and $t' \preccurlyeq t$.*

The progress theorem is slightly modified for MiniMAO$_1$, to include the validity of the store

Theorem 16 (Progress)**.** *For an expression e, a valid store S, a stack J consistent with S, and a type environment Γ consistent with S, if $\Gamma \vdash e : t$ then either:*

— *$e = loc$ and $loc \in dom(S)$,*

— *$e = null$, or*

— *one of the following hold:*

- *$\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$*
- *$\langle e, J, S\rangle \hookrightarrow \langle \mathit{NullPointerException}, J', S'\rangle$*
- *$\langle e, J, S\rangle \hookrightarrow \langle \mathit{ClassCastException}, J', S'\rangle$*

Finally, the soundness theorem must be updated to consider the initial, non-empty store.

Theorem 17 (Soundness)**.** *Given a program $P = decl_1 \ldots decl_n\ e$, 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 $loc \in dom(S)$,*

— *$v = null$,*

— *$v = \mathit{NullPointerException}$, or*

— *$v = \mathit{ClassCastException}$*

Proof. If e diverges then the claim holds. If e converges, then note that the empty stack is consistent with any store and the validity of S_0 implies the existence of an initial type environment consistent with S_0. The proof (by induction on the number of evaluation steps) is immediate from Theorem 15 (Subject Reduction) and Theorem 16 (Progress). □

4 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. [19] present a denotational semantics for an aspect-oriented language that includes temporal pointcut descriptors. Our use of an algebra of binding terms for advice matching is derived from their work. Their semantics binds all advice parameters at the join point instead of at each subsequent proceed expression. Their calculus is not object-oriented and so does not deal with the effects on method selection of changing the target object. Douence et al. [8] present a system for reasoning about temporal pointcut matching. They do not formalize advice parameter binding and do not include proceed in their language.

Jagadeesan et al. [12] present a multithreaded, class-based, aspect-oriented core language. They omit methods, using advice for all code abstraction. The lack of separate methods simplifies their semantics, but makes their language a poor fit for our planned studies of a verification logic for AspectJ-like languages. Also, their core language does not include the ability of advice to change the target object of an invocation. In an unpublished paper [13] they add a sound type system to their core language. Our type system is motivated by that work, but extends it to handle the separate this, target, and args binding forms and the ability of advice to change the target object.

Masuhara and Kiczales [16] give a Scheme-based model for an AspectJ-like language. They do not include around advice in their model. They do sketch how this 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 [7] 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. However, 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 directly, but can model it via the dynamic creation of advice. However, it is not obvious how such a model of state could be used for our planned study of aspect-oriented reasoning when aspects may interfere with the base program via the heap. Also, 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. Finally, their calculus is untyped and is not class-based.

Walker et al. [18] 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.

5 Conclusion

This paper described a core aspect-oriented language, MiniMAO$_1$. MiniMAO$_1$ is designed to serve as a core language for studying modular specification and verification in the aspect-oriented paradigm.

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.

MiniMAO$_1$ has a sound static type system, a first for a language with around advice that can change the target object when proceeding from advice. The key to proving soundness for MiniMAO$_1$ is a binding soundness lemma, that relates the type of pointcut descriptors to the type of code that they match.

6 Acknowledgements

We thank the anonymous referees of an earlier version of this paper, presented at the Workshop on Foundations of Aspect-Oriented Languages 2005, for their helpful comments. We also thank the *Science of Computer Programming* referees for helping us to focus the presentation of this work.

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. Iowa State University, Dept. of Computer Science.

[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] Luca Cardelli, editor. *ECOOP '03 - Object-Oriented Programming European Conference*, volume 2743 of *Lecture Notes in Comp. Sci.*, Darmstadt, Germany, 2003. Springer-Verlag.

[5] Curtis Clifton. *A design discipline and language features for modular reasoning in aspect-oriented programs*. PhD thesis, Iowa State University, 2005.

[6] Curtis Clifton and Gary T. Leavens. MiniMAO: Investigating the semantics of proceed. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL 2005 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2005*, pages 51–61, Chicago, Illinois, USA, 2005. Iowa State University, Dept. of Computer Science.

[7] Daniel S. Dantas and David Walker. Harmless advice. In *The 12th international workshop on Foundations of object-oriented languages*, Long Beach, California, 2005. ACM.

[8] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Reflection 2001*, volume 2192 of *Lecture Notes in Comp. Sci.*, Heidelberg, Germany, November 2001. Springer-Verlag.

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

[10] 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.

[11] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In A. Michael Berman, editor, *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 132–146, Denver, Colorado, USA, 1999. ACM Press.

[12] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In Cardelli [4], pages 54–73.

[13] 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 on February 1, 2004, 2004.

[14] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference*, volume 1241 of *Lecture Notes in Comp. Sci.*, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP '01 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Comp. Sci.*, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.

[16] Hidehiko Masuhara and Gregar Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Cardelli [4], pages 2–28.

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

[18] 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.

[19] 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.

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

7 Appendix

This appendix contains full proofs of the most interesting lemmas and theorems. Other proofs are completely standard and may be found in Clifton's dissertation [5].

7.1 MiniMAO$_0$

Lemma 2 (Substitution)**.** *If* $\Gamma, var_1 : t_1, \ldots, var_n : t_n \vdash e : t$ *and* $\forall i \in \{1..n\} \cdot \Gamma \vdash e_i : s_i$ *where* $s_i \preccurlyeq t_i$ *then* $\Gamma \vdash e\{\!|e_1 / var_1, \ldots, e_n / var_n|\!\} : s$ *for some* $s \preccurlyeq t$.

Proof. To lighten the notational load, let $\Gamma' = \Gamma, var_1 : t_1, \ldots, var_n : t_n$ and let $\{\!|\bar{e} / \overline{var}|\!\}$ represent $\{\!|e_1 / var_1, \ldots, e_n / var_n|\!\}$. The proof proceeds by structural induction on the derivation of $\Gamma \vdash e : t$ and by cases based on the last step in that derivation. The base cases are T-NEW, T-OBJ, T-NULL, T-LOC, and T-VAR. The first four of these cases are trivial: e has no variables and $s = t$.

In the T-VAR base case, $e = var$, and there are two subcases. If $var \notin \{var_1, \ldots, var_n\}$ then $\Gamma'(var) = \Gamma(var) = t$ and the claim holds. Otherwise, without loss of generality, let $var = var_1$. Then $e\{\!|\bar{e} / \overline{var}|\!\} = e_1$ and, by the assumptions of the lemma, $\Gamma \vdash e\{\!|\bar{e} / \overline{var}|\!\} : s_1$ and $s_1 \preccurlyeq t_1 = t$.

The remaining cases cover the induction step. The induction hypothesis is that the claim of the lemma holds for all sub-derivations of the derivation being considered.

Case 1—T-CALL. Here $e = e'_0.m(\,e'_1,\ldots,e'_p\,)$. The last type derivation step has the following form:

$$\frac{\begin{array}{c}\Gamma' \vdash e'_0 : u'_0 \qquad \forall i \in \{1..p\} \cdot \Gamma' \vdash e'_i : u'_i \\ \mathit{methodType}(u'_0, m) = u_1 \times \ldots \times u_p \to t \qquad \forall i \in \{1..p\} \cdot u'_i \preccurlyeq u_i\end{array}}{\Gamma' \vdash e : t}$$

Let $e''_i = e'_i\{\bar{e}/\overline{var}\}$ for $i \in \{0..p\}$, then $e\{\bar{e}/\overline{var}\} = e''_0.m(\,e''_1,\ldots,e''_p\,)$.

We show that $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$ by T-CALL. By the induction hypothesis, $\Gamma \vdash e''_0 : u''_0$, where $u''_0 \preccurlyeq u'_0$. And $\mathit{methodType}(u''_0, m) = \mathit{methodType}(u'_0, m)$ by the definitions of *methodType* and *override*. Also by the induction hypothesis $\forall i \in \{1..p\} \cdot \Gamma \vdash e''_i : u''_i$ and $u''_i \preccurlyeq u'_i$. Finally, $\forall i \in \{1..p\} \cdot u''_i \preccurlyeq u_i$ by transitivity and thus the claim holds.

Case 2—T-EXEC. Here $e = (\,\mathsf{fun}\ m\langle var'_0,\ldots,var'_p\rangle.e' : \tau\ (\,e'_0,\ldots,e'_p\,)\,)$, where $\tau = u'_0 \times \ldots \times u'_p \to t$. The last derivation step is:

$$\frac{\begin{array}{c}\Gamma, var'_0 : u'_0,\ldots,var'_p : u'_p \vdash e' : s' \qquad s' \preccurlyeq t \\ \forall i \in \{0..p\} \cdot \Gamma \vdash e'_i : u_i \qquad \forall i \in \{0..p\} \cdot u_i \preccurlyeq u'_i \qquad \tau = u'_0 \times \ldots \times u'_p \to t\end{array}}{\Gamma \vdash e : t}$$

As in the preceding case, let $e''_i = e'_i\{\bar{e}/\overline{var}\}$ for $i \in \{0..p\}$. Also let $e'' = e'\{\bar{e}/\overline{var}\}$, then

$$e\{\bar{e}/\overline{var}\} = (\,\mathsf{fun}\ m\langle var'_0,\ldots,var'_p\rangle.e'' : \tau\ (\,e''_0,\ldots,e''_p\,)\,).$$

By T-EXEC, the induction hypothesis, and transitivity of subtyping, $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$.

Case 3—T-GET. Here $e = e'.f$. The last derivation step is:

$$\frac{\Gamma' \vdash e' : u \qquad \mathit{fieldsOf}(u)(f) = t}{\Gamma' \vdash e'.f : t}$$

Now $e\{\bar{e}/\overline{var}\} = e'\{\bar{e}/\overline{var}\}.f$. By the induction hypothesis, $\Gamma \vdash e'\{\bar{e}/\overline{var}\} : u'$ where $u' \preccurlyeq u$. By the definition of *fieldsOf* and by the first hypothesis of T-CLASS, $\mathit{fieldsOf}(u')(f) = \mathit{fieldsOf}(u)(f) = t$. Therefore $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$ and the claim holds.

Case 4—T-Set. Here $e = (e'_1.f = e'_2)$ and the last step in the type derivation is:

$$\frac{\Gamma' \vdash e'_1 : u'_1 \qquad \mathit{fieldsOf}(u'_1)(f) = u \qquad \Gamma' \vdash e'_2 : t \qquad t \preccurlyeq u}{\Gamma' \vdash e'_1.f = e'_2 : t}$$

Now $e\{\bar{e}/\overline{var}\} = (e'_1\{\bar{e}/\overline{var}\}.f = e'_2\{\bar{e}/\overline{var}\})$. By the induction hypothesis $\Gamma \vdash e'_1\{\bar{e}/\overline{var}\} : u''_1$, $u''_1 \preccurlyeq u'_1$, $\Gamma \vdash e'_2\{\bar{e}/\overline{var}\} : t'$, $t' \preccurlyeq t$. By definition of *fieldsOf* and by the first hypothesis of T-CLASS, we have $\mathit{fieldsOf}(u''_1)(f) = \mathit{fieldsOf}(u'_1)(f) = u$. By transitivity $t' \preccurlyeq u$. Therefore, $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t'$, where $t' \preccurlyeq t$ and the claim holds.

Case 5—T-CAST. In this case, $e = \mathsf{cast}\ t\ e' : t$. Here the last derivation step is:

$$\frac{\Gamma' \vdash e : s}{\Gamma' \vdash \mathsf{cast}\ t\ e' : t}$$

By the induction hypothesis, $\Gamma \vdash e'\{\bar{e}/\overline{var}\} : s'$, and so $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$ by T-CAST.

Case 6—T-SEQ. In this case $e = e'_1;\ e'_2$ and the last step in the type derivation is:

$$\frac{\Gamma' \vdash e'_1 : s \qquad \Gamma' \vdash e'_2 : t}{\Gamma' \vdash e'_1;\ e'_2 : t}$$

Now $e\{\bar{e}/\overline{var}\} = e'_1\{\bar{e}/\overline{var}\};\ e'_2\{\bar{e}/\overline{var}\}$. By the induction hypothesis, $\Gamma \vdash e'_1\{\bar{e}/\overline{var}\} : s'$, $\Gamma \vdash e'_2\{\bar{e}/\overline{var}\} : t'$, and $t' \preccurlyeq t$. Therefore, $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t'$, $t' \preccurlyeq t$, and the claim holds.

Thus, for all possible derivations of $\Gamma' \vdash e : t$ we see that $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t'$ for some $t' \preccurlyeq t$. □

Theorem 7 (Subject Reduction)**.** *Given a well typed MiniMAO$_0$ program, for an expression e, a stack J, a store S, and a type environment Γ consistent with S, if $\Gamma \vdash e : t$ and $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$, then there exist Γ' and t' such that $\Gamma' \approx S'$, $\Gamma' \vdash e' : t'$, and $t' \preccurlyeq t$.*

Proof. The proof is by cases on the reduction step applied. Based on the reduction step we can construct a Γ' consistent with S' such that the claim is satisfied.

Case 1—NEW. In this case $e = \mathbb{E}[\mathsf{new}\ c()]$, $e' = \mathbb{E}[loc]$, $loc \notin dom(S)$, and $S' = S \oplus (loc \mapsto [c\,.\,F])$ where $F = \{f \mapsto \mathsf{null} \cdot f \in dom(fieldsOf(c))\}$.

Let $\Gamma' = \Gamma, loc : c$.

We will see that $\Gamma' \approx S'$. Because $loc \notin dom(S)$, $(\Gamma \approx S) \implies loc \notin dom(\Gamma)$ by part 2 of Definition 1 (Environment-Store Consistency) on page 10. Thus part 1 of the definition for $\Gamma' \approx S'$ holds for all $loc' \in \mathscr{L}$, $loc' \neq loc$. Now $S'(loc) = [c\,.\,F]$, $\Gamma'(loc) = c$, $dom(F) = dom(fieldsOf(c))$, $rng(F) = \{\mathsf{null}\} \subseteq dom(s) \cup \{\mathsf{null}\}$, and 1(d) holds vacuously. So part 1 of $\Gamma' \approx S'$ holds. Parts 2 and 3 hold because $\Gamma \approx S$, $loc \in dom(\Gamma')$, and $loc \in dom(S')$.

We will see that $\Gamma' \vdash \mathbb{E}[loc] : t$. By Lemma 3 (Environment Extension) on page 11 and $loc \notin dom(\Gamma)$, we have $\Gamma' \vdash \mathbb{E}[\mathsf{new}\ c()] : t$. Now $\Gamma' \vdash \mathsf{new}\ c() : c$ and $\Gamma' \vdash loc : c$, so by Lemma 5 (Replacement) on page 11, $\Gamma' \vdash \mathbb{E}[loc] : t$.

Case 2—CALL. Here $e = \mathbb{E}[loc.m(\ v_1, \ldots, v_n\ )]$, $e' = \mathbb{E}[(\ \mathsf{fun}\ m\langle \mathsf{this}, var_1, \ldots, var_n\rangle.e'' : \tau\ (\ loc, v_1, \ldots, v_n\ )\ )]$ (where $S(loc) = [u\,.\,F]$, $methodBody(u, m) = \mathsf{fun}\ m\langle \mathsf{this}, var_1, \ldots, var_n\rangle.e'' : \tau$, and $\tau = u' \times t_1 \times \ldots \times t_n \to u_m$), and $S' = S$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$.

We will see that $\Gamma \vdash e' : t$. $\Gamma \vdash e : t$ implies that $loc.m(\ v_1, \ldots, v_n\ )$ and all its subterms are well typed in Γ. By part 1(a) of $\Gamma \approx S$, $\Gamma \vdash loc : u$. By the definition of *methodBody*, $u \preccurlyeq u'$. Let $\Gamma \vdash v_i : u_i$ for all $i \in \{1..n\}$ and let $\Gamma \vdash loc.m(\ v_1, \ldots, v_n\ ) : t_m$. This last judgment must be by T-CALL with $methodType(u, m) = t_1 \times \ldots \times t_n \to t_m$ where $\forall\, i \in \{1..n\} \cdot u_i \preccurlyeq t_i$.

By the definition of *methodType*, rules T-CLASS and T-MET, and the definition of *override*, we have $(var_1 : t_1, \ldots, var_n : t_n, \mathsf{this} : u') \vdash e'' : u'_m$ where $u_m \preccurlyeq u'_m = t_m$. By Lemma 3 (Environment Extension) on page 11 (and appropriate alpha conversion of free variables in e''), $\Gamma, var_1 : t_1, \ldots, var_n : t_n, \mathsf{this} : u' \vdash e'' : u'_m$. So

$$\frac{\begin{array}{c}\Gamma, \mathsf{this} : u', var_1 : t_1, \ldots, var_n : t_n \vdash e'' : u'_m \qquad u'_m \preccurlyeq t_m \\ \Gamma \vdash loc : u \qquad \forall\, i \in \{1..n\} \cdot \Gamma \vdash v_i : u_i \\ u \preccurlyeq u' \qquad \forall\, i \in \{1..n\} \cdot u_i \preccurlyeq t_i \qquad \tau = u' \times t_1 \times \ldots \times t_n \to t_m\end{array}}{\Gamma \vdash (\ \mathsf{fun}\ m\langle \mathsf{this}, var_1, \ldots, var_n\rangle.e'' : \tau\ (\ loc, v_1, \ldots, v_n\ )\ ) : t_m}$$

Finally, Lemma 6 (Replacement with Subtyping) on page 11 gives $\Gamma \vdash e' : t$.

Case 3—EXEC. Here $e = \mathbb{E}[(\ \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e'' : \tau\ (\ v_0, \ldots, v_n\ )\ )]$ (where $\tau = t_0 \times \ldots \times t_n \to u$), $e' = \mathbb{E}[e''\{v_0/var_0, \ldots, v_n/var_n\}]$, and $S' = S$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$.

We will see that $\Gamma \vdash e' : t'$ for some $t' \preccurlyeq t$. $\Gamma \vdash e : t$ implies that $(\,\mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e'' : \tau\ (\,v_0, \ldots, v_n\,)\,)$ and all its subterms are well typed in Γ. Let $\Gamma \vdash (\,\mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e'' : \tau\ (\,v_0, \ldots, v_n\,)\,) : u$. This must be by T-EXEC:

$$\frac{\begin{array}{c}\Gamma, var_0 : t_0, \ldots, var_n : t_n \vdash e'' : u' \qquad u' \preccurlyeq u \\ \forall i \in \{0..n\} \cdot \Gamma \vdash v_i : t_i' \qquad \forall i \in \{0..n\} \cdot t_i' \preccurlyeq t_i \\ \tau = t_0 \times \ldots \times t_n \to u\end{array}}{\Gamma \vdash (\,\mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e'' : \tau\ (\,v_0, \ldots, v_n\,)\,) : u}$$

By Lemma 2 (Substitution) on page 39, $\Gamma \vdash e''\{v_0/\,var_0, \ldots, v_n/\,var_n\} : u''$ for some $u'' \preccurlyeq u' \preccurlyeq u$. Finally, by Lemma 6 (Replacement with Subtyping) on page 11 $\Gamma \vdash e' : t'$ for some $t' \preccurlyeq t$.

Case 4—GET. In this case $e = \mathbb{E}[loc.f]$, $e' = \mathbb{E}[v]$ (where $S(loc) = [u\,.\,F]$ and $F(f) = v$), and $S' = S$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$.

We will see that $\Gamma \vdash \mathbb{E}[v] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma \vdash loc.f : s$. The last step in this derivation must be T-GET. By the first hypothesis of T-GET, by T-LOC, and by $\Gamma \approx S$, we have $\Gamma(loc) = u$. By the second hypothesis of T-GET, $fieldsOf(u)(f) = s$. Also by $\Gamma \approx S$, $S(v) = [u'\,.\,F']$ where $u' \preccurlyeq s$ and $\Gamma(v) = u'$.

Thus, $\Gamma \vdash v : u'$ and, by Lemma 6 (Replacement with Subtyping) on page 11, $\Gamma \vdash \mathbb{E}[v] : t'$ where $t' \preccurlyeq t$.

Case 5—SET. In this case $e = \mathbb{E}[loc.f = v]$, $e' = \mathbb{E}[v]$, and $S' = S \oplus (loc \mapsto [u\,.\,F \oplus (f \mapsto v)])$, where $S(loc) = [u\,.\,F]$.

Let $\Gamma' = \Gamma$.

We will see that $\Gamma \approx S'$. S' only changes in its mapping for *loc*. To see that part 1 of the consistency definition holds, note that $S'(loc) = [u\,.\,F \oplus (f \mapsto v)]$. For part 1(a) $\Gamma(loc) = u$, since $S(loc) = [u\,.\,F]$ and $\Gamma \approx S$. For part 1(b) $dom(F \oplus (f \mapsto v)) = dom(fieldsOf(u))$, since $loc.f = v$ is well typed.

For part 1(c), $rng(F \oplus (f \mapsto v)) = rng(F) \cup \{v\}$. Now since $loc.f = v$ is well typed, we have $v \in dom(\Gamma)$ or $v = \mathsf{null}$. In the former case, by $\Gamma \approx S$, we have $v \in dom(S)$. $v \in dom(S)$ implies $v \in dom(S')$. So in either case $rng(F) \cup \{v\} \subseteq dom(S') \cup \{\mathsf{null}\}$.

Part 1(d) holds for all $f' \in dom(F)$, $f' \neq f$. Part 1(d) holds vacuously for f if $v = \mathsf{null}$. Otherwise, $(F \oplus (f \mapsto v))(f) = v$ and, by T-SET and T-LOC, $\Gamma(v) \preccurlyeq fieldsOf(u)(f)$.

Parts 2 and 3 hold since $dom(S') = dom(S)$.

To see that $\Gamma \vdash \mathbb{E}[v] : t$, let $\Gamma \vdash loc.f = v : s$. By T-SET, $\Gamma \vdash v : s$ and by Lemma 5 (Replacement) on page 11, $\Gamma \vdash \mathbb{E}[v] : t$.

Case 6—CAST. Here $e = \mathbb{E}[\mathsf{cast}\ t''\ loc]$, $e' = \mathbb{E}[loc]$, $S' = S$, $S(loc) = [u\,.\,F]$, and $u \preccurlyeq t''$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$.

To see that $\Gamma \vdash \mathbb{E}[loc] : t'$ for some $t' \preccurlyeq t$, note that $\Gamma(loc) = u$ by consistency of Γ with S. Thus $\Gamma \vdash loc : u$. By T-CAST, $\Gamma \vdash \mathsf{cast}\ t''\ loc : t''$. Since $u \preccurlyeq t''$, by Lemma 6 (Replacement with Subtyping) on page 11 we have $\Gamma \vdash \mathbb{E}[loc] : t'$ where $t' \preccurlyeq t$.

Case 7—NCAST. Here $e = \mathbb{E}[\mathsf{cast}\ t''\ \mathsf{null}]$, $e' = \mathbb{E}[\mathsf{null}]$, $S' = S$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$.

Now $\Gamma \vdash \mathsf{cast}\ t''\ \mathsf{null} : t''$. By T-NULL, $\Gamma \vdash \mathsf{null} : t''$. So by Lemma 5 (Replacement) on page 11, $\Gamma \vdash \mathbb{E}[\mathsf{null}] : t$.

Case 8—SKIP. Here $e = \mathbb{E}[v;\ e'']$, $e' = \mathbb{E}[e'']$, $S' = S$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$.

Since $\Gamma \vdash \mathbb{E}[v;\ e''] : t$, let $\Gamma \vdash v;\ e'' : t''$. This derivation must be by T-SEQ, the second hypothesis of which says $\Gamma \vdash e'' : t''$. By Lemma 5 (Replacement) on page 11, $\Gamma \vdash \mathbb{E}[e''] : t$.

The remaining evaluation rules reduce e to an error condition and are not applicable to the theorem. □

Theorem 8 (Progress)**.** *For an expression e, a stack J, a store S, and a type environment Γ consistent with S, if $\Gamma \vdash e : t$ then either:*

— *$e = loc$ and $loc \in dom(S)$,*

— *$e = null$, or*

— *one of the following hold:*

- $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$
- $\langle e, J, S\rangle \hookrightarrow \langle \mathit{NullPointerException}, J', S'\rangle$
- $\langle e, J, S\rangle \hookrightarrow \langle \mathit{ClassCastException}, J', S'\rangle$

Proof. If $e = loc$, then $\Gamma \vdash loc : t$ by T-LOC. This means that $loc \in dom(\Gamma)$ and, since $\Gamma \approx S$ we have $loc \in dom(S)$.

If $e = \mathsf{null}$, then the claim holds.

Finally, when e is not a value we consider cases based on the current redex of e. Cases where the redex matches NEW, EXEC, NCAST, SKIP, NCALL, NGET, and NSET are trivial. For the remaining cases we must show that the side conditions of the appropriate evaluation rules are satisfied.

Case 1—$e = \mathbb{E}[loc.m(\ v_1, \ldots, v_n\ )]$. Because e is well typed, $\Gamma \vdash loc : s$ for some type s. Thus, $loc \in dom(\Gamma)$, and part 2 of $\Gamma \approx S$ implies $loc \in dom(S)$. Let $S(loc) = [s' \,.\, F]$. Now $s' = s$ by part 1(a) of $\Gamma \approx S$.

Because $loc.m(\ v_1, \ldots, v_n\ )$ is well typed, we know by the hypotheses of T-CALL that $methodType(s, m)$ yields an n-arity method type. By the correspondence between the definitions of *methodType* and *methodBody*, it must be the case that $methodBody(s, m) = l$ for some fun term l. Thus $\langle e, J, S\rangle$ evolves by CALL.

Case 2—$e = \mathbb{E}[loc.f]$. As in the preceding case, e well typed implies $S(loc) = [s \,.\, F]$ where $\Gamma(loc) = s$. Now $loc.f$ well typed implies $f \in dom(fieldsOf(s))$ by the hypotheses of T-GET. Finally, part 1(b) of $\Gamma \approx S$ gives $f \in dom(F)$, so $\langle e, J, S\rangle$ evolves by GET.

Case 3—$e = \mathbb{E}[loc.f\ =\ v]$. Similar to the preceding case.

Case 4—$e = \mathbb{E}[cast\ t'\ loc]$. As in Case 1, e well typed implies $S(loc) = [s \,.\, F]$, where $\Gamma(loc) = s$. If $s \preccurlyeq t'$, then $\langle e, J, S\rangle \hookrightarrow \langle \mathbb{E}[loc], J, S\rangle$ by CAST; otherwise $\langle e, J, S\rangle \hookrightarrow \langle \mathrm{ClassCastException}, J, S\rangle$ by XCAST. □

7.2 MiniMAO$_1$

Lemma 2 (Substitution)**.** *If $\Gamma, var_1 : t_1, \ldots, var_n : t_n \vdash e : t$ and $\forall\, i \in \{1..n\} \cdot \Gamma \vdash e_i : s_i$ where $s_i \preccurlyeq t_i$ then $\Gamma \vdash e\{\!|e_1 / var_1, \ldots, e_n / var_n|\!\} : s$ for some $s \preccurlyeq t$.*

Proof. Let $\Gamma' = \Gamma, var_1 : t_1, \ldots, var_n : t_n$ and let $\{\!|\bar{e} / \overline{var}|\!\}$ represent $\{\!|e_1 / var_1, \ldots, e_n / var_n|\!\}$. The proof proceeds by structural induction on the derivation of $\Gamma \vdash e : t$ and by cases based on the last step in that derivation. The base cases are T-NEW, T-OBJ, T-NULL, T-LOC, and T-VAR. In the first four of these cases, e has no variables and $s = t$.

In the T-VAR base case, $e = var$, and there are two subcases. If $var \notin \{var_1, \ldots, var_n\}$ then $\Gamma'(var) = \Gamma(var) = t$ and the claim holds. Otherwise, without loss of generality, let $var = var_1$. Then $e\{\!|\bar{e} / \overline{var}|\!\} = e_1$, $\Gamma \vdash e\{\!|\bar{e} / \overline{var}|\!\} : s_1$, and $s_1 \preccurlyeq t_1 = t$.

The remaining cases cover the induction step. The induction hypothesis is that the claim of the lemma holds for all sub-derivations of the derivation being considered.

Case 1—T-CALL. Unchanged from original proof of Lemma 2 (Substitution) on page 39.

Case 2—T-EXEC. Unchanged from original proof.

Case 3—T-GET. This case is essentially unchanged from the original proof, except for some details regarding the extended *fieldsOf* auxiliary function. We restate the entire case for clarity.

In this case $e = e'.f$. The last step in the type derivation for e is

$$\frac{\Gamma' \vdash e' : u \qquad \mathit{fieldsOf}(u)(f) = t}{\Gamma' \vdash e'.f : t}$$

Now $e\{\bar{e}/\overline{var}\} = e'\{\bar{e}/\overline{var}\}.f$, and by the induction hypothesis $\Gamma \vdash e'\{\bar{e}/\overline{var}\} : u'$, where $u' \preccurlyeq u$. Consider subcases on whether u' is a class or an aspect. If $\mathit{isClass}(u')$, then by the definition of *fieldsOf* and by the first hypothesis of T-CLASS, $\mathit{fieldsOf}(u')(f) = \mathit{fieldsOf}(u)(f) = t$. On the other hand, if u' is an aspect, then $u' = u$ (since an aspect is only a subtype of itself and Object, and $u \neq$ Object because $\mathit{fieldsOf}(u) \neq \emptyset$). So again $\mathit{fieldsOf}(u')(f) = \mathit{fieldsOf}(u)(f) = t$. In either case, $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$ and the claim holds.

Case 4—T-SET. Like the previous case, this case is essentially unchanged from Lemma 2 (Substitution) on page 39, but with the same concession made for the subcases on *fieldsOf*.

Case 5—T-CAST. Unchanged from original proof.

Case 6—T-SEQ. Unchanged from original proof.

Case 7—T-PROC. Here $e = e'_0.\text{proceed}(\, e'_1, \ldots, e'_p\,)$ and the last derivation step is

$$\frac{\forall i \in \{0..p\} \cdot \Gamma' \vdash e'_i : u'_i \qquad \Gamma'(\text{proceed}) = u_0 \times \ldots \times u_p \to t \qquad \forall i \in \{0..p\} \cdot u'_i \preccurlyeq u_i}{\Gamma' \vdash e'_0.\text{proceed}(\, e'_1, \ldots, e'_p\,) : t}$$

Let $e''_i = e'_i\{\bar{e}/\overline{var}\}$ for all $i \in \{0..p\}$. Then $e\{\bar{e}/\overline{var}\} = e''_0.\text{proceed}(\, e''_1, \ldots, e''_p\,)$. Now $\Gamma(\text{proceed}) = \Gamma'(\text{proceed}) = u_0 \times \ldots \times u_p \to t$ and by the induction hypothesis $\forall i \in \{0..p\} \cdot (\Gamma \vdash e''_i : u''_i$, where $u''_i \preccurlyeq u'_i \preccurlyeq u_i)$. Thus, by T-PROC, $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$ and the claim holds.

Case 8—T-UNDER. Here $e = \text{under } e'$ and the last derivation step is

$$\frac{\Gamma' \vdash e' : t}{\Gamma' \vdash \text{under } e' : t}$$

The claim is immediate by the induction hypothesis.

Case 9—T-CHAIN. Here $e = \text{chain } \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (u_0 \times \ldots \times u_p \to t)|\!)(\, e'_0, \ldots, e'_p\,)$. The last derivation step for the judgment $\Gamma' \vdash e : t$ is by T-CHAIN, with the first two hypotheses being:

$$\forall i \in \{0..p\} \cdot \Gamma' \vdash e'_i : u'_i \qquad\qquad \forall i \in \{0..p\} \cdot u'_i \preccurlyeq u_i$$

Let $e''_i = e'_i\{\bar{e}/\overline{var}\}$ for all $i \in \{0..p\}$. Then $e\{\bar{e}/\overline{var}\} = \text{chain } \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (u_0 \times \ldots \times u_p \to t)|\!)(\, e''_0, \ldots, e''_p\,)$. Substitution does not recurse into the advice list, $\bar{B}$, or the join point abstraction.

As in the T-PROC case, the induction hypothesis gives $\forall i \in \{0..p\} \cdot (\Gamma \vdash e''_i : u''_i$, where $u''_i \preccurlyeq u'_i \preccurlyeq u_i)$. Because substitution does not replace variables within $\bar{B}$, the remaining hypothesis of T-CHAIN are unchanged in the type derivation of $e\{\bar{e}/\overline{var}\}$, except for using Γ instead of Γ'. This fact does not change the judgments. Thus, $\Gamma \vdash e\{\bar{e}/\overline{var}\} : t$.

Case 10—T-JOIN. Here $e = \text{joinpt } (\!|k, v_{opt}, m_{opt}, l_{opt}, (u_0 \times \ldots \times u_p \to t)|\!)(\, e'_0, \ldots, e'_p\,)$. The proof is like that for Case 9. □

Lemma 12 (Binding Soundness)**.** *Let S be a valid store and $J = (\!|\ldots, t_0 \times \ldots \times t_n \to t|\!) + J'$ be a stack consistent with S. If $\bar{B} = \mathit{adviceBind}(J, S)$, then $\forall [\![b, loc, e, \tau, \tau']\!] \in \bar{B}$ the following conditions hold:*

1. $\tau' = t_0 \times \ldots \times t_n \to t$,
2. $\emptyset \vdash b$ *OK, and*

Advice declaration: s around(s_1 $var_1, \ldots, s_p$ var_p): pcd { e)

$$
\begin{aligned}
&[\![b, loc, e, \tau, \tau']\!] \in \bar{B} \\
&\qquad \tau = s_1 \times \ldots \times s_p \to s \\
&\qquad \tau' = u_0 \times \ldots \times u_q \to u \\
&\qquad \Gamma' = var_1 : s_1, \ldots, var_p : s_p \\
&\qquad \Gamma' \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, u \,.\, V \,.\, V
\end{aligned}
$$

Figure 22: Meta-variables Used in the Proof of the Binding Soundness Lemma

3. *for* $\Gamma \approx S$ *the judgment* $\Gamma, this : \Gamma(loc), proceed : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e : t'$ *holds for some* $t' \preccurlyeq t$.

Proof. We will use some common meta-variables throughout the proof. Pick an arbitrary element of $\bar{B}$, $[\![b, loc, e, \tau, \tau']\!]$, and let $\tau = s_1 \times \ldots \times s_p \to s$. Let the advice corresponding to $[\![b, loc, e, \tau, \tau']\!]$ be

$$s \text{ around}(\ s_1\ var_1, \ldots, s_p\ var_p\ ):\ pcd\ \{\ e\ \}$$

with advice table entry $\langle loc, pcd, e, \tau, \tau' \rangle$. Let this advice be declared in an aspect a. T-ADV gives

$$\frac{V = \{var_1, \ldots, var_p\} \qquad \begin{array}{c} var_1 : s_1, \ldots, var_p : s_p \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, u \,.\, V \,.\, V \\ var_1 : s_1, \ldots, var_p : s_p, \text{this} : a, \text{proceed} : (u_0 \times \ldots \times u_q \to u) \vdash e : s' \end{array} \qquad s' \preccurlyeq s \preccurlyeq u}{\vdash s \text{ around}(\ s_1\ var_1, \ldots, s_p\ var_p\ ) : pcd\ \{\ e\ \} \text{ OK in } a} \tag{1}$$

By the construction of AT, $\tau' = u_0 \times \ldots \times u_q \to u$. To simplify the notation, let $\Gamma' = var_1 : s_1, \ldots, var_p : s_p$. For convenience, Figure 22 summarizes the use of these meta-variables in the proof.

Because a well-typed pointcut descriptor in MiniMAO$_1$ must consist of multiple primitive pointcut descriptors, it is difficult to prove the consequents of the lemma using a single inductive argument. Instead, we propose and prove a series of simpler subclaims. Each subclaim is proven via a structural induction on the pointcut type derivation. A well-typed pointcut descriptor that matches J will satisfy the antecedents of all the subclaims, and the consequents of the subclaims will imply the consequents of the lemma.

Consequent 1 on the previous page relates the proceed type of the advice, τ', to the function type in the join point abstraction. The proceed type, $\tau' = u_0 \times \ldots \times u_q \to u$, is constructed from the pointcut typing for the advice, $pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, V \,.\, V$. To satisfy the consequent we must show that $\tau' = t_0 \times \ldots \times t_n \to t$. We use three separate subclaims, one for each pertinent position in the pointcut typing. The subclaims let us show:

— $u_0 = t_0$,

— $q = n$, $\forall i \in \{1..n\} \cdot u_i = t_i$, and

— $u = t$

Subclaim 1. Assume $\Gamma' \vdash pcd : \hat{u} \,.\, u_0 \,.\, U \,.\, \hat{u}' \,.\, V' \,.\, V''$ (i.e., the "target type" is not $\bot$). Then

$$matchPCD(J, pcd, S) \neq \bot \implies u_0 = t_0$$

Proof of subclaim.

— pcd = call(t'' $idPat(..)$). Subclaim assumption cannot hold.

— pcd = execution(t'' $idPat(..)$). Subclaim assumption cannot hold.

— pcd = this(…). Subclaim assumption cannot hold.

— $pcd = \mathsf{target}(\ t''\ var''\ )$. By T-TargPCD, $t'' = u_0$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies t_0 = t'' \\ &\implies u_0 = t_0. \end{aligned}$$

— $pcd = \mathsf{args}(\ \dots\ )$. Subclaim assumption cannot hold.

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UnionPCD, $\Gamma' \vdash pcd_1 : \hat{u}_1 \mathbin{.} u_0 \mathbin{.} U_1 \mathbin{.} \hat{u}'_1 \mathbin{.} V_1 \mathbin{.} V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \mathbin{.} u_0 \mathbin{.} U_2 \mathbin{.} \hat{u}'_2 \mathbin{.} V_2 \mathbin{.} V'_2$. By the induction hypothesis, $matchPCD(J, pcd_1, S) \neq \bot \implies u_0 = t_0$ and $matchPCD(J, pcd_2, S) \neq \bot \implies u_0 = t_0$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ or } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies u_0 = t_0 \end{aligned}$$

— $pcd = pcd_1\ \&\&\ pcd_2$. By T-IntPCD and the definition of $\sqcup$, one of the following hold:

- $\Gamma' \vdash pcd_1 : \hat{u}_1 \mathbin{.} u_0 \mathbin{.} U_1 \mathbin{.} \hat{u}'_1 \mathbin{.} V_1 \mathbin{.} V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \mathbin{.} \bot \mathbin{.} U_2 \mathbin{.} \hat{u}'_2 \mathbin{.} V_2 \mathbin{.} V'_2$
- $\Gamma' \vdash pcd_1 : \hat{u}_1 \mathbin{.} \bot \mathbin{.} U_1 \mathbin{.} \hat{u}'_1 \mathbin{.} V_1 \mathbin{.} V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \mathbin{.} u_0 \mathbin{.} U_2 \mathbin{.} \hat{u}'_2 \mathbin{.} V_2 \mathbin{.} V'_2$

So the induction hypothesis holds for the type derivation of at least one of pcd_1 and pcd_2. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ and } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies u_0 = t_0 \end{aligned}$$

— $pcd = !\ pcd_1$. Subclaim assumption cannot hold.

Subclaim-□

Subclaim 2. Assume $\Gamma' \vdash pcd : \hat{u} \mathbin{.} \hat{u}' \mathbin{.} \langle u_1, \dots, u_q \rangle \mathbin{.} \hat{u}'' \mathbin{.} V' \mathbin{.} V''$ (i.e., the argument type sequence is not $\bot$). Then

$$matchPCD(J, pcd, S) \neq \bot \implies (q = n \text{ and } \forall i \in \{1..n\} \cdot u_i = t_i)$$

Proof of subclaim.

— $pcd = \mathsf{call}(\ \dots\ )$. Subclaim assumption cannot hold.

— $pcd = \mathsf{execution}(\ \dots\ )$. Subclaim assumption cannot hold.

— $pcd = \mathsf{this}(\ \dots\ )$. Subclaim assumption cannot hold.

— $pcd = \mathsf{target}(\ \dots\ )$. Subclaim assumption cannot hold.

— $pcd = \mathsf{args}(\ t''_1\ var''_1, \dots, t''_w\ var''_w\ )$. By T-ArgsPCD, $w = q$ and $\forall i \in \{1..q\} \cdot u_i = t''_i$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies w = n \text{ and } \forall i \in \{1..n\} \cdot t_i = t''_i \\ &\implies q = n \text{ and } \forall i \in \{1..n\} \cdot u_i = t_i \end{aligned}$$

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UnionPCD, $\Gamma' \vdash pcd_1 : \hat{u}_1 \mathbin{.} \hat{u}'_1 \mathbin{.} \langle u_1, \dots, u_q \rangle \mathbin{.} \hat{u}''_1 \mathbin{.} V_1 \mathbin{.} V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \mathbin{.} \hat{u}'_2 \mathbin{.} \langle u_1, \dots, u_q \rangle \mathbin{.} \hat{u}''_2 \mathbin{.} V_2 \mathbin{.} V'_2$. By the induction hypothesis, $matchPCD(J, pcd_1, S) \neq \bot \implies q = n$ and $\forall i \in \{1..n\} \cdot u_i = t_i$ and similarly for $matchPCD(J, pcd_2, S)$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ or } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies q = n \text{ and } \forall i \in \{1..n\} \cdot u_i = t_i \end{aligned}$$

— $pcd = pcd_1$ && pcd_2. By T-INTPCD and the definition of $\sqcup$, one of the following hold:

- $\Gamma' \vdash pcd_1 : \hat{u}_1 . \hat{u}'_1 . \langle u_1, \ldots, u_q \rangle . \hat{u}''_1 . V_1 . V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 . \hat{u}'_2 . \bot . \hat{u}''_2 . V_2 . V'_2$
- $\Gamma' \vdash pcd_1 : \hat{u}_1 . \hat{u}'_1 . \bot . \hat{u}''_1 . V_1 . V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 . \hat{u}'_2 . \langle u_1, \ldots, u_q \rangle . \hat{u}''_2 . V_2 . V'_2$

So the induction hypothesis holds for the type derivation of at least one of pcd_1 and pcd_2. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ and } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies q = n \text{ and } \forall i \in \{1..n\} \cdot u_i = t_i \end{aligned}$$

— $pcd = !\ pcd_1$. Subclaim assumption cannot hold.

Subclaim-□

Subclaim 3. Assume $\Gamma' \vdash pcd : \hat{u} . \hat{u}' . U . u . V' . V''$ (i.e., the "return type" is not $\bot$). Then

$$matchPCD(J, pcd, S) \neq \bot \implies u = t$$

Proof of subclaim.

— $pcd = \mathsf{call}(\ t''\ idPat(..)\ )$. By T-CALLPCD, $t'' = u$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies t = t'' \\ &\implies u = t. \end{aligned}$$

— $pcd = \mathsf{execution}(\ t''\ idPat(..)\ )$. Similar to previous case, but by T-EXECPCD.

— $pcd = \mathsf{this}(\ \ldots\ )$. Subclaim assumption cannot hold.

— $pcd = \mathsf{target}(\ \ldots\ )$. Subclaim assumption cannot hold.

— $pcd = \mathsf{args}(\ \ldots\ )$. Subclaim assumption cannot hold.

— $pcd = pcd_1\ ||\ pcd_2$. By T-UNIONPCD, $\Gamma' \vdash pcd_1 : \hat{u}_1 . \hat{u}'_1 . U_1 . u . V_1 . V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 . \hat{u}'_2 . U_2 . u . V_2 . V'_2$. By the induction hypothesis, $matchPCD(J, pcd_1, S) \neq \bot \implies u = t$ and $matchPCD(J, pcd_2, S) \neq \bot \implies u = t$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ or } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies u = t \end{aligned}$$

— $pcd = pcd_1$ && pcd_2. By T-INTPCD and the definition of $\sqcup$, one of the following hold:

- $\Gamma' \vdash pcd_1 : \hat{u}_1 . \hat{u}'_1 . U_1 . u . V_1 . V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 . \hat{u}'_2 . U_2 . \bot . V_2 . V'_2$
- $\Gamma' \vdash pcd_1 : \hat{u}_1 . \hat{u}'_1 . U_1 . \bot . V_1 . V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 . \hat{u}'_2 . U_2 . u . V_2 . V'_2$

So the induction hypothesis holds for the type derivation of one of pcd_1 and pcd_2. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ and } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies u = t \end{aligned}$$

— $pcd = !\ pcd_1$. Subclaim assumption cannot hold.

Subclaim-□

With these three subclaims we can now prove consequent 1 on page 44. The first hypothesis of T-ADV (see (1) on page 45) is:

$$\Gamma' \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, u \,.\, V \,.\, V$$

Thus, the target type is not $\bot$, nor is the argument type sequence, nor the return type. So the assumptions of the first three subclaims all hold. Furthermore, by the definition of *adviceBind*, $[\![b, loc, e, \tau, \tau']\!] \in \bar{B}$ implies $matchPCD(J, pcd, S) \neq \bot$. Thus:

$$\begin{array}{ll}
\tau' = u_0 \times \ldots \times u_q \to u & \text{by construction of } AT \\
\quad = t_0 \times u_1 \times \ldots \times u_q \to u & \text{by Subclaim 1} \\
\quad = t_0 \times t_1 \times \ldots \times t_n \to u & \text{by Subclaim 2} \\
\quad = t_0 \times \ldots \times t_n \to u & \\
\quad = t_0 \times \ldots \times t_n \to t & \text{by Subclaim 3}
\end{array}$$

We next turn to consequent 2 on page 44. We can this prove consequent with a single subclaim. We use a subclaim that is stronger than the consequent, partly so that the induction hypothesis is sufficiently powerful. The stronger subclaim will also be useful in proving consequent 3. In the subclaim, $var(b)$ means all variables appearing in b (as defined in Figure 19 on page 32).

Subclaim 4. Assume $\Gamma' \vdash pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, \hat{u}'' \,.\, V' \,.\, V''$. Then $matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle$ implies all of the following:

$$\emptyset \vdash b \text{ OK} \tag{2a}$$

$$V' \subseteq var(b) \subseteq V'' \tag{2b}$$

$$\hat{u} = \bot \iff \alpha = - \tag{2c}$$

$$\hat{u}' = \bot \iff \beta_0 = - \tag{2d}$$

$$U = \bot \implies x = 0 \tag{2e}$$

$$U \neq \bot \implies x = n \tag{2f}$$

$$U = \bot \iff \forall i \in \{1..x\} \cdot \beta_i = - \tag{2g}$$

Proof of subclaim.

— $pcd = \text{call}(\ t''\ idPat(..)\ )$. By T-CALLPCD, $\Gamma' \vdash pcd : \bot \,.\, \bot \,.\, \bot \,.\, t'' \,.\, \emptyset \,.\, \emptyset$. By the definition of *matchPCD*,

$$\begin{array}{rl}
matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies & b = \langle -, - \rangle \\
\implies & \emptyset \vdash b \text{ OK} \\
& V' = \emptyset \subseteq var(b) \subseteq \emptyset = V'' \\
& \hat{u} = \bot \text{ and } \alpha = - \text{ so (2c) holds} \\
& \hat{u}' = \bot \text{ and } \beta_0 = - \text{so (2d) holds} \\
& U = \bot \text{ and } x = 0 \text{ so (2e) holds} \\
& U = \bot \text{ so (2f) holds} \\
& U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously true, so (2g) holds}
\end{array}$$

— $pcd = \text{execution}(\ t''\ idPat(..)\ )$. Similar to previous case, but by T-EXECPCD.

— $pcd = \text{this}(\ t''\ var''\ )$. By T-THISPCD, $\Gamma' \vdash pcd : t'' \,.\, \bot \,.\, \bot \,.\, \bot \,.\, \{var''\} \,.\, \{var''\}$. By the definition of *matchPCD*,

$$\begin{array}{rl}
matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies & b = \langle var'' \mapsto v, - \rangle \text{ for some } v \in \mathcal{V} \\
\implies & \emptyset \vdash b \text{ OK} \\
& V' = \{var''\} \subseteq var(b) \subseteq \{var''\} = V'' \\
& \hat{u} \neq \bot \text{ and } \alpha \neq - \text{ so (2c) holds} \\
& \hat{u}' = \bot \text{ and } \beta_0 = - \text{ so (2d) holds} \\
& U = \bot \text{ and } x = 0 \text{ so (2e) holds} \\
& U = \bot \text{ so (2f) holds} \\
& U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously true, so (2g) holds}
\end{array}$$

— $pcd = \text{target}(\ t''\ var''\ )$. By T-TargPCD, $\Gamma' \vdash pcd : \bot \,.\, t'' \,.\, \bot \,.\, \bot \,.\, \{var''\} \,.\, \{var''\}$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, var'' \rangle \\ &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' = \{var''\} \subseteq var(b) \subseteq \{var''\} = V'' \\ &\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (2c) holds} \\ &\quad \hat{u}' \neq \bot \text{ and } \beta_0 \neq - \text{ so (2d) holds} \\ &\quad U = \bot \text{ and } x = 0 \text{ so (2e) holds} \\ &\quad U = \bot \text{ so (2f) holds} \\ &\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously true, so (2g) holds} \end{aligned}$$

— $pcd = \text{args}(\ t''_1\ var''_1, \ldots, t''_w\ var''_w\ )$. By T-ArgsPCD, $\Gamma' \vdash pcd : \bot \,.\, \bot \,.\, \langle t''_1, \ldots, t''_w \rangle \,.\, \bot \,.\, V' \,.\, V''$ where $V' = V'' = \{var''_1, \ldots, var''_w\}$, and all var''_i are unique. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, -, var''_1, \ldots, var''_w \rangle \\ &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' \subseteq var(b) \subseteq V'' \\ &\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (2c) holds} \\ &\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{ so (2d) holds} \\ &\quad U \neq \bot \text{ so (2e) holds} \\ &\quad U \neq \bot \text{ and } x = w = n \text{ by Subclaim 2, so (2f) holds} \\ &\quad U \neq \bot \text{ and } \exists i \in \{1..0\} \cdot \beta_i \neq - \text{ so (2g) holds} \end{aligned}$$

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UnionPCD, let

$$\begin{aligned} \Gamma' \vdash pcd_1 &: \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, \hat{u}''_1 \,.\, V_1 \,.\, V'_1 \\ \Gamma' \vdash pcd_2 &: \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, \hat{u}''_2 \,.\, V_2 \,.\, V'_2 \end{aligned}$$

Also let $matchPCD(J, pcd_1, S) = r_1$ and $matchPCD(J, pcd_2, S) = r_2$.

By elementary set theory, $V' = V_1 \cap V_2 \implies V' \subseteq V_1$ and $V' \subseteq V_2$. Dually, $V'_1 \subseteq V''$ and $V'_2 \subseteq V''$. By the definition of *matchPCD*,

$$matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies b = r_1 \neq \bot \text{ or } b = r_2 \neq \bot$$

Without loss of generality, let $b = r_1$. Then the induction hypothesis gives:

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' \subseteq V_1 \subseteq var(b) \subseteq V'_1 \subseteq V'' \\ &\quad (\hat{u} = \bot \iff \alpha = -) \\ &\quad (\hat{u}' = \bot \iff \beta_0 = -) \\ &\quad (U = \bot \implies x = 0) \\ &\quad (U \neq \bot \implies x = n) \\ &\quad (U = \bot \iff \forall i \in \{1..x\} \cdot \beta_i = -) \end{aligned}$$

— $pcd = pcd_1\ \&\&\ pcd_2$. By T-IntPCD, let

$$\begin{aligned} \Gamma' \vdash pcd_1 &: \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, \hat{u}''_1 \,.\, V_1 \,.\, V'_1 \\ \Gamma' \vdash pcd_2 &: \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, \hat{u}''_2 \,.\, V_2 \,.\, V'_2 \end{aligned}$$

Also let $matchPCD(J, pcd_1, S) = r_1$ and $matchPCD(J, pcd_2, S) = r_2$. By the definition of *matchPCD*:

$$matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies r_1 \neq \bot,\ r_2 \neq \bot, \text{ and } b = r_1 \sqcup r_2$$

Thus, all the consequents of the subclaim hold for pcd_1 and pcd_2 Assume $matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle$, let

$$r_1 = \langle \alpha_1, \beta_{0,1}, \ldots, \beta_{x_1,1} \rangle$$
$$r_2 = \langle \alpha_2, \beta_{0,2}, \ldots, \beta_{x_2,2} \rangle$$

and consider each consequent of the subclaim.

– By T-INTPCD, $\hat{u} = \hat{u}_1 \sqcup \hat{u}_2$. By the definition of $\sqcup$,

$$\begin{aligned} \hat{u} = \bot &\implies \hat{u}_1 = \bot = \hat{u}_2 \\ &\implies \alpha_1 = -, \alpha_2 = - \text{ by induction hypothesis} \\ &\implies \alpha = - \sqcup - = - \text{ by definition of } \sqcup \end{aligned}$$

On the other hand,

$$\hat{u} \neq \bot \implies \hat{u}_1 \neq \bot \text{ or } \hat{u}_2 \neq \bot, \text{ but not both}$$

Without loss of generality, let $\hat{u}_2 = \bot$

$$\begin{aligned} \hat{u}_1 \neq \bot \text{ and } \hat{u}_2 = \bot &\implies \alpha_1 \neq -, \alpha_2 = - \text{ by induction hypothesis} \\ &\implies \alpha = \alpha_1 \neq - \text{ by definition of } \sqcup \end{aligned}$$

So $\hat{u} = - \iff \alpha = -$, and (2c) holds.

– Similarly, $\hat{u}' = - \iff \beta_0 = -$, and (2d) holds.

– By T-INTPCD, $U = U_1 \sqcup U_2$. By the definition of $\sqcup$,

$$\begin{aligned} U = \bot &\implies U_1 = \bot = U_2 \\ &\implies x_1 = 0 = x_2 \text{ by induction hypothesis} \\ &\implies x = 0 \text{ by definition of } \sqcup \\ &\implies \forall i \in \{1..x\} \cdot \beta_i = -, \text{ vacuously} \end{aligned}$$

On the other hand,

$$U \neq \bot \implies U_1 \neq \bot \text{ or } U_2 \neq \bot, \text{ but not both}$$

Without loss of generality, let $U_2 = \bot$

$$\begin{aligned} U_1 \neq \bot \text{ and } U_2 = \bot &\implies x_1 = n, x_2 = 0, \exists i \in \{1..n\} \cdot \beta_{i,1} \neq - \text{ by induction hypothesis} \\ &\implies x = n, \forall i \in \{1..x\} \cdot \beta_i = \beta_{i,1} \text{ by definition of } \sqcup \\ &\implies \exists i \in \{1..x\} \cdot \beta_i \neq - \end{aligned}$$

So $(U = - \implies x = 0)$, $(U \neq - \implies x = n)$, and $(U = - \iff \forall i \in \{1..x\} \cdot \beta_i = -)$. Thus, (2e), (2f), and (2g) all hold.

– The above arguments also demonstrate that $var(b) = var(r_1) \cup var(r_2)$, since at each position at most one of r_1 and r_2 is not "$-$". Thus, there are no collisions that could cause $\sqcup$ to drop a variable that appears in r_2. By the induction hypothesis, $V_1 \subseteq var(r_1) \subseteq V_1'$ and $V_2 \subseteq var(r_2) \subseteq V_2'$. By T-INTPCD,

$$\begin{aligned} V_1' \cap V_2' = \emptyset &\implies var(r_1) \cap var(r_2) = \emptyset \\ &\implies \emptyset \vdash b \text{ OK} \end{aligned}$$

Thus, (2a) holds.

– Finally, T-INTPCD, the induction hypothesis, and some set theory gives

$$V' = V_1 \cup V_2 \subseteq var(r_1) \cup var(r_2) = var(b).$$

and

$$var(b) = var(r_1) \cup var(r_2) \subseteq V_1' \cup V_2' = V''$$

Thus, $V' \subseteq var(b) \subseteq V''$ and (2b) holds.

— $pcd = !\ pcd_1$. By T-NEGPCD $\Gamma' \vdash pcd : \bot . \bot . \bot . \bot . \emptyset . \emptyset$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies & b = \langle -, - \rangle \\ \implies & \emptyset \vdash b \text{ OK} \\ & V' = \emptyset \subseteq var(b) \subseteq \emptyset = V'' \\ & \hat{u} = \bot \text{ and } \alpha = - \text{ so (2c) holds} \\ & \hat{u}' = \bot \text{ and } \beta_0 = - \text{so (2d) holds} \\ & U = \bot \text{ and } x = 0 \text{ so (2e) holds} \\ & U = \bot \text{ so (2f) holds} \\ & U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously true, so (2g) holds} \end{aligned}$$

Subclaim-□

By T-ADV, the assumption of the subclaim holds. Therefore, consequent 2 on page 44 holds by (2a).

Consequent 3 is more complex. To prove this consequent, it will suffice to show that

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s_1, \ldots, var_p : s_p \tag{3}$$

We will see that this juxtaposition of t_i in *typeBind* and s_i in the result is resolved by the pointcut descriptor typing rules and *matchPCD*, which will impose constraints on the types. We use a final subclaim.

Subclaim 5. Assume $\Gamma' \vdash pcd : \hat{u} . \hat{u}' . U . \hat{u}'' . V' . V''$, where $V'' \subseteq \{var_1, \ldots, var_p\}$. Then

$$\begin{aligned} & matchPCD(J, pcd, S) = b \neq \bot \\ & \qquad \implies \forall var \in var(b) \cdot (\exists i \in \{1..p\} \cdot (var = var_i \text{ and } typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_i) = s_i)) \end{aligned}$$

Proof of subclaim. The assumption of this subclaim implies the assumption for Subclaim 4 on page 48; we will make free use of the earlier result.

— $pcd = \mathsf{call}(\ \ldots\ )$. By T-CALLPCD, $V' = V'' = \emptyset$. By (2b) on page 48, $matchPCD(J, pcd, S) = b \neq \bot$ implies $var(b) = \emptyset$, satisfying the subclaim.

— $pcd = \mathsf{execution}(\ \ldots\ )$. Similar to previous case, but by T-EXECPCD.

— $pcd = \mathsf{this}(\ t''\ var''\ )$. By T-THISPCD, $V' = V'' = \{var''\}$. By the subclaim assumption, $var'' \in \{var_1, \ldots, var_p\}$. Without loss of generality, let $var'' = var_1$. By the hypothesis of T-THISPCD and the definition of Γ', $t'' = s_1$.

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = \langle var_1 \mapsto loc_1, - \rangle$$

for some loc_1 in J, where $loc_1 \in dom(S)$ by $J \approx S$, $S(loc_1) = [s_1 . F]$ by definition of *matchPCD*, and $\Gamma(loc_1) = s_1$ by $\Gamma \approx S$. Thus,

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s_1.$$

— pcd = target(t'' var''). By T-TargPCD, $V' = V'' = \{var''\}$. By the subclaim assumption, $var'' \in \{var_1, \ldots, var_p\}$. Without loss of generality, let $var'' = var_1$. By the hypothesis of T-TargPCD and the definition of Γ', $t'' = s_1$.

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = \langle -, var_1 \rangle$$

where $t_0 = t''$ by definition of *matchPCD*. So $t_0 = s_1$ and

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s_1.$$

— pcd = args(t_1'' var_1'', …, t_w'' var_w''). By T-ArgsPCD and the subclaim assumption, all var_i'' are unique and $V' = V'' = \{var_1'', \ldots, var_w''\} \subseteq \{var_1, \ldots, var_p\}$. Thus,

$$\forall i \in \{1..w\} \cdot (\exists! j \in \{1..p\} \cdot (t_i'' = s_j \text{ and } var_i'' = var_j)) \tag{4}$$

The definition of *matchPCD* gives

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = \langle -, -, var_1'', \ldots, var_w'' \rangle$$

where $n = w$ and $\forall i \in \{1..w\} \cdot (t_i'' = t_i)$. So

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1'' : t_1'', \ldots, var_w'' : t_w''$$

Let $var \in var(b)$. Without loss of generality, let $var = var_1''$. Now

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_1'') = t_1''.$$

By (4), there exists j such that $var_1'' = var_j$ and $t_1'' = s_j$, thus the subclaim holds.

— $pcd = pcd_1 \;||\; pcd_2$. By T-UnionPCD and the subclaim assumption, let

$$\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}_1' \,.\, U_1 \,.\, \hat{u}_1'' \,.\, V_1 \,.\, V_1' \qquad matchPCD(J, pcd_1, S) = r_1$$
$$\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}_2' \,.\, U_2 \,.\, \hat{u}_2'' \,.\, V_2 \,.\, V_2' \qquad matchPCD(J, pcd_2, S) = r_2$$

By the definition of *matchPCD*,

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = r_1 \neq \bot \text{ or } b = r_2 \neq \bot$$

So either

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = typeBind(\Gamma, r_1, \langle t_0, \ldots, t_n \rangle)$$

or

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = typeBind(\Gamma, r_2, \langle t_0, \ldots, t_n \rangle).$$

As noted in the corresponding case of the proof of Subclaim 4, $V_1' \subseteq V''$ and $V_2' \subseteq V''$. Thus, we can apply the induction hypothesis to the type derivations for pcd_1 and pcd_2, and the subclaim holds.

— $pcd = pcd_1 \;\&\&\; pcd_2$. By T-IntPCD and the subclaim assumption, let

$$\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}_1' \,.\, U_1 \,.\, \hat{u}_1'' \,.\, V_1 \,.\, V_1' \qquad matchPCD(J, pcd_1, S) = r_1$$
$$\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}_2' \,.\, U_2 \,.\, \hat{u}_2'' \,.\, V_2 \,.\, V_2' \qquad matchPCD(J, pcd_2, S) = r_2$$

By the definition of *matchPCD*,

$$matchPCD(J, pcd, S) = b \neq \bot \implies r_1 \neq \bot \text{ and } r_2 \neq \bot$$

As argued in the corresponding case of Subclaim 4, $var(r_1)$ and $var(r_2)$ are disjoint. Also, since $V'' = V_1' \cup V_2'$, we have $V_1' \subseteq V''$ and similarly for V_2. Thus, the induction hypothesis is applicable to the type derivations for pcd_1 and pcd_2. Let $var \in var(b)$. By definition of the union of bindings, var is in exactly one of $var(r_1)$ and $var(r_2)$. In either case, the claim holds by the induction hypothesis.

— $pcd = !\ pcd_1$. By T-NEGPCD and subclaim assumption, $V' = V'' = \emptyset$.

$$\begin{aligned} matchPCD(J, pcd, S) = b \neq \bot &\implies b = \langle -, - \rangle \\ &\implies var(b) = \emptyset \end{aligned}$$

Subclaim-□

With this last subclaim in hand we can now prove the final consequent of the lemma. The first two hypotheses of T-ADV (see (1) on page 45) are:

$$\begin{gathered} \Gamma' \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, u \,.\, V \,.\, V \\ V = \{var_1, \ldots, var_p\} \end{gathered}$$

By definition of *adviceBind*, $[\![b, loc, e, \tau, \tau']\!] \in \bar{B}$ implies $matchPCD(J, pcd, S) \neq \bot$. We first use Subclaim 4 and Subclaim 5 to prove equation (3) from page 51.

$$\begin{aligned} V = \{var_1, \ldots, var_p\} & && \text{by T-ADV} \\ &\implies var(b) = \{var_1, \ldots, var_p\} && \text{by (2b)} \\ &\implies \forall i \in \{1..p\} \cdot && \\ &\qquad (typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle))(var_i) = s_i) && \text{by Subclaim 5} \end{aligned}$$

Thus, all $var \in V$ are bound appropriately. By examination of the definition of *typeBind*, we see that

$$dom(typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)) = var(b) = V.$$

Thus, no additional variables are bound and

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s_1, \ldots, var_p : s_p$$

The third hypothesis of T-ADV gives

$$\begin{aligned} & var_1 : s_1, \ldots, var_p : s_p, \mathsf{this} : a, \mathsf{proceed} : \tau' \vdash e : s' \\ & \quad \implies \mathsf{this} : a, \mathsf{proceed} : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e : s' && \text{by (3)} \\ & \quad \implies \Gamma, \mathsf{this} : a, \mathsf{proceed} : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e : s' \end{aligned}$$

where the last implication is by Lemma 3 (Environment Extension), with appropriate α-conversion of b and e. Finally, the last hypothesis of T-ADV gives $s' \preccurlyeq s \preccurlyeq u = t$. Thus the final consequent holds. □

Lemma 13 (Advice Chaining)**.** *Let* $\Gamma, proceed : \tau \vdash e : t$, $j = (\!|\_, \_, \_, \_, \tau|\!)$, $\tau = t_0 \times \ldots \times t_n \to t$, *and for all* $B = [\![b, loc, e', \tau', \tau]\!] \in \bar{B}$ *let*

— $\Gamma, this : \Gamma(loc), proceed : \tau, typeBind(\Gamma, b, (t_0, \ldots, t_n)) \vdash e' : s'$,

— $\Gamma \vdash b$ *OK, and*

— $s' \preccurlyeq t$.

Then $\Gamma \vdash \langle\!\langle e \rangle\!\rangle_{\bar{B}, j} : t$.

Proof. The proof is by structural induction on the type derivation for e. In the base case, the type derivation for e is by one of T-NEW, T-OBJ, T-VAR, T-LOC, or T-NULL. For all of these rules e does not contain a proceed expression. Therefore, $\langle\!\langle e \rangle\!\rangle_{\bar{B}, j} = e$ and the claim holds by Lemma 4 (Environment Contraction) on page 11.

The induction hypothesis is that the claim holds for all type derivations smaller than the one for e. For all the remaining expression typing rules but T-PROC, the claim follows immediately from the induction hypothesis. So the only interesting case is for

$$\begin{gathered} e = e_0.\mathsf{proceed}(\, e_1, \ldots, e_n \,) \text{ and} \\ \langle\!\langle e \rangle\!\rangle_{\bar{B}, j} = \mathsf{chain}\ \bar{B}, j(\, \langle\!\langle e_0 \rangle\!\rangle_{\bar{B}, j}, \ldots, \langle\!\langle e_n \rangle\!\rangle_{\bar{B}, j} \,) \end{gathered}$$

Assuming that $\Gamma, \text{proceed}:\tau \vdash e:t$, we need to show that $\Gamma \vdash \langle\!\langle e \rangle\!\rangle_{\bar{B},j} : t$. The later must be by T-CHAIN, so we must establish the hypotheses for that rule. Now the last step in the type derivation for e must be T-PROC:

$$\frac{\forall i \in \{0..n\} \cdot \Gamma, \text{proceed}:\tau \vdash e_i : u_i \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i}{\Gamma, \text{proceed}:\tau \vdash e_0.\text{proceed}(\, e_1, \ldots, e_n\,) : t}$$

By the hypotheses of this judgment, the induction hypothesis, and transitivity of subtyping we have:

$$\forall i \in \{0..n\} \cdot \Gamma \vdash \langle\!\langle e_i \rangle\!\rangle_{\bar{B},j} : u_i' \text{ where } u_i' \preccurlyeq u_i \preccurlyeq t_i$$

The remaining hypotheses of T-CHAIN hold by the assumptions of the lemma regarding $\bar{B}$ and j, thus $\Gamma \vdash \langle\!\langle e \rangle\!\rangle_{\bar{B},j} : t$. □

Theorem 15 (Subject Reduction)**.** *Given a well typed MiniMAO$_1$ program, for an expression e, a valid store S, a stack J consistent with S, and a type environment Γ consistent with S, if $\Gamma \vdash e:t$ and $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$, then $J' \approx S'$, S' is valid, and there exist Γ' and t' such that $\Gamma' \approx S'$, $\Gamma' \vdash e' : t'$, and $t' \preccurlyeq t$.*

Proof. The proof is by cases on the evaluation rule applied. We note that the evaluation rules obey a monotonicity property with regard to the store: none of evaluation rules remove a location from the domain of S, nor do they change the type of the object in any store location. Because none of the evaluation rules inherited from MiniMAO$_0$ modify the stack, $J' \approx S'$ for the proof cases corresponding to those rules. Also by the monotonicity property, S valid implies that part 1 of Definition 11 (Store Validity) on page 35 holds for S'. Based on the reduction step we can construct a Γ' consistent with S' that witnesses to the validity of S' and satisfies the claim. The cases for NEW, GET, SET, CAST, NCAST, and SKIP are unchanged from the original proof of Theorem 7 (Subject Reduction) on page 41.

Case 1—CALL$_A$. Here $e = \mathbb{E}[loc.m(\, v_1, \ldots, v_n\,)]$, $e' = \mathbb{E}[\text{joinpt } (\!|\text{call}, -, m, -, (s_0 \times \ldots \times s_n \to s)|\!)(\, loc, v_1, \ldots, v_n\,)]$ (where $S(loc) = [u\,.\,F]$, $methodType(s_0, m) = s_1 \times \ldots \times s_n \to s$, and $origType(u, m) = s_0$), $J' = J$, and $S' = S$.

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$ and $J' \approx S'$.

We will see that $\Gamma \vdash e' : t$. The judgment $\Gamma \vdash e : t$ implies that $loc.m(\, v_1, \ldots, v_n\,)$ and all its subterms are well typed in Γ. Let $\Gamma \vdash v_i : t_i$ for all $i \in \{1..n\}$. By part 1(a) of $\Gamma \approx S$, $\Gamma \vdash loc : u$. The type judgment for $loc.m(\, v_1, \ldots, v_n\,)$ must be by T-CALL with $\forall i \in \{1..n\} \cdot t_i \preccurlyeq s_i$ and $\Gamma \vdash loc.m(\, v_1, \ldots, v_n\,) : s$. By the definition of *origType*, $u \preccurlyeq s_0$. T-JOIN gives:[5]

$$\frac{\Gamma \vdash loc : u \qquad \forall i \in \{1..n\} \cdot \Gamma \vdash v_i : t_i \qquad u \preccurlyeq s_0 \qquad \forall i \in \{1..n\} \cdot t_i \preccurlyeq s_i}{\Gamma \vdash \text{joinpt } (\!|\text{call}, -, m, -, (s_0 \times \ldots \times s_n \to s)|\!)(\, loc, v_1, \ldots, v_n\,) : s}$$

Therefore, Lemma 5 (Replacement) on page 11 gives $\Gamma \vdash e' : t$.

Case 2—CALL$_B$. Here $e = \mathbb{E}[\text{chain } \bullet, (\!|\text{call}, -, m, -, \tau|\!)(\, loc, v_1, \ldots, v_n\,)]$, $e' = \mathbb{E}[(\, l\, (\, loc, v_1, \ldots, v_n\,)\,)]$ (where $S(loc) = [t_0\,.\,F]$ and $methodBody(t_0, m) = l$), $J' = J$, and $S' = S$.

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$ and $J' \approx S'$.

We will see that $\Gamma \vdash e' : t$. Let $e_{\text{left}} = \text{chain } \bullet, (\!|\text{call}, -, m, -, \tau|\!)(\, loc, v_1, \ldots, v_n\,)$. The judgment $\Gamma \vdash e : t$ implies that e_{left} and all its subterms are well typed. Let $\Gamma \vdash v_i : t_i$ for all $i \in \{1..n\}$ and let $\Gamma \vdash e_{\text{left}} : s$. By part 1(a) of $\Gamma \approx S$, $\Gamma \vdash loc : t_0$. The type judgment for e_{left} must be by T-CHAIN with τ of arity $n+1$ and return type s. Let $\tau = s_0 \times \ldots \times s_n \to s$. Then T-CHAIN gives $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$.

By Lemma 14 (Join Point Abstractions) on page 36, it must be the case that $methodType(s_0, m) = s_1 \times \ldots \times s_n \to s$. By the correspondence between the definitions of *methodType* and *methodBody*, and by T-CLASS, T-MET, and *override*, it must be the case that

$$l = methodBody(t_0, m) = \text{fun } m\langle \text{this}, var_1, \ldots, var_n\rangle.e'' : (u \times s_1 \times \ldots \times s_n \to s)$$

where $t_0 \preccurlyeq u$ and $\Gamma, \text{this}:u, var_1 : s_1, \ldots, var_n : s_n \vdash e'' : s'$ for some $s' \preccurlyeq s$.

[5] We omit the v_{opt} hypothesis because "–" is not a location.

Thus, T-EXEC gives

$$\frac{\Gamma, \mathsf{this}: u, var_1: s_1, \ldots, var_n: s_n \vdash e'': s' \qquad s' \preccurlyeq s \qquad \Gamma \vdash loc: t_0 \qquad \forall i \in \{1..n\} \cdot \Gamma \vdash v_i : t_i \qquad t_0 \preccurlyeq u \qquad \forall i \in \{1..n\} \cdot t_i \preccurlyeq s_i}{\Gamma \vdash (\,\mathsf{fun}\ m\langle \mathsf{this}, var_1, \ldots, var_n\rangle.e'' : (u \times s_1 \times \ldots \times s_n \to s)\ (\ loc, v_1, \ldots, v_n\ )) : s}$$

and Lemma 5 (Replacement) on page 11 gives $\Gamma \vdash e' : t$.

Case 3—EXEC$_A$. Here $e = \mathbb{E}[(\ l\ (\ v_0, \ldots, v_n\ )\ )]$ (where $l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e'' : (s_0 \times \ldots \times s_n \to s)$), $e' = \mathbb{E}[\mathsf{joinpt}\ (\!|\mathsf{exec}, v_0, m, l, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )]$, $J' = J$, and $S' = S$.

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$ and $J' \approx S'$.

We will see that $\Gamma \vdash e' : t$. The judgment $\Gamma \vdash e : t$ implies that $(\ l\ (\ v_0, \ldots, v_n\ )\ )$ and all its subterms are well typed. Let $\Gamma \vdash v_i : t_i$ for all $i \in \{0..n\}$. The type derivation of $(\ l\ (\ v_0, \ldots, v_n\ )\ )$ must be by T-EXEC with $\Gamma \vdash (\ l\ (\ v_0, \ldots, v_n\ )\ ) : s$ and $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$. If v_0 is a location, then $\Gamma \vdash v_0 : t_0$ must be by T-LOC, so $v_0 \in dom(\Gamma)$. Thus, $\Gamma \vdash \mathsf{joinpt}\ (\!|\mathsf{exec}, v_0, m, l, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ ) : s$ by T-JOIN. Lemma 5 (Replacement) on page 11 gives $\Gamma \vdash e' : t$.

Case 4—EXEC$_B$. Here $e = \mathbb{E}[\mathsf{chain}\ \bullet, (\!|\mathsf{exec}, v, m, l, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )]$ (where $l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e''$: $(s_0 \times \ldots \times s_n \to s)$), $e' = \mathbb{E}[\mathsf{under}\ e''\{\!|v_0/\ var_0, \ldots, v_n/\ var_n|\!\}]$, $J' = (\!|\mathsf{this}, v_0, -, -, -|\!) + J$, and $S' = S$.

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$.

We will see that $J' \approx S' = S$. Let $e_{\mathsf{left}} = \mathsf{chain}\ \bullet, (\!|\mathsf{exec}, v, m, l, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )$. Because e is well typed, it must be the case that e_{left} and all its subterms are well typed. Let $\Gamma \vdash v_i : t_i$ for all $i \in \{0..n\}$. If $v_0 = \mathsf{null}$, then $J' \approx S$ because J' has no new location. On the other hand, if v_0 is a location, then then judgment $\Gamma \vdash v_0 : t_0$ must be by T-LOC with $v_0 \in dom(\Gamma)$. By $\Gamma \approx S$, we have $v_0 \in dom(S)$. Because $J \approx S$ and v_0 is the only potentially new location in J', we have that $J' \approx S$.

We will also see that $\Gamma \vdash e' : t'$ for some $t' \preccurlyeq t$ by appealing to the Substitution Lemma. Rule T-CHAIN must be the last step in the type derivation for e_{left} with $\Gamma \vdash e_{\mathsf{left}} : s$. The second hypothesis of T-CHAIN says that $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$.

It remains to be seen that $\Gamma, var_0 : s_0, \ldots, var_n : s_n \vdash e'' : u$ for some $u \preccurlyeq s$. No fun terms may appear in user programs; they can only be introduced by the evaluation rules. By examination of the evaluation rules, we see that the only rule that introduces a new fun term is CALL$_B$. The term it introduces is provided by the *methodBody* auxiliary function. By the definition of *methodBody* and by T-MET it must be the case that $var_0 : s_0, \ldots, var_n : s_n \vdash e'' : u$ for some $u \preccurlyeq s$. By α-conversion and Lemma 3 (Environment Extension) on page 11 we have $\Gamma, var_0 : s_0, \ldots, var_n : s_n \vdash e'' : u$. Thus, by Lemma 2 (Substitution) on page 43, $\Gamma \vdash e''\{\!|v_0/\ var_0, \ldots, v_n/\ var_n|\!\} : u'$ where $u' \preccurlyeq u \preccurlyeq s$. So Lemma 6 (Replacement with Subtyping) on page 11 gives $\Gamma \vdash e' : t'$ for some $t' \preccurlyeq t$.

Case 5—BIND. Here:

$$\begin{aligned}
e &= \mathbb{E}[\mathsf{joinpt}\ (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )] \\
e' &= \mathbb{E}[\mathsf{under}\ \mathsf{chain}\ \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )] \\
\bar{B} &= adviceBind((\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!) + J, S) \\
J' &= (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!) + J \\
S' &= S
\end{aligned}$$

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$.

We will see that $J' \approx S'$. Let $e_{\mathsf{left}} = \mathsf{joinpt}\ (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )$. Because e is well typed, it must be the case the e_{left} and all its subterms are well typed. The typing derivation for e_{left} must be by T-JOIN. Thus, if v_{opt} is a location it must be in $dom(\Gamma)$ and so $J' \approx S'$.

It remains to show that $\Gamma \vdash e' : t$. Let $e_{\mathsf{right}} = \mathsf{chain}\ \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )$. (By T-UNDER, e_{right} has the same type as under e_{right}, so we can focus on the smaller expression.) The typing judgment for e_{right} must be by T-CHAIN. So we next show that all the hypotheses of T-CHAIN are satisfied by e_{right}.

By the well-typedness of e_{left} and its subterms, let $\Gamma \vdash v_i : t_i$ for all $i \in \{0..n\}$. By T-JOIN, we have $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$.

The remaining hypotheses of T-CHAIN are related to the elements of the advice list, $\bar{B}$. Let

$$B = [\![b, loc, e'', \tau, \tau']\!]$$

be an arbitrary element of $\bar{B}$. By the definition of *adviceBind*, it must be the case that there exists a piece of advice with aspect table entry $\langle loc, pcd, e'', \tau, \tau' \rangle$ such that $matchPCD(J', pcd, S) = b \neq \bot$. By Lemma 12 (Binding Soundness) on page 44 we have:

$$\tau' = s_0 \times \ldots \times s_n \to s$$
$$\emptyset \vdash b \text{ OK}$$
$$\Gamma, \text{this} : \Gamma(loc), \text{proceed} : \tau', typeBind(\Gamma, b, \langle s_0, \ldots, s_n \rangle) \vdash e'' : s' \text{ for some } s' \preccurlyeq s$$

By appropriate α-conversion of b and e'', we have $\Gamma \vdash b$ OK. The remaining hypotheses of T-CHAIN are satisfied directly by the results of the lemma. Thus, $\Gamma \vdash e_{\text{right}} : s$ and by T-UNDER and Lemma 5 (Replacement) on page 11, $\Gamma \vdash e' : t$.

Case 6—ADVISE. Here

$$\begin{aligned} e &= \mathbb{E}[\text{chain } [\![b, loc, e'', \tau', \tau'']\!] + \bar{B}, j(\, v_0, \ldots, v_n \,)] \\ e' &= \mathbb{E}[\text{under } \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} \{\!\{ loc/\text{this} \}\!\} \{\!\{ (v_0, \ldots, v_n)/b \}\!\}] \\ J' &= (\!|\text{this}, loc, -, -, -|\!) + J \\ S' &= S \end{aligned}$$

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$. Because $[\![-]\!]$ terms can only be added to a program by the auxiliary function *adviceBind* called by BIND, we know from the definition of *adviceBind* and the validity and monotonicity of S that $loc \in dom(S)$. By $\Gamma \approx S$, we know $loc \in dom(\Gamma)$. Thus, $J' \approx S'$.

It remains to be shown that $\Gamma \vdash e' : t'$ for some $t' \preccurlyeq t$. Let

$$\begin{aligned} e_{\text{left}} &= \text{chain } [\![b, loc, e'', \tau, \tau']\!] + \bar{B}, j(\, v_0, \ldots, v_n \,) \text{ and} \\ e_{\text{right}} &= \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} \{\!\{ loc/\text{this} \}\!\} \{\!\{ (v_0, \ldots, v_n)/b \}\!\}. \end{aligned}$$

Because e is well typed, we know that e_{left} and all its subterms are also well typed. The type derivation for e_{left} must be by T-CHAIN. Let the last element of j be $t_0 \times \ldots \times t_n \to t_c$. Then by T-CHAIN the proceed type $\tau' = t_0 \times \ldots \times t_n \to t_c$. From the hypotheses of T-CHAIN, we have

$$\Gamma, \text{this} : \Gamma(loc), \text{proceed} : (t_0 \times \ldots \times t_n \to t_c), typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e'' : s$$

where $s \preccurlyeq t_c$. The constraints on $\bar{B}$ and j imposed by T-CHAIN satisfy the conditions of Lemma 13 (Advice Chaining) on page 53, so we have

$$\Gamma, \text{this} : \Gamma(loc), typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} : s \tag{5}$$

Next we will appeal to the Substitution Lemma. To do so, we will need to expand *typeBind* so that we can demonstrate that the conditions for the lemma hold. Let $b = \langle \alpha, \beta_0, \ldots, \beta_p \rangle$. Assume $\alpha = var' \mapsto loc'$ and $\beta_0 = var_0$.[6] Then (5) expands to

$$\Gamma, \text{this} : \Gamma(loc), var' : \Gamma(loc'), (var_i : t_i)_{i \in \{0..p\} \cdot \beta_i = var_i} \vdash \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} : s'.$$

and the binding substitution in e_{right} expands to give

$$\langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} \{\!\{ loc/\text{this}, loc'/var', (v_i/var_i)_{i \in \{0..p\} \cdot \beta_i = var_i} \}\!\}.$$

[6] The argument connecting *typeBind* to binding substitution is similar if α (resp β_0) is "$-$", but with typings and substitutions for var' (resp var_0) omitted.

Finally, by the hypotheses of T-CHAIN in the typing of e_{left} we have $\forall i \in \{0..n\} \cdot (\Gamma \vdash v_i : u'_i$ where $u'_i \preccurlyeq t_i)$. Thus, Lemma 2 (Substitution) on page 43 gives $\Gamma \vdash e_{\text{right}} : s'$ where $s' \preccurlyeq s \preccurlyeq t_{\text{C}}$. By T-UNDER and Lemma 6 (Replacement with Subtyping) on page 11, $\Gamma \vdash e' : t'$ for some $t' \preccurlyeq t$.

Case 7—UNDER. Here $e = \mathbb{E}[\mathsf{under}\ v]$, $e' = \mathbb{E}[v]$, $J = j + J'$ for some j, and $S' = S$.

Let $\Gamma' = \Gamma$.

Clearly $\Gamma' \approx S'$. Since the set of location is J' is a subset of those in J, $J' \approx S'$.

We will see that $\Gamma \vdash e' : t$. The judgment $\Gamma \vdash e : t$ implies that $\mathsf{under}\ v$ is well typed. Let $\Gamma \vdash \mathsf{under}\ v : t'$. This judgment must be by T-UNDER with the hypothesis $\Gamma \vdash v : t'$. So by Lemma 5 (Replacement) on page 11, we have $\Gamma \vdash e' : t$.

The remaining evaluation rules reduce e to an error condition and are not applicable to the theorem. □

Theorem 16 (Progress)**.** *For an expression e, a valid store S, a stack J consistent with S, and a type environment* Γ *consistent with S, if* $\Gamma \vdash e : t$ *then either:*

— *e = loc and loc* $\in$ *dom*(S)*,*

— *e = null, or*

— *one of the following hold:*

- $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$
- $\langle e, J, S\rangle \hookrightarrow \langle \mathit{NullPointerException}, J', S'\rangle$
- $\langle e, J, S\rangle \hookrightarrow \langle \mathit{ClassCastException}, J', S'\rangle$

Proof. If $e = loc$, then $\Gamma \vdash loc : t$ by T-LOC. This means that $loc \in dom(\Gamma)$ and, since $\Gamma \approx S$ we have $loc \in dom(S)$.

If $e = \mathsf{null}$, then the claim holds.

Finally, when e is not a value we consider cases based on the current redex of e. Cases where the redex matches NEW, NCAST, SKIP, NGET, NSET, EXEC_A, NCALL_A, and ADVISE are trivial. For the remaining cases we must show that the side conditions hold and the join point abstractions are of the correct form. The cases for redexes matched by GET, SET, and CAST are unchanged from the proof of Theorem 8 (Progress) on page 43.

Case 1—$e = \mathbb{E}[loc.m(\ v_1, \ldots, v_n\ )]$. Because e is well typed, $\Gamma \vdash loc : s$ for some type s. Thus, $loc \in dom(\Gamma)$, and part 2 of $\Gamma \approx S$ implies $loc \in dom(S)$. Let $S(loc) = [s' \,.\, F]$. Now $s' = s$ by part 1(a) of $\Gamma \approx S$.

Because $loc.m(\ v_1, \ldots, v_n\ )$ is well typed, we know by the hypotheses of T-CALL that $methodType(s, m)$ yields an n-arity method type. Thus, $\langle e, J, S\rangle$ evolves by CALL_A.

Case 2—$e = \mathbb{E}[\mathit{chain}\ \bar{B}, j(\ v_0, \ldots, v_n\ )]$. If $\bar{B}$ is non-empty, then $\langle e, J, S\rangle$ evolves by ADVISE. Otherwise, we must consider cases based on the value of j. By Lemma 14 (Join Point Abstractions) on page 36, there are two cases:

— $j = (\!|\mathsf{exec}, v, m, l, \tau|\!)$: By Lemma 14, $l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e : \tau$. Thus, $\langle e, J, S\rangle$ evolves by EXEC_B.

— $j = (\!|\mathsf{call}, -, m, -, \tau|\!)$: There are two subcases. If $v_0 = \mathsf{null}$, then $\langle e, J, S\rangle$ evolves by NCALL_B to a triple with a NullPointerException. Otherwise, v_0 is a location. Because e is well typed we have $\Gamma \vdash v_0 : u'_0$ for some u'_0; this is by T-LOC with $v_0 \in dom(\Gamma)$. By $\Gamma \approx S$, $S(v_0) = [u'_0 \,.\, F]$. Let $\tau = t_0 \times \ldots \times t_n \to t$, where the arity is $n+1$ by T-CHAIN and the well-typedness of e. By Lemma 14, $methodType(t_0, m) = t_1 \times \ldots \times t_n \to t$. Also by T-CHAIN, $u'_0 \preccurlyeq t_0$. By the correspondence between the definitions of *methodType* and *methodBody*, and by the definitions of T-CLASS, T-MET, and *override*, it must be the case that there exists a fun term l such that $methodBody(u'_0, m) = l$. Therefore, $\langle e, J, S\rangle$ evolves by CALL_B in this subcase.

Case 3—$e = \mathbb{E}[\mathit{under}\ v]$. In this case, we only need to argue that the stack, J, is not empty. Note that under expressions are not part of the static syntax. These expressions are only introduced during the evaluation of a program, by rule BIND, EXEC_B, and ADVISE. Each of those rules also pushes a join point abstraction onto the stack. The UNDER rule removes the under expression and pops the stack. Thus, the size of the stack corresponds to the number of under expressions present in the expression. The presence of an under expression in the evaluation context implies that the stack is non-empty. Therefore, $\langle \mathbb{E}[\mathsf{under}\ v], j + J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J, S\rangle$ by rule UNDER. □