A design discipline and language features for modular reasoning in aspect-oriented programs

Curtis Charles Clifton

TR #05-15
July 2005

Keywords: MAO discipline, MiniMAO calculus, aspect-oriented programming, AspectJ, spectators, assistants, AspectJML, modular reasoning

2003 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented programming—aspect-oriented programming; D.3.1 [*Programming Languages*] Formal Definitions and Theory—Semantics D.3.2 [*Programming Languages*] Language Classifications—object-oriented languages, Java, AspectJ; D.3.3 [*Programming Languages*] Language Constructs and Features—control structures, modules, packages, procedures, advice, spectators, assistants, aspects.

The Author's PhD Dissertation

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

A design discipline and language features for modular reasoning in aspect-oriented programs

by

Curtis Charles Clifton

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Gary T. Leavens, Major Professor
Jien Morris Chang
Markus Lumpe
Robyn R. Lutz
Jonathan D. H. Smith

Iowa State University
Ames, Iowa
2005

Graduate College
Iowa State University

This is to certify that the doctoral dissertation of
Curtis Charles Clifton
has met the dissertation requirements of Iowa State University

_____
Major Professor

_____
For the Major Program

To Pastor Daniel Solomon,
a model of perseverance and faithful dedication,
and a good friend

TABLE OF CONTENTS

LIST OF TABLES ix

LIST OF FIGURES xi

ACKNOWLEDGMENTS xiv

ABSTRACT xv

CHAPTER 1. INTRODUCTION 1

1.1 Aspect-oriented Programming 2

1.2 Practical Benefits 3

1.2.1 Applications of Aspect-Oriented Features 3

1.3 Theoretical Challenges 4

1.3.1 Object-Oriented Non-modularity 5

1.3.2 Behavioral Subtyping 6

1.3.3 Non-modularity in Aspect-Oriented Languages 7

1.3.4 Modular Aspect-oriented Reasoning 8

1.4 Scope 9

1.5 Statement of the Thesis 9

CHAPTER 2. THE MAO DISCIPLINE 11

2.1 The Discipline 11

2.2 Proposed Language Features 12

2.2.1 Assistants 14

2.2.2 Spectators 19

2.3 Evaluation 21

2.3.1 ATLAS Case Study 22

2.3.2 Impact of Restrictions 22

2.3.3 Summary of Evaluation 27

2.4 Specification and Reasoning 28

2.4.1 Specifying Around Advice 28

2.4.2 Specification Composition 32

2.5 Discussion 44

2.5.1 Language Issues 45

2.5.2 Specification Issues 47

2.5.3 Tool Support 47

2.6 Related Work 48

2.7 Conclusion 50

CHAPTER 3. MINIMAO$_1$: INVESTIGATING THE SEMANTICS OF PROCEED 53
3.1 MiniMAO$_0$: A Core Object-Oriented Calculus with Classes 53
3.1.1 Syntax of MiniMAO$_0$ 53
3.1.2 Operational Semantics of MiniMAO$_0$ 55
3.1.3 Static Semantics of MiniMAO$_0$ 62
3.1.4 Meta-theory of MiniMAO$_0$ 64
3.2 MiniMAO$_1$: Adding Aspects 72
3.2.1 Syntax of MiniMAO$_1$ 73
3.2.2 Operational Semantics of MiniMAO$_1$ 75
3.2.3 Static Semantics of MiniMAO$_1$ 92
3.2.4 Meta-theory of MiniMAO$_1$ 95
3.3 Related Work 117
3.4 Discussion 118
3.5 Conclusion 119

CHAPTER 4. MINIMAO$_2$: PARTITIONING THE HEAP BY CROSS-CUTTING CONCERNS 121
4.1 Intuition 121
4.2 Syntax 124
4.2.1 Public Concern Domain Declarations 124
4.2.2 Class and Aspect Instantiation 124
4.2.3 Refined Types 127
4.2.4 Effects Clauses 128
4.2.5 New Pointcut Descriptor 128
4.2.6 Concern Domain Dependencies 129
4.3 Semantics 129
4.3.1 Operational Semantics 129
4.3.2 Static Semantics 138
4.4 Meta-theory 151
4.4.1 Auxiliary Definitions and Lemmas 151
4.4.2 Type Safety 176
4.4.3 Effects Properties 187
4.5 Related Work 196
4.6 Conclusion 199

CHAPTER 5. MiniMAO$_3$: SPECTATORS REALIZED 201
5.1 Differences Versus MiniMAO$_2$ 201
5.1.1 Syntax of MiniMAO$_3$ 202
5.1.2 Operational Semantics of MiniMAO$_3$ 203
5.1.3 Static Semantics of MiniMAO$_3$ 208
5.2 Meta-Theory of MiniMAO$_3$ 213
5.2.1 Supporting Definitions and Lemmas 213
5.2.2 Type Safety 220
5.2.3 Effects 225
5.3 Discussion 231
5.4 Related Work 232

5.5 Conclusion 232
CHAPTER 6. CONCLUSIONS AND FUTURE WORK 233
6.1 Support for the Thesis 233
6.2 Open Problems 234
6.2.1 Verification 235
6.2.2 MAO 237
6.3 Future Work 239
6.3.1 Alias Control 239
6.3.2 Late Binding and Aspect-Oriented Virtual Machines 239
6.3.3 Concurrent Aspect-oriented Programming 240
6.3.4 Subtype Matching in Around? Unsound! 240
6.3.5 Component-based Programming 241
6.4 Postscript 242
BIBLIOGRAPHY 243

LIST OF TABLES

2.1 Categorization of Examples from the AspectJ Programming Guide 24
2.2 Categorization of Examples from Kiselev's Text 25

4.1 Reach and Writable Reach for the Store, *S*, of Figure 4.24 191

LIST OF FIGURES

1.1 Point Class . . . 6
1.2 Sample Client Code . . . 6
1.3 RightMovingPoint Class . . . 7
1.4 OneWayMoving Aspect . . . 8

2.1 FigureElement Example (in Java with JML annotations) . . . 13
2.2 MoveLimiting Example (in AspectJ) . . . 15
2.3 Syntax of Concern Maps . . . 18
2.4 Example Concern Map . . . 18
2.5 Example Spectator Aspect . . . 20
2.6 Example of JML Specification Cases . . . 29
2.7 Example JML Specification Showing Overlapping Specification Cases . . . 29
2.8 Around Advice Specification in AspectJML . . . 31
2.9 Example Specification for Around Advice with Multiple proceed Expressions . . . 32
2.10 Specification Composition Graph Construction, Stage G_0 . . . 35
2.11 Specification Composition Graph Construction, Partially Complete Stage G_1 . . . 36
2.12 Specification Composition Graph Construction, Stage G_1 . . . 37
2.13 Unique Path through Specification Composition Graph, α-converted . . . 39
2.14 General Form of the Effective Specification . . . 42
2.15 Effective Specification for the Path Shown in Figure 2.13 . . . 44
2.16 Simplified Version of Effective Specification from Figure 2.15 . . . 45
2.17 Effective Specification Derived from the Specification Composition Graph in Figure 2.12 45

3.1 Syntax of MiniMAO$_0$ . . . 54
3.2 Operational Semantics of MiniMAO$_0$ . . . 56
3.3 Auxiliary Functions for MiniMAO$_0$ . . . 58
3.4 Subtyping in MiniMAO$_0$ . . . 59
3.5 Sample MiniMAO$_0$ Program . . . 59
3.6 Static Semantics of MiniMAO$_0$ . . . 63
3.7 Syntax Extensions for MiniMAO$_1$ . . . 74
3.8 Join Point Stack . . . 75
3.9 Additional Expression Forms for the Operational Semantics of MiniMAO$_1$ . . . 76
3.10 Changes to the Operational Semantics for MiniMAO$_1$ . . . 78
3.11 Additional Subtyping Rule for MiniMAO$_1$ . . . 78
3.12 Auxiliary Functions for MiniMAO$_1$ Operational Semantics . . . 79
3.13 Boolean Algebra over Binding Terms . . . 83

3.14 Pointcut Descriptor Matching for MiniMAO$_1$ . . . 84
3.15 Comparison of Evaluation in MiniMAO$_0$ and MiniMAO$_1$ . . . 86
3.16 Sample Program Showing Advice Binding . . . 87
3.17 Sample Derivation of Pointcut Descriptor Matching . . . 88
3.18 Sample Program Showing Advice Chaining . . . 89
3.19 Sample Program Contrasting this vs. target Binding and call vs. execution Advice . . . 90
3.20 Additions to the Static Semantics for MiniMAO$_1$ . . . 93
3.21 Binding for Type Environments . . . 94
3.22 Static Semantics of Pointcuts in MiniMAO$_1$ . . . 96
3.23 Meta-variables Used in the Proof of Lemma 3.15 . . . 100

4.1 Schematic View of a Store in MiniMAO$_2$ . . . 123
4.2 Syntax of MiniMAO$_2$ . . . 125
4.3 Fragment of a MiniMAO$_2$ Program Illustrating New Syntax . . . 126
4.4 Syntax Extensions for the Operational Semantics of MiniMAO$_2$ . . . 130
4.5 Join Point Stack in MiniMAO$_2$ . . . 131
4.6 Example of Advice Table Construction . . . 132
4.7 Evaluation Relation for the Operational Semantics of MiniMAO$_2$ . . . 133
4.8 Evaluation Relation for the Operational Semantics of MiniMAO$_2$ (Exceptional Rules) . . . 134
4.9 Auxiliary Functions for Operational Semantics of MiniMAO$_2$ . . . 136
4.10 More Auxiliary Functions for MiniMAO$_2$ Operational Semantics . . . 137
4.11 Subtyping in MiniMAO$_2$ . . . 138
4.12 Pointcut Descriptor Matching for MiniMAO$_2$ . . . 139
4.13 Bindings in MiniMAO$_2$ . . . 140
4.14 Static Semantics of Declarations in MiniMAO$_2$ . . . 142
4.15 Auxiliary Functions for Static Semantics of MiniMAO$_2$ . . . 144
4.16 Auxiliary Typing Judgments for Declarations in MiniMAO$_2$ . . . 144
4.17 Static Semantics of Expressions in MiniMAO$_2$ . . . 147
4.18 Sample Expression Type Errors in MiniMAO$_2$ . . . 148
4.19 Binding for Type Environments . . . 149
4.20 Static Semantics of Pointcuts in MiniMAO$_2$ . . . 150
4.21 Venn Diagram Illustrating Lemma 4.6 . . . 153
4.22 Setup and Common Meta-variable Bindings Used in the Proof of Lemma 4.14 . . . 163
4.23 Auxiliary Functions for the Meta-theory of MiniMAO$_2$ . . . 190
4.24 Schematic View of a Sample Store, S . . . 190
4.25 Recursive Definition of the Locations Included in an Expression . . . 192

5.1 Differences in Syntax of MiniMAO$_3$ vs. MiniMAO$_2$ . . . 202
5.2 Differences in the Operational Semantics of MiniMAO$_3$ vs. MiniMAO$_2$ . . . 204
5.3 Pointcut Descriptor Matching for Surround Advice . . . 207
5.4 Example Illustrating Relaxed Pointcut Matching for Surround Advice . . . 208
5.5 Auxiliary Functions for the Static Semantics of MiniMAO$_3$ . . . 209
5.6 Differences in the Static Semantics of MiniMAO$_3$ vs. MiniMAO$_2$ . . . 210
5.7 Static Semantics of Pointcuts for Surround Advice . . . 212
5.8 Setup and Common Meta-variable Bindings Used in the Proof of Lemma 5.6 . . . 217

5.9 Type Derivation for Result of SURROUND Rule in Subject Reduction Proof 222

6.1 Array Co- and Contravariance Problem in Java 240

ACKNOWLEDGMENTS

I want to offer my sincerest thanks to several people and one dog for their contributions to this work, and to my reaching this milestone:

— to my wife Lisa Laxson, whose financial support made this feasible, and whose love, patience, dedication, and confidence in me made this possible;

— to our beagle Molly, who can always make me smile and who sleeps enough to make up for my lack of the same;

— to Gary Leavens, for his guidance, support, encouragement, patience and confidence in me, and for his extreme dedication during my sprint to the finish;

— to my parents, Jim and Connie Clifton, for their love and encouragement and for teaching me the value of hard work, striving for perfection, and service to others;

— to Janet (Lamont) Leonard, for putting an Apple II in front of a scrawny eleven-year-old boy and changing his path in life forever;

— to the faculty, staff, and most especially the students in the Dept. of Computer Science and Software Engineering at the Rose-Hulman Institute of Technology, whose enthusiasm gave me the final push to complete this work;

— to Morris Chang, Markus Lumpe, Robyn Lutz, and Jonathan Smith for their contributions on my committee, and for not groaning too loudly when they saw the thickness of the final draft;

— to Jerry Anderson, for keeping me grounded and Atanasoff Hall spotless;

— to Linda Dutton, for help in clearing the administrative hurdles and for providing interesting conversation and a ready supply of Altoids Sours;

— to the friendly staff of Taraccino Coffee in Ames, especially Laurie, Rigel, Kelsey, and Jeremiah, for "proudly serving legal addictive stimulants since 1997" and getting me started with my writing most every morning;

— to Gerben Wierda for his outstanding LaTeX distribution and to the BibDesk team, in particular Michael McCracken, Adam Maxwell, and Christiaan Hofman, for their volunteer efforts to making Mac OS X an outstanding platform for mathematical typesetting; and

— to God for His many gifts, including the people who have nurtured me along the way, strength for the final push, and Jesus Christ, the perfect example of compassion and loving service.

ABSTRACT

Aspect-oriented programming lets programmers modularize concerns that are orthogonal to the main decomposition of a program. To do this, aspect-oriented programming includes modules called aspects that may modify the behavior, or advise, code in the main decomposition. Aspect-oriented programming also allows aspects to declaratively specify what code should be advised. This means that a whole-program search is required to find all the aspects that might advise a given piece of code. The problems this causes are somewhat analogous to overriding methods and polymorphic method dispatch in traditional object-oriented programming.

In object-oriented programming, the discipline of behavioral subtyping permits reasoning about polymorphic methods even when overriding methods remain unseen. The discipline gives guidance to the author of an overriding method: the overriding method must satisfy the specification of the overridden, superclass method. If the author follows the discipline, then other programmers can reason about a method invocation based on the specification of the superclass method, even if an unseen overriding method might actually be executed.

This dissertation describes an analogous discipline for aspect-oriented programming. The basic premise is that modular reasoning about aspect-oriented programs requires shared responsibility between the aspect author and the client programmer, whose code might be advised by the aspect.

To mediate this sharing, this dissertation proposes that aspects be categorized into two sorts: “spectators” and “assistants”. Spectators are statically restricted to not modify the behavior of the code that they advise. Because of their restricted behavior, spectators may remain unseen by the client programmer. The burden is on the aspect programmer to ensure that spectators satisfy their restrictions. Unlike spectators, assistants are not restricted in their behavior. The burden of reasoning about their effects falls to the client programmer. To do this, the client programmer must be able to identify all applicable assistants. Thus, assistants must be explicitly accepted by the advised code. This discipline allows modular reasoning, permits the use of existing aspect-oriented idioms, and appears to be practical and statically verifiable. A formal study demonstrates that the restrictions on spectators may be statically checked.

CHAPTER 1. INTRODUCTION

Aspect-oriented programming [81] deals with the problem of modularizing "cross-cutting concerns". Cross-cutting concerns are features of a program that are orthogonal to its main decomposition [153]. Because of their orthogonality, cross-cutting concerns inherently result in the *scattering* of code across various modules in the main decomposition of the program. From the perspective of the main decomposition, such cross-cutting code is said to be *tangled*. Aspect-oriented programming addresses this problem by allowing software engineers to write code for cross-cutting concerns in separate modules, called aspects, and to declaratively specify how that code is to be associated with events in the main decomposition at run time.

The declarative association of aspects with the code for the main decomposition is a powerful technique for eliminating the scattering of code. It can result in modules that are more focused and concise. However, the technique can also be used to write code that is difficult to understand and maintain. For example, the declarative manner in which aspects are introduced means that a programmer must, in general, have whole-program knowledge to reason about any operation in the main decomposition.

In the early years of object-oriented programming, similar concerns were raised about polymorphic method invocations. The discipline of behavioral subtyping [11, 12, 50, 92, 105, 109] evolved to address these concerns. Behavioral subtyping places a constraint on subtype programmers: overriding methods in a subtype must satisfy the specification of the overridden supertype methods. In exchange for programming within this constraint, clients of the supertype may reason about invocations of supertype methods without worrying about the effects of overriding methods in unseen subtypes. Behavioral subtyping provides both practical guidance to programmers and formal soundness for theorists.

Despite the theoretically increased complexity in reasoning introduced by aspect-oriented languages, programmers are putting them to use in a wide variety of projects [20, 86, 147]. This indicates that there are practical mechanisms for controlling aspect-oriented complexity. In fact, as Laddad [86, §1.3.3] points out, well-written aspects consolidate code for a common concern that would be scattered in an object-oriented implementation. Thus, although a whole program analysis is needed with an aspect-oriented program, such an analysis would also be needed to find the scattered code in an object-oriented implementation.

In the subsequent section I present more background information on aspect-oriented programming, including terms and history. Then I discuss how the practical benefits of aspect-oriented programming are drawing a growing user community and consider how the members of that community are applying aspect-oriented techniques. Next I describe modular reasoning, give more detail on how behavioral subtyping allows modular reasoning in object-oriented programming, and consider the theoretical challenges presented by aspect-oriented programming. I conclude the introduction by defining the scope of my work, stating my thesis, and outlining the rest of this dissertation.

1.1 Aspect-oriented Programming

Separation of concerns is the well-known software engineering concept that code for different subdomains, or aspects, of a problem should be made as independent as possible to encourage comprehensibility and efficiency (in both re-use and parallel development) [132, 133]. Object-oriented languages encourage the separation of concerns into code representing individual objects in a model of the problem domain. However, there are some aspects which cross-cut the decomposition of a problem domain into objects [68, 81, 153]. Common examples of such cross-cutting concerns are logging, tracing, persistence, and what Filman et al. [60] call *ilities*: reliability, availability, and manageability among others. The subfield that arose to deal with this problem was known as *advanced separation of concerns*.

Harrison and Ossher [68] describe a programming paradigm that they call *subject-oriented programming*. Subject-oriented programming generalizes the object-oriented paradigm. A subject is roughly equivalent to an entire program in an object-oriented language in that all code within that subject shares the same set of class and type hierarchies, operations, and object state. What makes subject-oriented programming unique is that disparate subjects—with distinct class and type hierarchies, operations, and object state—can share access to the same set of objects. The only property necessarily shared by subjects on a given object is object identity.

Various composition rules are used to combine subjects into programs. These rules specify mappings between class and type hierarchies in the composed subjects and describe how method dispatch from within one subject impacts the other subjects in the composition. For example, suppose several subjects each declared an operation with the same name and arguments for a given object. A composition rule might specify that an invocation of this operation in one subject should also execute the code for this operation in the other subjects. More complex composition rules can be imagined that map between operations of different names and parameters and specify compositions of return types of the methods.

Subject-oriented programming has been realized in the Hyper/J language [129, 153] and the Concern Manipulation Environment [72].

The term *aspect-oriented programming* was coined by Kiczales et al. [81]. Aspect-oriented languages provide support for advanced separation of concerns via *aspects*. An aspect may specify additional code to be executed at "certain well-defined points in the execution of the program" [83, p. 329] known as *join points*. The construct for declaring this additional code is called *advice*. A piece of advice defines the set of join points at which the advice should be executed, known as its *pointcut*. Advice includes a *pointcut description*, made up of primitive predicates called *pointcut descriptors*, that defines its pointcut. A piece of advice is sometimes said to *advise* the join points in its pointcut. (Later examples will illustrate these concepts.)

Advice provides support for a sort of pattern-based metaprogramming, allowing one to specify, for example, that a certain body of code should be executed whenever a method whose name begins with the string open is invoked. An aspect may also introduce new methods to existing classes without modifying those classes, thus supporting open classes [43, 114]. The aspect-oriented approach to advanced separation of concerns is typified by the language AspectJ [14, 83].

In previous work on MultiJava [38, 43, 46], colleagues and I demonstrated how to extend Java [13, 64] with open classes and multimethods. Unlike AspectJ, MultiJava supports lexical scoping of its open class extensions. AspectJ weaves extensions into the extended class, where they will be visible to all clients of the extended class. Instead in MultiJava, extensions are only available to a client that explicitly imports them. This avoids polluting the interface of the extended class with extensions that are not needed by all clients. On the other hand, the explicit import of extensions in MultiJava can represent code tangling (e.g., if the extension represents a cross-cutting concern).

AspectJ and MultiJava can both be viewed as incremental approaches toward the more general subject-oriented philosophy. AspectJ maintains the central control structure of a single program, but allows additional operations and state to be in separate aspects. The dispatch flexibility of subject-oriented programming's composition rules is achieved through AspectJ's join points. To provide this flexibility AspectJ requires a whole program analysis. MultiJava's open classes allow additional operations to be specified via its open classes while maintaining modular static typechecking and compilation, though its extension mechanisms are more limited than AspectJ's.

Beginning in 2002, the term *aspect-oriented software development* came into use to describe all of the various approaches for advanced separation of concerns [79]. In addition to Hyper/J and AspectJ, other languages for aspect-oriented software development include Composition Filters [17], DemeterJ [101], CaesarJ [113] and a host of languages applying AspectJ-like enhancements to core languages other than Java [15, 47, 149, 154].

Filman and Friedman [59] have tried to identify the distinguishing features of languages for aspect-oriented software development. They assert that such languages are characterized by two features:

— *quantification*, declarative specification of a set of points, in either the static code or the dynamic control flow graph of a program, where aspect-oriented code is to be added; and

— *non-invasiveness*,[1] the execution of additional aspect-oriented code at a program point, P, without effort by the programmer of the code containing P.

These two characteristics can complicate reasoning about aspect-oriented programs. Non-invasiveness implies that a programmer may not be able to determine from local code what aspect-oriented code might be executed. Quantification implies that even a whole-program search may not definitively identify whether any aspect-oriented code might be executed—for example, if the quantification is over events in the program's dynamic call graph. Section 1.3 on the following page discusses these complications in more detail. But first I discuss the practical benefits afforded by the aspect-oriented paradigm.

1.2 Practical Benefits

Despite the apparent complexity in reasoning about aspect-oriented programs, the paradigm is being widely adopted. This is evidenced by the volume of traffic on the AspectJ mailing lists, AspectJ development environment downloads exceeding 20,000 per month,[2] and the variety of commercial software frameworks that include aspect-oriented features [21, 29, 120, 123]. No less an authority than Daniel Sabbah, IBM Vice President of Strategy and Technology, Software Products Group, said, "Aspect-oriented programming is vital to the success of our business" [147].

1.2.1 Applications of Aspect-Oriented Features

Based on a review of code in the literature and discussions with programmers using aspect-oriented languages, it seems that the common applications of these languages may be classified in two ways:[3]

[1] Filman and Friedman use the term "obliviousness"; however, non-invasiveness is becoming the more accepted term for this concept within the community, perhaps for obvious reasons.

[2] Download statistics posted to the aspectj-users mailing list, May 2005.

[3] Thanks to Arno Schmidmeier, Juri Memmert, Karl Lieberherr, Frank Sauer, and others for discussions at AOSD '02 on the ways they are using aspect-oriented programming.

— *Code Recycling*: Aspects are used to transform the behavior or interface of an existing program without modifying the source code of that program.

— *Separation of Concerns*: Aspects are used to separate the code for a program into syntactically distinct modules, each dealing solely with a particularly concern.

I discuss each application briefly below.

1.2.1.1 Code Recycling

By virtue of their ability to insert code into an existing program, aspects can be used to transform both the behavior and interface of legacy software. Tzilla Elrad said, at FOAL 2004, that while object-oriented programming allows code re-use, aspect-oriented programming allows "code recycling".

For example, one might use an aspect to add authentication to an existing web server [84] [86, Ch. 10]. Kiselev's "runtime aspects" provide another example. A more elaborate application might involve using aspects to modify the application programming interface (API) of a program by introducing additional methods and redirecting existing calls to the new methods.

I consider program rewriting to be beyond the scope of this work and will instead focus on the second main application of aspect-oriented languages: separation of concerns.

1.2.1.2 Separation of Concerns

Experienced aspect-oriented programmers separate concerns into orthogonal aspects. For example, the base program might handle the functional concerns of the problem domain, while separate aspects might handle persistence, logging, and security. A persistence concern might be separated by writing an aspect that advises the main entry point of a program to establish a database connection. Additional advice could advise object factories in the main program to return instances from the database in response to requests and advise state-changing methods to store mutated objects back to the database [86, ch. 11][142]. A logging concern might be separated from a base program by quantitatively specifying all the program points to be logged. If the logging architecture should need to be changed, for example from using println calls to using a more sophisticated framework, all changes are localized in the logging aspect [86, §5.4.1]. Thus, properly separated concerns support evolution of code.

Also, by using different global configurations—combinations of aspects and classes—one can use the separation-of-concerns style to generate a variety of systems from a common code base. For example, a programmer might treat customer-specific requirements as a separate concern and then construct different implementations of that concern for each customer.

Application of aspects for separation of concerns is analogous to the use of behavioral subtypes in object-oriented languages; both techniques seek to enhance existing behavior without introducing surprising behavior. Both rely on underspecification: A behavioral subtype can add behavior that is not reserved by the supertype's specification; aspects can add behavior that deals with concerns that are orthogonal to those of the advised methods.

1.3 Theoretical Challenges

Properly written object-oriented code—code satisfying behavioral subtyping—makes reasoning easier, despite a theoretical increase in complexity of reasoning versus procedural programming. Similarly, properly

written aspect-oriented code should make reasoning easier, despite a theoretical increase in complexity of reasoning versus object-oriented programming. In this section I describe the increase in reasoning complexity that results from polymorphic method invocations in object-oriented languages. I then discuss how the discipline of behavioral subtyping makes reasoning easier in these languages. I use this as an analogy to consider reasoning in aspect-oriented languages. (The use of this analogy arose in work joint with Gary Leavens [41].)

First I need to clarify my terminology. It is nebulous to say that reasoning is "easier" in one case versus another. To be concrete, I need a definition of modular reasoning. There are various definitions of modularity in the literature. The weakest definition might be that an analysis is modular if the portion of the program that must be considered to perform the analysis is a well-defined, proper subset of the whole program. At the other end of the spectrum, the definition might require that an analysis must consider only the code for a single compilation unit. Work that uses this definition, particularly in the area of component-based programming [63], typically requires a compilation unit to declare its expectations of external modules [152]. These expectations can then be verified during composition of the unit with external modules.

I will use a definition between these two extremes: a language allows *modular reasoning* if it is possible to reason about a compilation unit in that language based on the code of that compilation unit and the specifications of any *modules* (e.g., classes, interfaces, and packages) referred to by that compilation unit. A compilation unit *refers* to a module M if it explicitly names M, is lexically nested within M, or if M is a standard module in a fixed location (such as Object in Java). Java [13, 64] with JML [93, 95] and Eiffel [111] both satisfy this definition.

It is easier to program in languages that allow modular reasoning, since the cognitive burden on the programmer is reduced—the specifications of referenced modules serve as behavioral abstractions of all the code implementing those modules.

1.3.1 Object-Oriented Non-modularity

In typical (single-dispatch) object-oriented languages, the dynamic type of the receiver object is used to select the appropriate method to execute for a given invocation. Such dynamic selection of the target method can prevent modular reasoning. For example, consider the declaration of Point in Figure 1.1 on the next page and its method, move. The //@-comments before move's declaration give its behavioral specification in JML [93, 94].

— The clause "requires true" says that clients are not obliged to establish any precondition.

— The clause "assignable pos" says that the pos field of the object, but no other locations, may be changed by the method.

— The clause "ensures ..." says that, after move returns, the value returned by getPos() is equal to the sum of the dist argument and the value returned by getPos() before move was called.

(Formal specifications as used here are not a necessary condition for modular reasoning. The behavior of a module can be thought of concretely as its code. Often programmers reason about modules using informal abstractions, e.g., "This method returns true if the given file exists". In a more expressive language, such as Eiffel [110] or Java annotated with JML as used here, the abstract behavior can be specified using pre- and postconditions, frame axioms, and invariants; such specifications serve as contracts that allow one to separately reason about the behavior and correctness of an implementation.)

```
public class Point {
   private /*@ spec_public @*/ int pos;
   public final /*@ pure @*/ int getPos() {
      return pos;
   }

   ...

   //@ requires true;
   //@ assignable pos;
   //@ ensures getPos() == dist + \old(getPos());
   public void move(int dist) {
      pos = pos + dist;
   }
}
```

Figure 1.1 Point Class

```
public void client(Point p) {
   ...
   assert p.getPos() == 0;
   p.move(-10);
   assert p.getPos() == -10;
}
```

Figure 1.2 Sample Client Code

Suppose an object of static type Point is passed to a method client, as in Figure 1.2. If modular reasoning is sound, then the programmer can reason about the invocation of move based on its specification in Point. That is, if the first assertion in the figure holds, then the second assertion is valid based on the specification of Point's move method. The definition of modular reasoning requires that the programmer should not have to consider possible unseen subtypes of Point when reasoning, since they are not mentioned in the client code. But, by subsumption, an instance of just such an unseen subtype may be passed to client. What if (as in Figure 1.3 on the next page) the subtype RightMovingPoint overrides method move, but does not satisfy the specification of move in Point? Then modular reasoning such as that described for client is not valid. If an instance of RightMovingPoint is passed to client, then after the invocation of p.move(-10), the assertion fails: p.getPos() returns 10, not -10.

1.3.2 Behavioral Subtyping

Modular reasoning is not an inherent property of object-oriented languages. However, the discipline of behavioral subtyping restores sound modular reasoning by imposing the specification of Point on all its subtypes [50, 92, 105]. RightMovingPoint does not correctly implement a behavioral subtype of Point, because its implementation does not satisfy the specification of move in Point. Behavioral subtyping is often described

```
public class RightMovingPoint extends Point {
   public void move(int dist) {
      if (dist < 0) super.move(-dist);
      else super.move(dist);
   }
}
```

Figure 1.3 RightMovingPoint Class

by saying that the behavior of a subtype should not be surprising with respect to the specified behavior of a supertype. Behavior is *surprising* if the (possibly unseen) code executed in response to a method invocation fails to satisfy the visible method's specification.

Two complementary notions of behavioral subtyping (for types with mutable objects) have been proposed. Liskov and Wing [105] propose a notion of behavioral subtyping that does not allow subtypes to mutate state inherited from immutable types. Their notion does not place restrictions on aliasing. Dhara and Leavens [50] propose weak behavioral subtyping[4] that allows such mutation by subtypes but imposes some restrictions on object aliasing to avoid surprising behavior. Dhara and Leavens [51] also propose specification inheritance as a mechanism for enforcing weak behavioral subtyping. The same authors provide a review of the research on behavioral subtyping [91, §6.3] and a particularly clear characterization of modular reasoning (§6.1.3).

As pointed out by Filman and Friedman [59, §2.2], subtyping with subsumption, as in the Point example, is a form of non-invasiveness. Aspect-oriented programming languages allow programmers much greater latitude in defining behaviors with unseen code.

1.3.3 Non-modularity in Aspect-Oriented Languages

Just as modular reasoning is not a general property of object-oriented programming languages in the absence of behavioral subtyping, modular reasoning is not a general property of aspect-oriented languages. To show this, I present an aspect-oriented extension to the Point example.

Figure 1.4 on the following page gives an aspect, OneWayMoving, that modifies the behavior of Point instances in the same way as RightMovingPoint [39, 41]. OneWayMoving declares a piece of *around advice*. This advice intercepts calls to Point's move method. If the argument passed to the client is negative, then, just as in RightMovingPoint, control proceeds to Point's move method with the parameter set to the absolute value of the original parameter. As with RightMovingPoint, the client programmer's reasoning in Figure 1.2 on the preceding page is not correct in the presence of the OneWayMoving aspect.

In AspectJ the advice is applied by the compiler without explicit reference to the aspect from either the Point module or a client module. Instead the classes and aspect are simply passed as arguments to the same compiler invocation, perhaps under the control of an Integrated Development Environment (IDE) or build system. Thus, modular reasoning about the Point module or a client module has no way to detect that the behavior of the move method will be changed when the Point module and OneWayMoving are compiled together. In AspectJ the programmer must potentially consider all such aspects and the Point class together in

[4] The "weak" in weak behavioral subtyping implies that this formulation is less restrictive than that of Liskov and Wing.

```
public aspect OneWayMoving {
   void around(int dist): call(void *.move(int)) && args(dist) {
      if (dist < 0) proceed(-dist);
      else proceed(dist);
   }
}
```

Figure 1.4 OneWayMoving Aspect

order to reason about the Point module. Some potentially applicable aspects, such as OneWayMoving, may not even name Point directly, but instead may use wild card type patterns.

Therefore, just as in object-oriented programming without behavioral subtypes, the non-invasiveness of aspect-oriented languages can prevent modular reasoning.

1.3.4 Modular Aspect-oriented Reasoning

The OneWayMoving aspect represents poor aspect-oriented programming, just as the RightMovingPoint class in Figure 1.3 on the preceding page represents poor object-oriented programming. This is because both examples change the behavior of move with regard to a Point's position; behavior that is, by virtue of move's strong specification, restricted to Point itself.

Behavioral subtyping serves to formalize the programming discipline that allows modular reasoning about object-oriented programs. This, in turn, provides useful insights into how best to harness the power of the object-oriented paradigm. In this work I provide the formal basis of a similar discipline for aspect-oriented programming. I do this by defining a small set of language extensions. I explicate the design as a series of core languages, each with a sound, static type system. I demonstrate the utility of the design for modular reasoning by proving some modularity properties that cannot be shown to hold in a core aspect-oriented language that omits the extensions (but closely models key features of AspectJ). Finally, I consider the implications of this formal work for future aspect-oriented programming language design and for aspect-oriented specification and verification.

We have seen that experienced aspect-oriented programmers separate concerns into orthogonal aspects. Orthogonality of these aspects helps the reader of a program to understand it, provided she can find the applicable aspects. Specifically, if she wants to reason just about the functional behavior of a code fragment, she must just consider the base program code. If she is concerned with the persistence behavior of a code fragment in the base program, she must just consider the single aspect for persistence. Based on these observations, a detailed design discipline must have two key features:

— easy identification of applicable aspects, and

— orthogonality of the concerns expressed by those aspects.

The first feature is provided in some aspect-oriented languages, for example, in Hyper/J's module interconnect language and in Weave.NET's XML-encoded aspect bindings [87]. In AspectJ, the first feature is generally provided by tool support, though this limits the possible analyses to those supported by the tool. I demonstrate that the easy identification of applicable aspects can be more generally accommodated at the language level.

The second feature is possible in existing aspect-oriented languages (indeed, this is largely what makes them aspect-oriented). However, I am not aware of any languages that explicitly help software engineers to statically verify such orthogonality.

1.4 Scope

This dissertation focuses on what I call *aspect-oriented languages with dynamic-context pointcut descriptors*. In such languages:

— join points in the program may be specified in terms of the run-time call stack. Such join points are typified by AspectJ's cflow, cflowbelow, call, and execute pointcut descriptors.

— aspect-oriented code specifies changes in the behavior of some base program. Filman and Friedman [59] characterize this as *asymmetric* aspect-oriented programming. AspectJ is the prototypical asymmetric aspect-oriented language. Asymmetry stands in contrast to *symmetric* aspect-oriented languages, like Hyper/J, where the code for various concerns has equal standing and is composed to produce the final program. (In unpublished work, colleagues and I demonstrate that an asymmetric aspect-oriented programming language with names can model symmetric aspect-oriented programming [44, 45].)

By aspect-oriented languages with dynamic-context pointcut descriptors, I specifically do not include those languages that support the run-time insertion and removal of aspect-oriented code, so called "dynamic weaving" [57, 112, 113, 135, 138, 151]. I only consider the use of aspect-oriented code for separation of concerns (as discussed in Section 1.2.1 on page 3), not for "code recycling", and I assume that any base program can be refactored as needed. I also do not consider concurrency issues or "per" aspects, thus focusing on sequential aspect-oriented programs.

To keep the problem tractable, I do not consider aspect-oriented modeling and design [36, 37, 66, 150] and the various framework-based approaches to aspect-orientation [21, 29, 120, 123]. I choose to focus on programming language design instead. I also do not consider the decomposition of existing systems into cross-cutting modules as provided by Hyper/J and the Concern Manipulation Environment.

1.5 Statement of the Thesis

My thesis is that *there exists a discipline for programming in aspect-oriented languages with dynamic-context pointcut descriptors that (1) allows modular reasoning, (2) permits the use of existing aspect-oriented idioms for separation of concerns, (3) can be verified by a combination of static typechecking and simple verification conditions, and (4) can be incorporated into a practical, aspect-oriented language.*

I support my thesis in this dissertation by:

— describing such a discipline, which I call the "MAO discipline";

— presenting a small set of language features, as an extension to AspectJ (version 1.2), designed to facilitate the discipline;

— developing extensions to the Java Modeling Language for specifying features of aspect-oriented programs;

— sketching an algorithm for calculating the *effective specification* of an expression in my AspectJ extension, given the specifications for any potentially applicable advice;

— presenting MiniMAO$_1$, a core calculus that models AspectJ;

— designing an extended core calculus, MiniMAO$_2$, that includes *concern domains*, a type system for statically enforcing the separation of concerns; and

— formalizing my proposed extensions to AspectJ in another extended core calculus, MiniMAO$_3$.

For each of the core calculi, I describe, and prove sound, a static type system. I also prove key meta-theoretic properties of the extended calculi. These properties demonstrate the effectiveness of the MAO discipline and my proposed language features for modular aspect-oriented reasoning.

This dissertation is primarily formal in nature. However, in Chapter 2, I begin by informally describing the MAO discipline and my proposed language extensions, to both AspectJ and JML. Later chapters are dedicated to developing the formal machinery that supports the claims I make informally in the next one.

CHAPTER 2. THE MAO DISCIPLINE

I described, in Chapter 1, how object-oriented programming and aspect-oriented programming both present problems for modular reasoning. Polymorphic method calls in object-oriented programs allow unseen code (overriding methods) to affect a computation. Advice binding in aspect-oriented programs allows unseen code (the advice) to affect a computation. Both are examples of non-invasiveness.

The discipline of behavioral subtyping [11, 12, 50, 92, 105, 109] restores modular reasoning to object-oriented programming languages. It does this by requiring that an overriding method satisfy the specification that it inherits from the superclass method [51]. This discipline is enabled by the fact that *the superclass method is visible from the declaration of the overriding method.* This is a crucial fact. A class declares its superclass. The declaration allows the class to inherit or override methods from the superclass. So the class declaration containing an overriding method provides a reference to the overridden method.

Thus, the non-invasiveness in object-oriented programming only cuts one way. From a method call site, the actual code to be executed may be in an unseen, overriding method. However, from the declaration site of the overriding method, the superclass method is visible. Thus, the overriding method can satisfy the specification of the superclass method.

The non-invasiveness in aspect-oriented programming cuts both ways. From a method call site, the actual code to be executed may be in an unseen aspect. And from the declaration site of an aspect, because of quantification, the code to be advised may also be unseen. For example, the aspect might only advise code that implements some interface, and the code implementing that interface might not be known or even exist when the aspect is written. Thus, with aspect-oriented languages one cannot adopt the solution of behavioral subtyping: it is not enough to simply require that advice satisfy the specification of the code it augments.

This, then, is the core challenge in developing a programming discipline that allows modular reasoning about aspect-oriented programs. If reasoning is to be modular, then how can one reason about potentially advised code when (1) unseen aspects may apply to the code, and (2) aspects may be developed without (complete) knowledge of the code that will be advised?

2.1 The Discipline

A simple categorization of aspects lies at the heart of the *MAO discipline* for **m**odular, **a**spect-**o**riented reasoning. In the MAO discipline, I divide aspects into two sorts: those whose advice might introduce "surprising" behavior into a program, and those whose advice is "benign".[1]

In the discipline of behavioral subtyping all the burden of ensuring modular reasoning is placed on the author of an overriding method. A "client" programmer—that is, a programmer writing code that calls the

[1] In the subsequent section I will clarify what it means for advice to be benign, and propose statically verifiable restrictions to ensure that benign advice actually is so. Subsequent chapters of this dissertation focus on the formalization and proof of soundness for these restrictions.

method—may reason about a call without seeing the overriding method. The specification of the superclass method is normally sufficient for reasoning.

However, in the MAO discipline, the burden is shared. For benign advice, the advice author must satisfy the restrictions. Having done so, the client programmer can remain safely oblivious to the benign advice when reasoning about advised code. On the other hand, for advice that might introduce surprising behavior, the client programmer must be able to modularly identify what advice may apply. The programming language must allow this identification of surprising advice. Having identified this advice, the client programmer must compose the specifications of the advice and the advised code, thus finding the *effective specification* of the code in the presence of the advice.

The MAO discipline overcomes the two problems identified at the end of the previous section. (1) Aspects that would be unseen in regular AspectJ are divided into two kinds. Those with benign advice remain unseen, but do not affect the behavior of the code. Those with surprising advice must be made visible so that their effects can be considered. (2) For aspect development, the programmer of benign advice must satisfy a set of restrictions, but having done so, she does not need to consider the specific behavior of the code that will be advised. For surprising advice, she must simply satisfy the specification of the advice. The responsibility of reasoning about the interaction of the surprising advice and the advised code falls to the client programmer.

In this chapter I describe a small set of language features that allow a programmer to modularly identify all of the "surprising" advice that may apply to a given join point. The features also enable the static verification of the benignity of other advice. Section 2.2 describes my proposed language features. Because these features change the semantics of AspectJ, it is reasonable to wonder what effect they have on the expressiveness of the language. Section 2.3 evaluates my proposal against published examples of AspectJ code. This evaluation shows that my features do not result in any practical loss of expressiveness. To demonstrate the reasoning process in the MAO discipline, Section 2.4 proposes some simple extensions to JML (the Java Modeling Language [93, 95]) for giving behavioral specifications of advice, and formalizes specification composition. Section 2.5 discusses some additional issues related to AspectJ, specification, and tool support. Section 2.6 outlines related work, and Section 2.7 concludes. The basic ideas discussed in this chapter originated in a paper co-authored with Gary Leavens for the 2002 Foundations Of Aspect-oriented Languages workshop (FOAL '02) [39].

2.2 Proposed Language Features

In this section I describe some language features that are sufficient to support the MAO discipline. For concreteness I describe these features as extensions to AspectJ. I use a running example, introduced in Figure 2.1 on the facing page, that expands on the Point example from Chapter 1. FigureElements have a two-dimensional position and include a move method that makes "self calls" to get and set the x and y position fields. These self calls serve expository purposes later in the chapter.

The key feature to support modular reasoning in my proposal is to divide aspects into two sorts: spectators and assistants. "Spectators" are limited in that they may not change the behavior of the modules they apply to (in a way to be made more concrete later, and fully formalized in Chapter 5); their advice is benign. "Assistants" are not limited in this way. Since spectators do not change behavior, they preserve modular reasoning even when applied without explicit reference by the modules they view. Hence spectators preserve most of the flexibility of the current version of AspectJ. Because assistants can change the behavior of the modules to which they apply, to maintain modular reasoning they can only be applied in modules that reference them.

```
1  package mao;
2
3  public class FigureElement {
4
5     private /*@ spec_public @*/ float x = 0;
6     private /*@ spec_public @*/ float y = 0;
7
8     //@ requires true;
9     //@ assignable x, y;
10    //@ ensures (getX() == \old(getX()) + dx) && (getY() == \old(getY()) + dy)
11    //@                 && \result == this;
12    public FigureElement move(float dx, float dy) {
13       this.setX(getX() + dx);
14       this.setY(getY() + dy);
15       return this;
16    }
17
18    //@ requires true;
19    //@ assignable \nothing;
20    //@ ensures \result == x;
21    public /*@ pure @*/ float getX() {
22       return x;
23    }
24
25    //@ requires true;
26    //@ assignable \nothing;
27    //@ ensures \result == y;
28    public /*@ pure @*/ float getY() {
29       return y;
30    }
31
32    //@ requires true;
33    //@ assignable this.x;
34    //@ ensures getX() == x;
35    public void setX(float x) {
36       this.x = x;
37    }
38
39    //@ requires true;
40    //@ assignable this.y;
41    //@ ensures getY() == y;
42    public void setY(float y) {
43       this.y = y;
44    }
45 }
```

Figure 2.1 FigureElement Example (in Java with JML annotations)

2.2.1 Assistants

I call aspects that can change the behavior of a module *assistants*. The MoveLimiting aspect of Figure 2.2 on the next page is an assistant; it changes the behavior of FigureElement's move, setX, and setY methods to limit the maximum change in position from any single call. The term "assistant" is intended to connote a participatory role for these aspects.

What information is needed to modularly reason about behavior when assistants are present? Quite simply, a module must explicitly name those assistants that may change its behavior or the behavior of modules that it uses. I say that a module *accepts assistance* when it names the assistants that are allowed to change its behavior or the behavior of modules that it uses. Assistance may be accepted by either:

— the module to which the assistance applies (called the *implementation module*), or

— a client of that module.

2.2.1.1 Explicit Acceptance of Assistance

AspectJ does not currently include syntax for explicitly accepting assistance. Explicit acceptance of assistance can, however, be roughly simulated by the "hyper-cutting" pattern in AspectJ. In this pattern, one creates a marker interface, and the pointcuts of assistants would only apply to types that implement that interface [84, pp. 214–216]. An implementation module can then implement the marker interface, and thus indirectly accept the advice of the assistant. However, if a single client declares that the implementation module is a subtype of the marker interface (using the declare parent syntax of AspectJ), then the change affects all clients of the implementation module, but no trace appears in the implementation module; hence such changes are not modular.[2]

To automate this hyper-cutting pattern, and to avoid these non-modular uses of it, I propose a simple syntax extension for accepting assistance:

accept *TypeName*;

where *TypeName* must be the name of an assistant respecting Java's usual namespace rules for packages and imports [64, §6.5]. Multiple accept clauses may appear in a single module, following any import clauses. For example, the FigureElement module could accept the MoveLimiting assistant by declaring:

accept MoveLimiting;

I will generalize this idea with concern maps below.

When an implementation module accepts assistance, that assistance is applied to every applicable join point within the implementation module, regardless of the client making the call.

On the other hand, if the assistance is accepted by a client module, then that assistance is only applied to applicable join points in that client. Other clients that did not accept the assistance would not have it applied to their join points.

AspectJ includes two pointcut descriptors that roughly simulate this behavior. Advice on join points described via call pointcuts is woven into all client code. Advice on join points described via execution

[2] An upcoming version of AspectJ will take advantage of the annotation syntax of Java 5 [65]. Annotations on types could then be used in place of the marker interface approach. However, the modularity problem will likely remain, because the new version of AspectJ is planned to include annotation introductions.

```
1  public aspect MoveLimiting {
2     private static float MAX_DISTANCE = 10.0;
3     private static float distance(float x, float y) { ... }
4     /* Constrains distance of any single movement to MAX_DISTANCE */
5     FigureElement around(float argX, float argY) :
6        execution(* mao.FigureElement.move(float, float)) && args(argX, argY)
7     {
8        float moveDistance = distance(argX, argY);
9        if ( moveDistance > MAX_DISTANCE ) {
10          float ratio = MAX_DISTANCE / moveDistance;
11          return proceed( argX * ratio, argY * ratio );
12       } else {
13          return proceed( argX, argY );
14       }
15    }
16    /* Constrains distance of any x-axis movement to MAX_DISTANCE */
17    void around(mao.FigureElement targFE, float x) :
18       execution(* mao.FigureElement.setX(float)) && target(targFE) && args(x)
19    {
20       float currentX = targFE.getX();
21       if (Math.abs(x - currentX) > MAX_DISTANCE) {
22          if (x > currentX) {
23             proceed( targFE, MAX_DISTANCE );
24          } else {
25             proceed( targFE, -MAX_DISTANCE );
26          }
27       } else {
28          proceed( targFE, x );
29       }
30    }
31    /* Constrains distance of any y-axis movement to MAX_DISTANCE */
32    void around(mao.FigureElement targFE, float y) :
33       execution(* mao.FigureElement.setY(float)) && target(targFE) && args(y)
34    {
35       float currentY = targFE.getY();
36       if (Math.abs(y - currentY) > MAX_DISTANCE) {
37          if (y > currentY) {
38             proceed( targFE, MAX_DISTANCE );
39          } else {
40             proceed( targFE, -MAX_DISTANCE );
41          }
42       } else {
43          proceed( targFE, y );
44       }
45    }
46 }
```

Figure 2.2 MoveLimiting Example (in AspectJ)

pointcuts is woven into the implementation code. Unfortunately, clients of such an implementation module have no (modular) way to know that such advice will be applied to their calls to the implementation module. In my proposal clients of such an implementation module would know about the advice; this is an example of how explicitly accepted assistance allows modular reasoning.

In general a module may accept assistance from multiple assistants and both a client and an implementation module may accept assistance. The composition of assistant and implementation code is formed respecting the following symmetric order at each join point:

— Client Assistance

1. Apply any *before advice* accepted by the client module in the order that it is accepted.[3]
2. Apply the "before part" (i.e., the code preceding a proceed expression) of any around advice accepted by the client module in the order that it is accepted.

— Implementation Assistance

3. Apply any before advice accepted by the implementation module in the order that it is accepted.
4. Apply the before part of any around advice accepted by the implementation module in the order that it is accepted.

— Implementation

5. Execute the implementation module code.

— Implementation Assistance

6. Apply the "after part" (i.e., the code following a proceed expression) of any around advice accepted by the implementation module in the reverse order from which it is accepted.
7. Apply any *after advice* accepted by the implementation in the reverse order from which it is accepted.[4]

— Client Assistance

8. Apply the after part of any around advice accepted by the client module in the reverse order from which it is accepted.
9. Apply any after advice accepted by the client module in the reverse order from which it is accepted.

This ordering ensures that the first assistance accepted by the client is "nearest" to the client and that the last assistance accepted by the implementation is nearest to the implementation on any control flow path. Multiple applicable advice bodies in a single assistant are accepted in the order given in the assistant's declaration, or in the reverse order for after advice and the after part of around advice. Inherited advice is considered to appear at the end of the inheriting aspects; this respects the ordering for inherited advice defined

[3] In AspectJ, before advice is a variety of advice that executes prior to the execution of the advised code. Before advice does not use proceed and is primarily evaluated for its side effects, though it may throw exceptions.

[4] Like before advice, after advice in AspectJ is evaluated primarily for its side effects. After advice executes following the completion of the code it advises. AspectJ includes three sorts of after advice: after returning, after throwing, and general after advice. The first two execute when the advised code completes normally or completes abruptly, respectively [64, §14.1]. The third sort of after advice always executes.

for AspectJ [83, §3.5]. The ordering of advice is underspecified in AspectJ. My symmetric, total ordering differs from the asymmetric ordering of advice implemented in the current version of AspectJ [84, p. 182]. I believe that the symmetric ordering is more intuitive. But it is the total ordering that is most important. Because assistants may modify behavior, a total ordering helps in reasoning about the composition of these modifications. I discuss this more in Section 2.4.

For simplicity and modularity I propose to confine acceptance of assistance to the module in which it is explicitly accepted. For example, suppose there was a Rectangle subclass of FigureElement that overrode the move method. Assistance, like MoveLimiting, accepted by FigureElement would not automatically be applied to executions of Rectangle's move method. On the other hand, if Rectangle did not override FigureElement's implementation of setX, then the inherited method would carry with it the assistance accepted by FigureElement. This approach also provides flexibility since the programmer can always add an accept clause to the subclass module or override a superclass method; this gains assistance in the first case and "shadows" assistance acceptance in the second. Also for simplicity I propose not allowing interfaces to accept assistance. Experience with a working implementation may prompt reevaluation of these ideas.

Finally, I propose that nested aspects—aspects declared inside another module—be considered implicitly accepted by the containing module. This enables some useful idioms. For example, suppose a client module is a subclass of some generic class. That generic class might be designed to interface with modules that conform to a certain implementation interface. Now suppose some module exists that does not so conform. The subclass might use a nested aspect to modularize adapter code between the expected implementation interface and that actually provided.

2.2.1.2 Concern Maps

Modular reasoning in aspect-oriented programming languages can be achieved if modules explicitly accept assistance. But some assistants are applicable to code throughout an entire package or program, for example, a common exception handler. It would be inconvenient (to say the least) to include accept clauses for these assistants in every module, and error prone to have to remember to add accept clauses for these assistants to every new module.[5]

I propose concern maps to avoid these problems. A *concern map* is a source code construct that specifies a mapping from modules in a package, or set of packages, to the assistance that is accepted by those modules. In my initial design, each package may contain at most one concern map. In file-system-based implementations, the concern map for a package would be given in a file named package.map stored in the directory containing the package source code. The syntax for concern maps is given in Figure 2.3 on the following page. Figure 2.4 gives an example concern map for the package named mao.

The type pattern "*", in line 3 of Figure 2.4, says that all types in the mao package accept the MoveLimiting assistant. (I do not allow concern maps to specify fully qualified names in type patterns; instead I implicitly concatenate the name of the map's package with the given pattern. Thus the pattern "*" in the example, signifies the pattern mao.* in the global namespace. I do this because the map should only be able to specify acceptance of assistance for local types and types in subpackages.) The Rectangle pattern in line 7 of the example says that, in addition to the MoveLimiting assistant, the mao.Rectangle class also accepts the AreaStretching assistant.[6] As with accept clauses in modules, the identifier in an accept clause of a concern map is subject to Java's usual namespace rules for packages and imports.

[5] Such accept clauses would also represent code tangling.

[6] Both Rectangle and AreaStretching are elided in the current work.

$$
\begin{aligned}
\mathit{AspectMap} &::= \mathit{PackageDecl}\ \mathit{ImportDecls}_{opt}\ \mathit{MappingList}_{opt} \\
\mathit{PackageDecl} &::= \text{package}\ \mathit{Identifier};\\
\mathit{MappingList} &::= \mathit{Mapping}\ \mathit{MappingList}_{opt} \\
\mathit{Mapping} &::= \mathit{TypePat}\ \{\ \mathit{AcceptList}_{opt}\ \} \\
\mathit{AcceptList} &::= \mathit{AcceptClause}\ \mathit{AcceptList}_{opt} \\
\mathit{AcceptClause} &::= \text{accept}\ \mathit{Identifier};
\end{aligned}
$$

where *TypePat* refers to type patterns in the AspectJ Programming Guide [14, Appendix A], and *ImportDecls* refers to regular Java import declarations [64, §7.5].

Figure 2.3 Syntax of Concern Maps

```
1  package mao;
2
3  * {
4      accept MoveLimiting;
5  }
6
7  Rectangle {
8      accept AreaStretching;
9  }
```

Figure 2.4 Example Concern Map

One can think of concern maps as like an AspectJ "introduction"; they add accepts clauses to modules in the local package and subpackages. It would defeat the purpose of accepts clauses to allow their global introduction. So unlike AspectJ introductions, concern maps are lexically scoped.

The assistance accepted via concern maps still allows modular reasoning. To wit, the package clause at the beginning of a module names all the possible locations where a concern map naming that module might appear. The programmer, or a tool, must only look in that package, or possibly any outer packages, to find the applicable concern map. More specifically, the assistance accepted by a given module consists of:

1. all assistants named in accept clauses in the module,
2. all assistants to which the module is mapped by the package.map file for the module's package, and
3. all assistants to which the module is mapped by any package.map files in *outer packages* (i.e., packages surrounding the module's package).

To accommodate concern maps I extend the ordering of accepted assistance discussed in Section 2.2.1.1 by letting the search order described here define the ordering of acceptance.

This recursive search for acceptance of assistance in the module's package and outer packages allows the programmer to specify widely-applied assistance in the root of a package hierarchy, package-specific

assistance in the concern map of the package it applies to, and module-specific assistance in the modules it applies to.

(Strictly speaking, packages in Java and AspectJ are not hierarchical. They merely provide a hierarchical namespace. For example, code in an inner-package in Java does not have access to package-privileged code from any outer packages. My treatment of concern maps reflects the namespace hierarchy of packages, while still respecting their non-hierarchical encapsulation properties.)

2.2.1.3 Prototype Implementation

I have developed a prototype implementation of concern maps and accepts clauses. The prototype is based on the Polyglot compiler framework [122].[7] The prototype translates modules written in AspectJ with concern maps and accepts clauses into intermediate code that is pure AspectJ. Then the AspectJ compiler is used to generate bytecode. This compilation is modular without having to rely on the global dependency tracking that the AspectJ Development Toolkit (AJDT) inside ECLIPSE uses for its "modular" compilation.[8] (Because of the global nature of advice application in AspectJ without my extensions, the AJDT maintains a global, two-way mapping between advice and join points. When any code is changed in a program, this mapping is used to calculate the possibly affected code. Only that code is then re-compiled. This results in more efficient compilation, solving some of the practical problems of whole-program compilation. However, the need for the global mapping illustrates that whole-program reasoning is still required.)

Future work on concern maps will evaluate inheritance mechanisms that might allow finer grained control than the simple unioning of maps from outer packages presented above. Another simple, but useful extension to concern maps would be to allow multiple concern maps in a given package, each with a different name. So in addition to the default package.map, a programmer could include (for example) customer1.map and customer2.map, with each file activating some customer-specific code. These additional maps could be activated on a project-wide basis. So the programmer could choose to use all of the default concern maps, plus any customer1 concern maps that appear in the code base. In this case, modular reasoning would require knowledge of which set of concern maps was being used.

2.2.2 Spectators

Explicitly accepted assistance supports modular reasoning. Concern maps give the programmer flexibility in accepting assistance. But what about "development aspects"[82, p. 61], like tracing or debugging code, that are only sometimes included in an executing program? In a language that just supported explicitly accepted assistance, a programmer would need to edit concern maps or source code modules to control the application of development aspects.

To resolve this I propose that an aspect-oriented programming language should also support a category of aspects that I call *spectators*. A spectator is an aspect that does not change the behavior of any other module. Because it does not change the behavior, I will say that a spectator *views* (rather than "advises") methods.

In concrete terms, a spectator may only mutate the state that it owns (in the sense of alias control systems like [9, 10, 25, 35, 116, 117, 118, 121]) and it must not change the control flow to or from a viewed method. In addition to mutating owned state it seems reasonable to allow spectators to change accessible, global state as

[7] Were I developing the prototype today, I would instead use the AspectBench compiler [54], itself a Polyglot extension, to avoid the work of developing a full AspectJ front-end.

[8] The AspectJ Development Toolkit is available from http://www.eclipse.org/aspectj, URL valid as of July 17, 2005.

```
1  package mao;
2
3  spectator DistanceTracking {
4
5     /** Tracks total distance moved by all figure elements. */
6     private double distance;
7
8     before(float dx, float dy):
9        execution(* FigureElement.move(float, float)) && args(dx, dy)
10    {
11       distance += Math.sqrt( dx*dx + dy*dy );
12       System.err.println(''Total: '' + distance);
13    }
14 }
```

Figure 2.5 Example Spectator Aspect

well, since a Java module cannot rely on that state not changing during an invocation (modulo synchronization mechanisms).[9] The term "spectator" is intended to connote the hands-off role of these aspects.

For example, Figure 2.5 gives a spectator called DistanceTracking. The spectator declaration (in line 3) declares that this aspect does not change the behavior of any other module. This spectator mutates its own state by incrementing distance (in line 11) and mutates the global state by printing to System.err (in line 12). However, it does not change the behavior of FigureElement's move method. DistanceTracking merely views the arguments to the move method. The arguments are passed on to the method unchanged and the method's result is unchanged. In addition to cross-cutting concerns like this tracking example, spectators would also be useful for logging, tracing, and as the observer in the observer design pattern [62, pp. 293–303]. For example, one can imagine a traffic simulation program that uses spectators for visualization, thus separating the visualization and simulation concerns.

Because spectators do not change the behavior of the methods they view, code outside an existing program can apply a spectator to any join point in the original program without loss of modular reasoning. In reasoning about the client and implementation code for a method, a maintainer of the original program does not need any information from the spectator.

The primary challenge of implementing this part of my proposal lies in determining whether a given aspect is really a spectator. I envision a static analysis that conservatively verifies this. This analysis has two parts—verifying control flow and verifying that only appropriate locations are mutated.

In general the problem of verifying that a spectator does not disrupt control flow is undecidable (by reduction to the halting problem); however, one can restrict the sort of control flow allowed in spectators to achieve an approximate solution. I propose that in spectators:

— before advice must not throw a checked exception and must not explicitly throw an unchecked exception on any control flow path,

[9] It may be that experience with a practical implementation of this proposal would indicate that spectators not be allowed to mutate global state. My formalism, introduced in subsequent chapters, does not include global state. So the current work does not address this issue in detail.

— around advice must proceed, exactly once, to the advised method on all control flow paths, and

— after advice must not throw a checked exception and must not explicitly throw an unchecked exception on any control flow path.

This solution is approximate because it still allows advice in spectators to include (possibly infinite) looping constructs and to call other (possibly non-terminating) methods, provided any checked exceptions declared by those methods are caught and handled. These conditions correspond to partial correctness (ignoring termination) and ignore Java Errors, which I treat as outside the scope of specification.[10]

A more conservative solution to control flow might disallow loops, method calls, and synchronized code within a spectator's advice. A completely conservative solution is not possible in a Java-like language since executing any advice in a spectator requires more memory than just executing the viewed method. This additional memory usage could result in an OutOfMemoryError that prevents control flow from continuing to the advised method. Because of this, and the draconian nature of the more conservative solution, my approximate solution that disallows all explicitly thrown exceptions in the advice and handles any checked exceptions in methods called by the advice seems reasonable.[11] (I contend that this solution is also "Java-like" in only requiring the programmer to deal with checked exceptions.)

In addition to these control flow checks, the checks for "spectatorhood" must also verify that the proceed expression in around advice passes all arguments to the advised method in their original positions and without mutation. Any value returned from the advised method (or exception thrown) must be passed on by the advice without mutation.

The requirement that around advice in spectators proceeds exactly once, and with the same arguments, can be solved syntactically. The language can just separate the advice into before and after parts separated by an implicit proceed using the original arguments. The problem of returning the value of the advised code can also be solved through language design, by designing the semantics of advice in spectators to do that.

The mutation analysis for spectators is more challenging. It is closely related to the problem of verifying frame axioms [22]. In fact one can think of spectators as having an implicit frame axiom that prevents modification of locations that are relevant to the receiver, the arguments, or the value returned or exception thrown by the viewed method. (Intuitively the relevant locations are those that, if changed, would change the abstract state of the object [116, 118].)

The main difficulty with statically verifying this lack of relevant mutations is how to deal with aliasing. For example, suppose a logging spectator uses an array to track the elements added to some Set object. Suppose Set uses an array for its representation. If the spectator's array and the Set's array are aliased, the program might add an element to this array twice—possibly violating Set's invariant and changing its behavior. In this dissertation I introduce, and prove sound, mechanisms for statically ensuring spectatorhood, even in the presence of aliasing.

2.3 Evaluation

This section evaluates the expressiveness of my proposal. My evaluation is limited to a review of existing programs. I first consider the programming guidelines suggested in the ATLAS case study [77]. Then I survey the example aspects from the *AspectJ Programming Guide* [14] and the books by Kiselev [84] and Laddad [86].

[10] This also corresponds to JML's treatment of Errors.

[11] I imagine that, in many cases, program verification techniques could be used to prove termination and that no unchecked exceptions are thrown.

2.3.1 ATLAS Case Study

In the ATLAS case study [77], the authors propose several guidelines to make working with aspects easier. These are proposed since they had discovered that (p. 346):

> [the] extra flexibility provided by aspects is not always an advantage. If too much functionality is introduced from an aspect it may be difficult for the next developer—or the same developer a few months later—to read through and understand the code base.

One of Kersten and Murphy's suggestions is to limit coupling between aspects and classes to promote reuse. Specifically, they suggest that one should avoid the case where an aspect explicitly references a class and that class explicitly references the aspect, since then the class and aspect are mutually dependent. Such mutual dependencies prevent independent reuse. Is this suggestion problematic for my proposal that modules explicitly accept assistance? No, because explicit acceptance does not necessarily imply mutual dependence between aspects and classes. Suppose an implementation module, M, accepts assistance from an assistant, A, and A is applicable to M. If A explicitly references M, then the modules are mutually dependent. However, if A only applies to M because of pattern matching and does not explicitly reference M, then the modules are not mutually dependent. Another option when A references M is to include A as a nested aspect of M (i.e., an aspect declared inside M), confining their dependence to a single file.

Client acceptance provides another way to avoid mutual dependence. Suppose a client module, C, accepts assistance from an assistant, A', and A' only changes the behavior of modules referenced by C, but does not change C's behavior. In this case A' and C are not mutually dependent. In sum, programmers can reduce mutual dependency by having clients accept assistance, by limiting explicit references to classes from assistants, and by using nested aspects.

Kersten and Murphy also suggest using aspects as factories by having them provide only after-returning advice on constructors. This after-returning advice mutates the state of every object instantiated to change its default behavior. Limiting the aspects in this way restricts the scope of object–aspect interaction. In my proposal a simple assistant can fill the role of such a factory aspect.

For aspects that do not act as factories, Kersten and Murphy propose three style rules that restrict the use of aspects (pp. 349–350):

> Rule #1: Exceptions introduced by a weave must be handled in the code comprising the weave. . . . Rule #2: Advise [sic] weaves must maintain the pre- and postconditions of a method. . . . Rule #3: Before advise [sic] weaves must not include a return statement.

These rules are very similar to my definition of spectators in that they prevent aspects from changing the behavior of the viewed method. However, I propose elevating these style rules to the level of statically checked restrictions.

2.3.2 Impact of Restrictions

To better understand how my proposed restrictions might limit the practical expressiveness of AspectJ, I review several examples from three separate sources.

2.3.2.1 AspectJ Programming Guide

I use the examples in the AspectJ Programming Guide to see if my restrictions prohibit any recommended idioms. The programming guide's examples can serve this purpose since they "not only show the features [of

AspectJ] being used, but also try to illustrate recommended practice" [14] (from the Preface). I separate the example aspects into categories based on how I would implement them with my restrictions. Table 2.1 on the following page lists the examples by category; I describe the categories here.[12]

SPECTATORS Many of the example aspects clearly meet my definition of spectator. To satisfy my restrictions these would only require the spectator syntax.

ASSISTANTS Aspects in the examples that could be implemented as assistants can be divided into two kinds. *Client utilities* are used by client modules to change the effective behavior of objects whose types are declared in other modules. The changes in effective behavior do not affect the representation of those objects. To satisfy my restrictions, client utilities' assistance would have to be explicitly accepted by the clients. In fact, some of the client utility assistants are declared as nested aspects. These are similar in spirit to explicitly excepted assistance and would be implicitly accepted under my proposal.

Other example aspects that could be implemented as assistants might be considered *implementation utilities*. These assistants encapsulate some unit of cross-cutting concern related to a single module, for example, enforcing a common precondition across the methods of a class. In my proposal each implementation utility would be accepted by the module that it advises, creating a mutual dependency. However, in all the examples this mutual dependency could be fixed by nesting the implementation utility inside the advised module. I would also require that the call join points in these aspects be changed to execution join points.

The Coordinator aspect of the coordination package is abstract. This abstract aspect modifies the behavior of the modules to which it refers, making it an assistant in my terminology. However, Coordinator only refers to abstract pointcuts. Thus, for the advice in Coordinator to be applicable to any module a concrete aspect extending Coordinator would have to be declared. This concrete aspect would be an assistant and would need to be accepted per my design. In fact, the two "synchronization" implementation utilities listed in Table 2.1 on the next page are concrete assistants extending Coordinator.

COMBINED To satisfy my restrictions, one example aspect, the Debug aspect of the spacewar example, would require a combination of spectators and assistants. This aspect would be a spectator, except that it provides after advice to a GUI frame's constructor, to add debugging options to the frame's menu bar. To support this pattern with my restrictions, the GUI frame would have to accept assistance from an assistant, say AdditionalMenuConcern. This assistant would provide methods allowing other code to add to the GUI frame's menu bar. The debugging aspect would become a spectator viewing the program and using the methods provided by AdditionalMenuConcern to display the debugging menus.

To summarize, even with my proposal's restrictions it is easy to express AspectJ's recommended idioms.

2.3.2.2 Kiselev

While the AspectJ Programming Guide provides many small examples demonstrating recommended idioms, Kiselev's book *Aspect-Oriented Programming with AspectJ* [84] provides an extensive case study. The aspects given in the book in chapters 5–8 are all related to this case study, which concerns a web service that is supposed to store and retrieve users' stories. Table 2.2 on page 25 gives a summary of these aspects and how they relate to my categories.

[12]The examples listed in the table are from the examples directory of the Version 1.0.6 release of AspectJ, available from http://www.eclipse.org/aspectj.

Table 2.1 Categorization of Examples from the AspectJ Programming Guide

Example	*Category*
telecom/TimerLog	spectator
tjp/GetInfo	spectator
tracing/lib/AbstractTrace	spectator
tracing/lib/TraceMyClasses	spectator
tracing/version1/TraceMyClasses	spectator
tracing/version2/Trace	spectator
tracing/version2/TraceMyClasses	spectator
tracing/version3/Trace	spectator
tracing/version3/TraceMyClasses	spectator
bean/BoundPoint	client utility
introduction/CloneablePoint	client utility
introduction/ComparablePoint	client utility
introduction/HashablePoint	client utility
observer/SubjectObserverProtocol	client utility
observer/SubjectObserverProtocolImpl	client utility
spacewar/Display.DisplayAspect	client utility
spacewar/Display1.SpaceObjectPainting	client utility
spacewar/Display2.SpaceObjectPainting	client utility
telecom/Billing	client utility
telecom/Timing	client utility
spacewar/EnsureShipIsAlive	impl. utility
spacewar/GameSynchronization	impl. utility[a]
spacewar/RegistrySynchronization	impl. utility[a]
spacewar/Registry.RegistrationProtection	impl. utility
coordination/Coordinator	assistant[b]
spacewar/Debug	combined

[a]Extends the abstract coordination/Coordinator assistant.

[b]Refers only to abstract pointcuts.

Table 2.2 Categorization of Examples from Kiselev's Text

Example	*Category*
Development Aspects[a]	
Logger	spectator
Tracer	spectator
Profiler	spectator[b]
CodeSegregation	not defined[c]
Production Aspects	
Authentication	client utility[d]
Exceptions	client utility
NullChecker	spectator[b]
Runtime Aspects	
OutputStreamBuffering	impl. utility
Pooling	impl. utility
ConnectionChecking	impl. utility
ReadCache	impl. utility
not categorized	
NewLogging	client utility
PaidStories	spectator

[a] Subheadings give Kiselev's categorization.
[b] Requires minor changes to be a spectator.
[c] Introduces warnings and errors.
[d] Includes parent declarations.

Kiselev categorizes his examples as "development", "production", or "runtime" aspects [84, Chapter 9]. It is useful to discuss how these categories relate to my division of aspects into spectators and assistants.

The development aspects (Logger, Tracer, Profiler and CodeSegregation) are "used as development aids" (p. 115), but are not useful during production use of the system. Since these are optional aspects one would hope that they are spectators in my categorization. This is clearly the case for Logger and Tracer.

The Profiler aspect would be considered a spectator in my categorization—except for one issue. Profiler declares before and after advice that can change the control flow by explicitly throwing an exception when an I/O error occurs while writing profiling data to disk. I could categorize Profiler as an assistant and use a root-level concern map to apply it to the entire project. However, Profiler is a development aspect; one would like to be able to switch it off and on without editing the source code. I can resolve this difficulty by using a non-default concern map, which could be included or excluded from the configuration. Alternatively, one could simply change Profiler to swallow any I/O exceptions and report the problem to the developer (via System.err, for example) or to convert I/O exceptions into unchecked Errors. This would make Profiler a spectator, as intended.

CodeSegregation uses AspectJ's declare error and declare warning constructs to introduce additional compile-time checks. These constructs are outside the scope of the current work, but are discussed in Section 2.5.

The production aspects (Authentication, Exceptions, and NullChecker) are "absolutely essential to the application" [84, p. 115]. Since they are absolutely essential it is reasonable to include them in the appropriate concern maps knowing that these references will not have to be changed. These aspects do have some interesting properties *vis-à-vis* my categorization. The Authentication aspect is, at its core, a client utility.[13] It is applied to objects that render web pages and manage the current user session to ensure that the user of the session is validated. I say that this is a client utility because it may be the case that some client code of these objects should allow unauthenticated access to some pages. In Kiselev's example the Authentication aspect is applied broadly. This is easily implemented using a root-level aspect map. The Authentication aspect also uses introduction, via AspectJ's declare parents construct, to add additional methods to the classes it assists. Introductions are also beyond the scope of the current work, but are discussed in Section 2.5.

The Exceptions aspect is a client utility in my categorization, a fact emphasized by Kiselev when he argues for using a call join point instead of an execution join point by saying that, with the execution join point, "some other application would not be able to utilize this class without the Exceptions aspect attached to it" (p. 76).

I would argue that Kiselev's third production aspect, NullChecker, should actually be considered a development aspect, since it is an aid to contract enforcement that might not be included in a production system. In Kiselev's code NullChecker throws an exception and so would need to be modified in a similar way as Profiler, for example to throw an Error, to be a spectator in my categorization.[14]

Kiselev's runtime aspects (ConnectionChecking, Pooling, OutputStreamBuffering, and ReadCache) are all "useful but not critical" [84, p. 115]. These are units of cross-cutting concern that might not be part of an initial implementation. But once added to the system they are likely to remain part of it. They are all implementation utilities that apply to a single class. Under my proposal, as these aspects were written and added to the application, an accept clause would be added to the advised class. To use this approach the call join points in each of these aspects would be changed to execution join points.

The last two aspects in the table (NewLogging and PaidStories) are not categorized by Kiselev, but should be considered production aspects. The are used to change the behavior of the application without invasive editing. Under my proposal these aspects could be accepted using a root-level concern map. (PaidStories is an example of code that qualifies as a spectator but would likely be named in a concern map so as to make its functionality visible to all maintainers of the program.)

To summarize, except for the CodeSegregation aspect and a portion of the Authentication aspect, both of which use AspectJ features that are outside the scope of the current work, all the aspects in Kiselev's case study can be easily implemented under my proposal .

2.3.2.3 Laddad

I have also reviewed the examples in Laddad's text [86]. Except for ones that use features outside the scope of the current work—introductions, declare error/warning, declare soft, precedence declarations, and the "worker object creation pattern", discussed below—the previous categorizations apply. Having discussed most of the interesting issues in the preceding, I will not belabor them here. However, a few unique points bear mentioning.

[13] Despite discussing a web application, I am still using "client" here in the sense of some code which uses another module, not in the sense of a web client or browser.

[14] The Runtime Assertion Checker in the JML tools suite uses this Error technique for signaling specification violations in a way that is unlikely to be caught and suppressed by the code being checked [32, 33].

Laddad's ThreadPoolingAspect (Listing 7.13 on p. 228) uses both call and execution pointcuts in a single aspect. Nevertheless, this aspect is a client utility. The aspect allows clients of Java's thread API to use a thread pool, instead of using a new Thread object for each thread needed (and relying on the garbage collector to dispose of old Thread objects). The aspect uses call pointcuts to intercept calls to the Thread API. In particular, the aspect intercepts calls to Thread's constructor and returns a custom DelegatingThread object taken from the thread pool (or newly instantiated if the pool is empty). The DelegatingThread class is part of the thread pool implementation. The aspect uses execution advice on this class to detect when the thread has completed execution and can be placed back in the thread pool for reuse. In effect, the client utility ThreadPoolingAspect includes a small implementation utility within it to manage it's implementation of pool-able threads. Thus, the DelegatingThread class in the thread pool implementation would have to accept the ThreadPoolingAspect.

Laddad's worker object creation pattern (§8.1) uses *proceed closures*. In this pattern, advice captures a proceed expression inside an instance of an anonymous Runnable class. He gives the template for such capture as:

```
1  void around() : <pointcut> {
2      Runnable worker = new Runnable() {
3          public void run() {
4              proceed();
5          }
6      }
7      ...
8  }
```

When the worker object is run, the advised code will resume, but in a new thread! The elided code in line 7 might place worker in a queue, or might call its run method immediately and let Java manage the multi-threading. Such use of proceed is fascinating, but is outside the scope of my work which focuses on sequential aspect-oriented programs.

Laddad's participant pattern (§8.4) is another variation on hyper-cutting discussed in Section 2.2.1.1. Rather than using marker interfaces to allow a class to accept assistance from an aspect, in the participant pattern a class declares a nested aspect that extends some pre-defined abstract aspect. This nested aspect declares a concrete pointcut, overriding one from the abstract aspect. The concrete pointcut allows the class to specify which of its methods should be advised by advice in the abstract aspect. Thus the class "participates" in the decision about where advice should apply. Such an abstract aspect is clearly an example of an implementation utility and would be an (implicitly accepted) assistant in my categorization.

2.3.3 Summary of Evaluation

My proposed language features add restrictions to AspectJ. But my evaluation shows that these restrictions do not restrict the expressiveness of the language. In fact, most of the examples studied fall neatly into three categories: spectators, implementation utilities, and client utilities. The latter two are assistants with natural locations for explicit acceptance. This supports my contention (in Section 1.3.4) that experienced aspect-oriented programmers are already following disciplines, like the MAO discipline, that enable modular reasoning.

2.4 Specification and Reasoning

When a client invokes a method for which either the client or implementation module has accepted assistance, the behavior of that invocation is based on the sequential composition of the code along a particular control flow path. Similarly, one can reason abstractly about the possible behavior of the invocation by considering specifications for the method and the advice that might be triggered. To do so, one needs specifications for advice and some means for sequentially composing them.

In this section, I sketch some specification constructs that might be added to JML to allow specification of programs written in AspectJ with my proposed language extensions. I refer to the extended JML as AspectJML. I describe how the specifications from aspects and the base program may be composed to yield an effective specification for any code.

The discussion in this section ignores spectator advice. The static checks for spectatorhood, sketched above and formalized in Chapter 5, ensure the validity of this approach.

In order to focus on the most interesting issues, I just consider around advice with pre- and postconditions and frame axioms. I do not treat exceptions, or before and after advice.[15] I also concentrate on advice on method call and execution. The handling of advice for other join points in AspectJ would be similar, with the specification of the method implementation in my formalism replaced with the semantics of the advised operations. For example, accessing a field named beans of an object coffee can be represented by a specification like:

```
requires true;
assignable \nothing;
ensures \result = coffee.beans;
```

2.4.1 Specifying Around Advice

Several interesting issues arise in the specification of around advice. The concept of specification cases, from the specification of methods in JML, provides some useful insight. Consider Figure 2.6 on the next page, showing a method with its JML specification. This specification includes two specification cases, separated by the also on line 6. The behavior of the method is the conjunction of the behavior defined by these two cases. Whenever the precondition of a specification case, given by the requires clause, is satisfied, then the frame axiom (assignable clause) of that case must be satisfied by the method, and the postcondition (ensures clause) of that case must hold when the method terminates [90, 158].

For example, suppose the method in Figure 2.6 were called with its first argument not equal to null, such that invoking isBottom() on the argument yielded false. In that case only the ignoredRight field could be mutated and the result of the method, denoted by the \result keyword, would have to be the first argument. But what if the method were called with null as the first argument? In that case, neither specification case would apply. The behavior of the method is undefined (by the specification) in that case. It is also possible that the requires clauses for multiple specification cases may be satisfied. In that case the conditions of all the matching cases must be satisfied. Thus, we could rewrite the above specification, separating out the requirement on left != null, as shown in Figure 2.7 on the facing page. This conjoining of specification cases with also is a form of parallel composition. I will take advantage of this to describe specification composition for aspects. Raghavan and Leavens [140] give the technical details on composing specification cases in JML.

[15] The specification constructs discussed in this section, and their composition, is based on joint work with Gary Leavens [39]. In that work we formally treat before and after advice, including exceptions; we do not

```
1  /*@
2    @ public behavior
3    @     requires left != null && left.isBottom();
4    @     assignable \nothing;
5    @     ensures \result == right;
6    @ also
7    @ public behavior
8    @     requires left != null && !left.isBottom();
9    @     assignable ignoredRight;
10   @     ensures \result == left; @*/
11 BindingTerm union(BindingTerm left, BindingTerm right) {
12     if (left.isBottom()) {
13         return right;
14     } else {
15         ignoredRight = true;
16         return left;
17     }
18 }
```

Figure 2.6 Example of JML Specification Cases

```
/*@
  @ public behavior
  @     requires left != null;
  @     assignable ignoredRight;
  @     ensures true;
  @ also
  @ public behavior
  @     requires left.isBottom();
  @     assignable \nothing;
  @     ensures \result == right;
  @ also
  @ public behavior
  @     requires !left.isBottom();
  @     assignable ignoredRight;
  @     ensures \result == left; @*/
```

Figure 2.7 Example JML Specification Showing Overlapping Specification Cases

The key problems in specifying around advice arise because of proceed expressions. In AspectJ, around advice might not proceed to the advised code, or it might proceed more than once. So in addition to specifying pre- and postconditions and frame axioms, specifications for around advice must be able to express the pattern of proceeds used in the advice. For modular verification of advice, the specification must also give the conditions that hold when proceeding and the conditions expected when the proceeded-to code terminates. Finally, around advice can use the result of the proceeded-to code. Similar to the \result keyword for specifying the return value of a method, specifications for around advice need some way to discuss the result of the proceeded-to code.

To solve these problems, I propose a new specification clause for AspectJML. This *proceed clause* has the same syntax as proceed expressions in AspectJ, but uses JML specification expressions for its arguments. (JML specification expressions are side-effect free, and include keywords like \old for referring to the pre-state value of a reference.) A proceed clause in a specification case converts that case into a compound specification case, each part of which has its own pre- and postcondition and frame axiom. In the case following a proceed clause, the specification may use the keyword \reply to refer to the result of the proceeded-to code. Figure 2.8 on the next page gives an AspectJML specification of one of the pieces of around advice from the MoveLimiting aspect of Figure 2.2.

The figure consists of two specification cases, one starting on line 2 and one starting on line 12. Each of these specification cases is a compound case, split by a proceed clause (see lines 6 and 16). Consider the first specification case. The precondition says that this case is applicable when the distance to be moved is greater than the maximum (line 3). The frame axiom says that no state may be mutated before proceeding to the advised code, and the postcondition of the case before proceeding (line 5) says that the advice makes no additional promises to the advised code, beyond those implied by the precondition and frame axiom. The proceed clause beginning in line 6 asserts that if the precondition of the specification case holds, then the advice will proceed to the advised code, using the given arguments. Finally, the case beginning in line 8, says that no expectations are placed on the advised code and no state is mutated after the advised code completes, but line 10 says that the result (\result) of the advice is the result of the advised code (\reply).

Because of the simplicity of this advice, and my wish to convey the details of the specification language, this specification is actually more verbose than the method it specifies. This is not always the case. Typically a specification would provide a more abstract description of the behavior of the advice. Also, pre- and postconditions with the default value of true may be omitted.

Three more details of the proposed specification language design bear mentioning. (1) If a specification case for a piece of around advice does not include a proceed clause, then the advice will not proceed when the precondition of that specification case is satisfied. (2) The \old expression, for referring to pre-state values, always refers to the pre-state of the entire piece of advice, even when used after a proceed clause. (3) The \reply expression for referring to the result of proceeded-to code, always refers to the immediately preceding proceed clause. JML includes mechanisms that would allow a specifier to refer to results from prior proceed clauses. The code in Figure 2.9 on page 32 illustrates these ideas, and also demonstrates the specification of code that proceeds more than once.

The advice in the figure applies to calls to the nextInt(int) method of java.util.Random. The advice itself is rather silly, but serves to illustrate the ideas. The first specification case (lines 2 through 4) says that if the argument to the call is non-positive, then the advised method will not be executed and no state will be mutated. The result value is unspecified in this case.

---

provide a formal treatment for around advice.

```
1  /*@
2    @ public behavior
3    @     requires distance(argX,argY) > MAX_DISTANCE;
4    @     assignable \nothing;
5    @     ensures true;
6    @     proceed(argX * MAX_DISTANCE / distance(argX,argY),
7    @                   argY * MAX_DISTANCE / distance(argX,argY));
8    @     requires true;
9    @     assignable \nothing;
10   @     ensures \result == \reply;
11   @ also
12   @ public behavior
13   @     requires distance(argX,argY) <= MAX_DISTANCE;
14   @     assignable \nothing;
15   @     ensures true;
16   @     proceed(argX,argY);
17   @     requires true
18   @     assignable \nothing;
19   @     ensures \result == \reply;
20   @*/
21 FigureElement around(float argX, float argY) :
22     execution(* mao.FigureElement.move(float, float)) && args(argX, argY)
23 {
24     float moveDistance = distance(argX, argY);
25     if ( moveDistance > MAX_DISTANCE ) {
26         float ratio = MAX_DISTANCE / moveDistance;
27         return proceed( argX * ratio, argY * ratio );
28     } else {
29         return proceed( argX, argY );
30     }
31 }
```

Figure 2.8 Around Advice Specification in AspectJML

The other specification case, beginning in line 6 applies when the argument to the call is positive. Line 9 says that the advice proceeds to the method using an argument that is twice the original value. The implementation satisfies this specification because of the multiplication in line 24. Lines 10 and 11 demonstrate one way to capture values in a specification. The forall clause declares that the new variable firstResult may range over all possible int values. Then the requires clause restricts the variable to just that value that is the same as the result of the first execution of the advised code. Thus, when firstResult is used in line 15, it refers to the value of the first execution of the advised code, while \reply in that line refers to the second.

This technique of explicitly describing the control-flow structure of the advice relative to its use of proceed handles most advice. However, some advice is still outside the scope of the technique. For example, an aspect Insistent might declare advice on a method for opening a network connection so that the method is just repeatedly proceeded to until it succeeds. I leave to future work the specification of such advice that proceeds an indefinite number of times.

The specification of before and after advice is much simpler than around advice. Before advice implicitly

```
1  /*@
2    @ public behavior
3    @     requires max <= 0;
4    @     assignable \nothing;
5    @ also
6    @ public behavior
7    @     requires max > 0;
8    @     assignable \nothing; // no values that exist in pre-state are modified
9    @     proceed(targ, \old(max) * 2);
10   @     forall int firstResult;
11   @     requires firstResult == \reply;
12   @     assignable \nothing;
13   @     proceed(targ, \old(max) * 4);
14   @     assignable \nothing;
15   @     ensures \result > (firstResult + \reply);
16   @*/
17 int around( java.util.Random targ, int max ) :
18     call(int nextInt(int)) && target(targ) && args(max)
19 {
20     int maxStart = max;
21     if (max <= 0) {
22         return 0;
23     } else {
24         max = max * 2;
25         int resultA = proceed(targ, max);
26         max = max * 2;
27         int resultB = proceed(targ, max);
28         return maxStart + resultA + resultB;
29     }
30 }
```

Figure 2.9 Example Specification for Around Advice with Multiple proceed Expressions

proceeds to the advised code. So specifications for before advice look like regular method specifications in JML, but the postcondition of the specification gives the promises made to the advised code. After advice in AspectJ includes mechanisms for binding the result of the advised code to formal parameters, so \reply is not needed there. The precondition of a specification for after advice gives the conditions expected of the advised code, while the postcondition gives the conditions promised to the calling code.

2.4.2 Specification Composition

Having proposed extensions to JML for specifying around advice (and sketched their extension to before and after advice), I next describe how the effective specification of a piece of code may be determined from the specification of that code plus the specifications of any accepted assistants.

When reasoning about a call to an advised method from the client's perspective, one would like to use an effective specification that abstracts away the details of the control flow and intermediate state transformations. That is, the effective specification from the client's perspective should just concern the preconditions as control

flow leaves the client and the postconditions as control flow returns to the client, along with the relevant frame axioms. With explicitly accepted assistance, the effective specification may also need to refer to assistant instances, for example if the effective specification depends on assistant state.

I describe effective specifications in terms of paths. In a running program, the effective behavior is the sequential composition of the code executed along a control flow path. Similarly, the effective specification is formed by a kind of sequential composition of the specifications along an abstract path. When a set of paths are in parallel, then the effective specification of the set is a kind of parallel composition of the parallel paths' specifications.

I present my model in two stages. I first describe how to construct a specification composition graph, from the specifications of the implementation module and those of any assistants accepted by that module or the client module. I then describe how the graph is used to determine the effective specification of the invocation.

2.4.2.1 Constructing a Specification Composition Graph

The specification composition graph is an abstraction of the control flow graph for the corresponding code. The specification composition graph is based on the proceed clauses in the specification cases of applicable advice. The graph is used to determine the possible paths through the advice and method specifications (and hence the code if the implementation is correct). These paths are used to calculate the effective specification.

The nodes of a specification composition graph come in two flavors. Some nodes represent simple (i.e., not compound) specification cases. Other nodes represent parameter binding events. The edges of a specification composition graph also come in two flavors, representing abstract arguments and formal parameters. The nodes and edges alternate so that a specification-case node is connected to a parameter-binding node by an abstract-argument edge, and the parameter-binding node is connected to a subsequent specification-case node by a formal-parameter edge. The idea is that specification-case nodes along any path will give the conditions that must hold on that path, while the edges and parameter-binding nodes will allow the formal parameters used in any specification cases to be connected to each other, and ultimately to formal parameter names that make sense from the client's perspective. An example below will, hopefully, make this clear.

The parameter-binding nodes in a graph are denoted by start and end labels representing the passing of arguments and the return of results respectively. Each graph includes one start node of degree 1, representing the call site in client code, and one end node of degree 1, representing the return site in client code.[16]

For formalizing specification composition, I use a desugared form for simple specification cases [140]. I assume that methods referenced within the specification (which JML requires to be *pure*, i.e., side-effect free) are replaced with the value of \result from their postconditions, with appropriate substitution of actuals for formals [49]. Thus the requires clause

```
requires distance(argX,argY) > MAX_DISTANCE;
```

from line 3 of Figure 2.8, would be desugared to:

```
requires Math.sqrt(argX*argX + argY*argY) > MAX_DISTANCE;
```

assuming a reasonable specification for the distance method (and Euclidean space). I assume that the quantification and binding technique, used in lines 10 and 11 of Figure 2.9 on the preceding page, is used for recording any pre-state values used in the specification case. Thus, no \old expressions appear in the desugared cases.

[16] This formulation of the specification composition graph exchanges the meaning of nodes and edges from our original formulation [39]. I believe the present formulation is more satisfactory for dealing with around advice.

Because field references in specifications may use an implicit this target, and because this refers to different objects in the specification of a method versus that of advice, ambiguities might arise. So in a method specification, I make explicit all field references with an implicit target. And in a specification for advice, I replace all implicit or explicit this targets with a special variable \aspect_i. The special variables \aspect_1 through \aspect_q are used consistently to represent the assistant aspects, accepted by either the client or implementation module, that are in scope at the call site.

With this desugaring, all simple specification cases have the form:

forall Q;	where Q is a set of variable names with types
requires r;	where r is a predicate
assignable f;	where f is a set of variables
ensures e;	where e is a predicate

In the specification composition graph, the specification-case nodes correspond to such desugared specification cases. I also have to deal with the fact that the program state may change along a path because of allowed mutations. To handle this, I treat the predicates, r and e, as functions of the program state.

Let Σ represent the set of all possible program states. Then each specification-case node in the graph can be represented by a 5-tuple, $v = \langle Q', r, f, e, \sigma \rangle$, where Q' is Q along with \result and/or \reply if these keywords may appear in the specification case; the predicates r and e each have the type $\Sigma \rightarrow Bool$; and $\sigma \in \Sigma$ represents the pre-state of the specification case.

A recursive procedure builds the specification composition graph for a given method call. As an example, I will consider a call to FigureElement's move method, with FigureElement accepting the assistance of the MoveLimiting aspect. I will use the specification of the around advice from Figure 2.8 on page 31.

In general a module may accept assistance from multiple assistants and both a client and an implementation module may accept assistance. The specification composition graph is formed respecting the ordering given in Section 2.2.1.1. (The composition could be relaxed to allow any ordering of advice, but then the specifications would have to built from cases that non-deterministically combined all possible orderings. If the aspects did not interfere with each other, then this combination might actually be deterministic. For clarity, I simply assume a total, symmetric ordering here.)

GRAPH CONSTRUCTION ALGORITHM The graph construction begins with one node for each specification case from the specification of the called method, plus nodes start_0 and end_0. Add edges, labeled with the method's signature from start_0 to each of the specification-case nodes. Add edges, labeled "\result" from each of the specification-case nodes to end_0. Call the graph at this stage of construction G_0. Figure 2.10 shows G_0 for the example call, where the called method has but one specification case.

Next, a new graph is recursively generated for each piece of around advice, beginning with the one "nearest" to the actual method, according to the total, symmetric ordering of accepted advice. Number these pieces of advice from 1 to n in nearest-to-farthest order.

To construct graph G_i, start with an empty graph. Create nodes start_i and end_i. Consider each top-level specification case (i.e., each case separated by an also) in the specification of advice i. For each simple specification case within the top-level case (i.e., each case separated by a proceed clause), create nodes v_1 through v_m. Add an edge from start_i to v_1, labeled with the signature of the advice. For each node $v_j \in \{v_1, \ldots, v_{m-1}\}$, create a copy of graph G_{i-1}. Add an edge from v_j to start_{i-1} of the copy. Add an edge, labeled "\reply", from end_{i-1} of the copy to v_{j+1}. Finally, add an edge, labeled "\result" from v_m to end_i.

Figure 2.11 on page 36 shows a partially complete version of the graph G_1 for the running example, after the construction for just the first top-level specification case from Figure 2.8 on page 31 has been completed. Notice

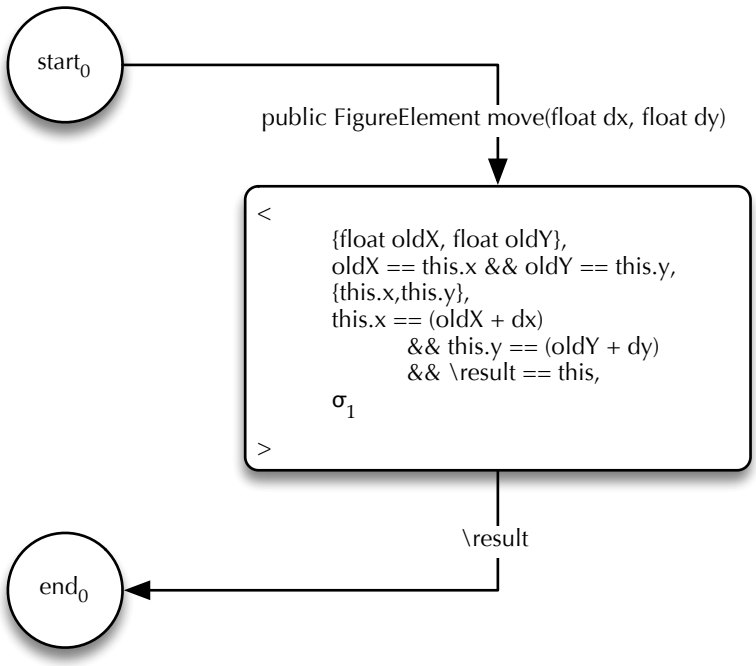

Figure 2.10 Specification Composition Graph Construction, Stage G_0

the single copy of G_0, representing the single proceed clause in the first top-level specification case. Figure 2.12 on page 37 shows the complete graph G_1 for the running example, adding the nodes and edges for the other top-level specification case.

The final specification composition graph for a method call is G_n. Since there is only one piece of advice in the running example, Figure 2.12 shows a complete specification composition graph.

2.4.2.2 Composing Specifications Along A Path

The specification composition graph, G_n, contains all the information needed to calculate the effective specification of a method invocation. I first describe how to compose specifications along any single path in G_n.

Consider a unique path from start to end in the graph. Because of top-level specification cases and around advice that does not proceed, this path may not visit every node in the graph. This path contains all the information necessary to (1) calculate the conditions that must hold if the corresponding path through the code is executed, and (2) map formal parameter names in one specification case to those in the adjoining ones.

α-CONVERTING SPECIFICATION CASES To prevent capture of locally bound variables when composing the specifications, I α-convert the specification cases and related advice signatures so that all bound variable names are unique. I reserve the method's formal parameter names for pre-state values of the effective specification. This allows the effective specification to use formal parameter names that make sense from the client's perspective. However, this reservation means that I must α-convert the signature and specification cases of the method also. Similarly, I reserve the \result keyword for the post-state of the effective specification, so all

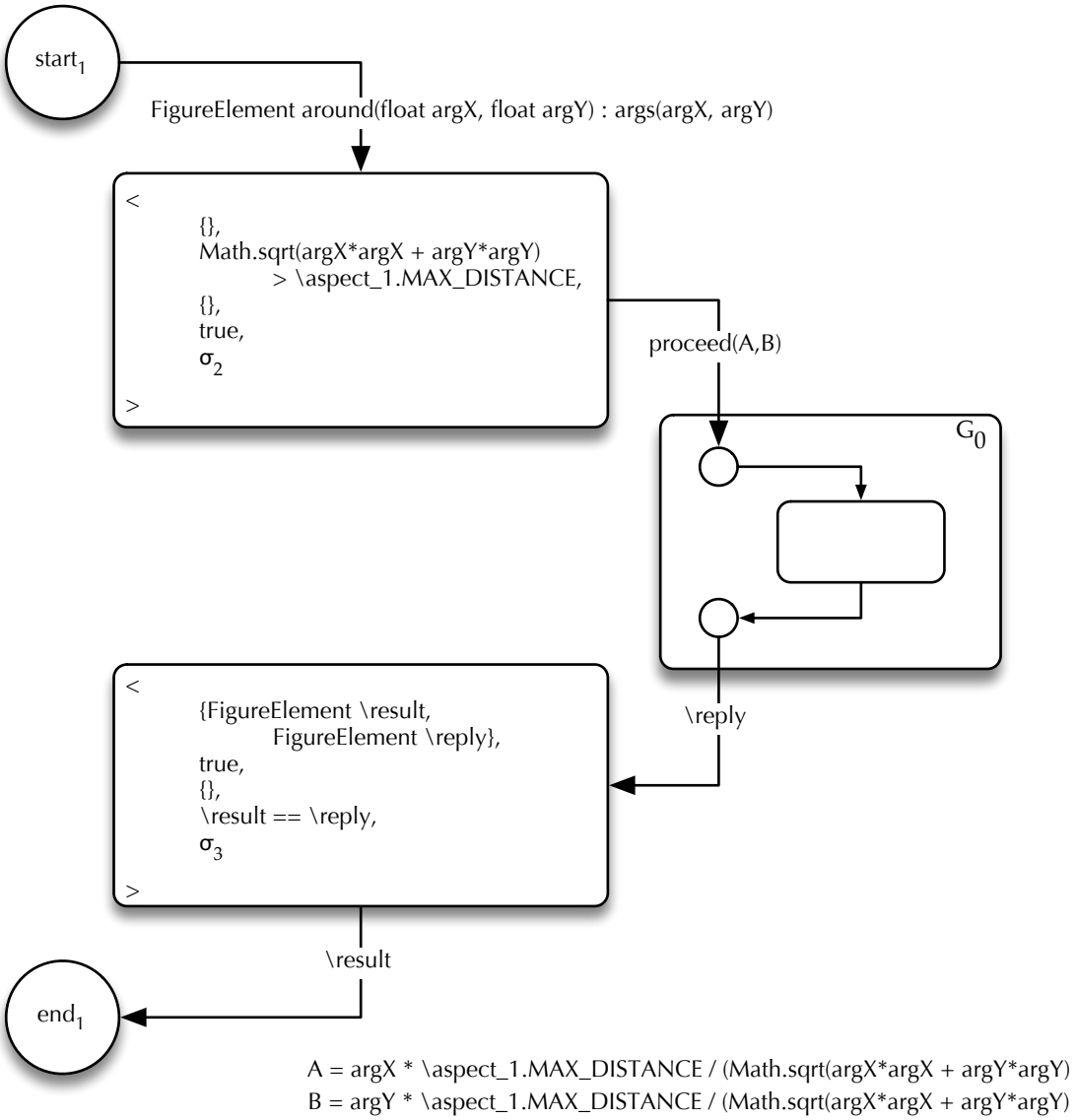

Figure 2.11 Specification Composition Graph Construction, Partially Complete Stage G_1

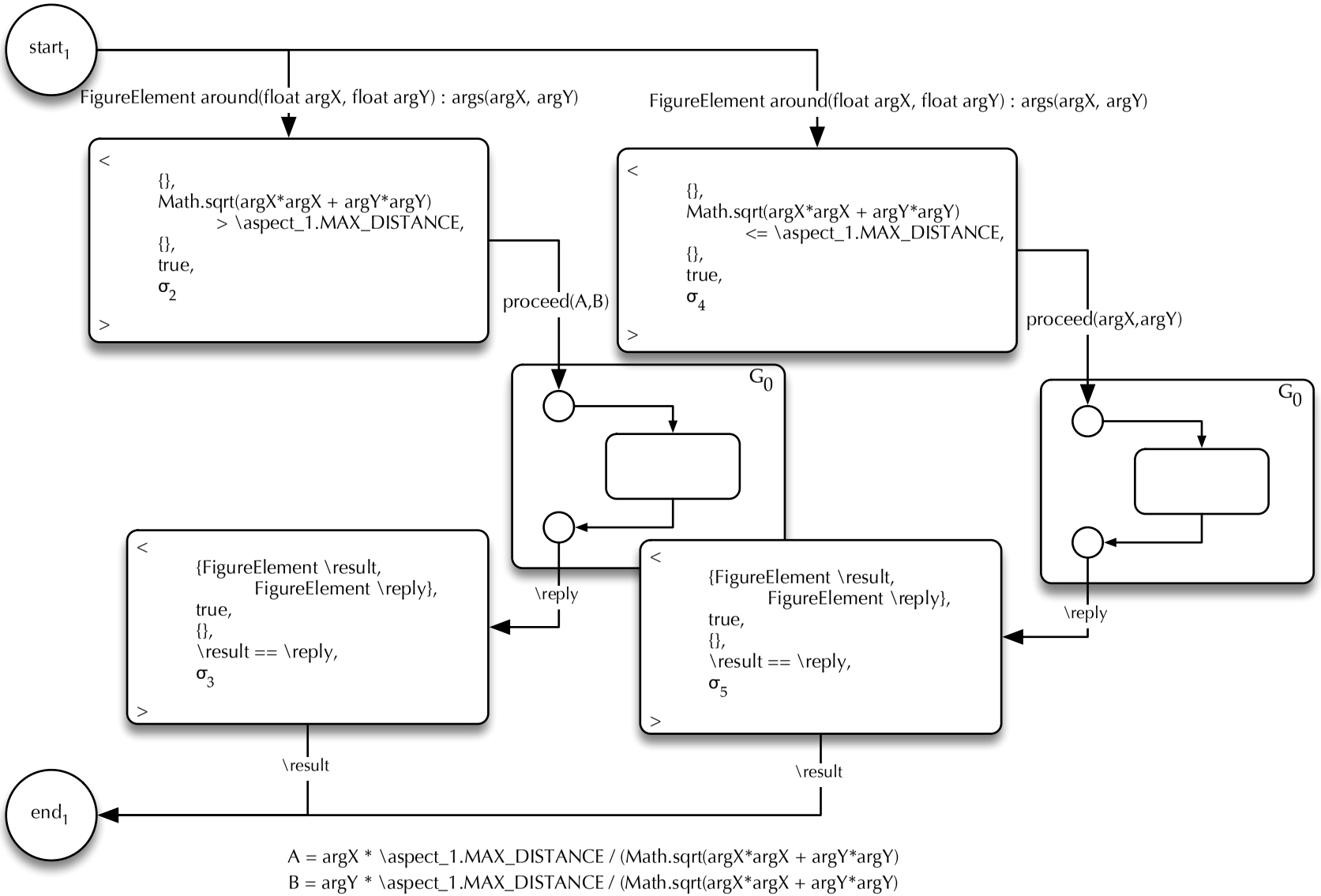

Figure 2.12 Specification Composition Graph Construction, Stage G_1

instances of \result in the graph must also be α-converted. Figure 2.13 on the facing page shows one such path through the specification control graph of Figure 2.12, after α-conversion. Compare this figure with Figure 2.10 and Figure 2.11 to see the results of α-conversion. For clarity, I renumber the states, σ_i, in order.

I represent the α-converted path from the specification composition graph as a sequence of alternating nodes and edges

$$\langle \mathsf{start}_1, P_1, v_1, A_1, \ldots, \mathsf{start}_m, P_m, v_m, A_m, \mathsf{end}_m, \ldots P_n, v_n, A_n, \mathsf{end}_n \rangle,$$

where n is number of specification-case nodes on the path, the P_i represent the labels on formal parameter edges, the $v_i = \langle Q_i', r_i, f_i, e_i, \sigma_i \rangle$ represent the specification cases, and the A_i represent the labels on actual argument edges. By the construction of the specification composition graph, each unique path will have a single "central" specification-case node, v_m, with one start neighbor and one end neighbor. This node represents either a specification case from the advised method, or else from the first advice on this path that does not proceed to the advised code.

CONNECTING BOUND VARIABLES If a given path is traversed in a program execution, then it must be the case that all the pre- and postconditions of specification cases along the path hold. This is the essential insight for constructing the effective specification. The abstract arguments and formal parameters given by edge labels in the graph provide context for connecting the bound variables from each of the specification-case nodes. This ensures that the effective pre- and postconditions are sensible from the client's context.

For the sketch presented here, I assume the existence of a function *bind* that encodes the semantics of parameter passing in AspectJ and Java. So that this parameter passing can be represented in the effective specification, I further assume that *bind* generates a JML predicate for any given parameter passing operation. Some examples follow:

Method call. For a direct method call, the effective specification should match that of the method. So for a parameter-binding node with in-edge

$$A_{i-1} = \text{public FigureElement move(float dx, float dy),}$$

and out-edge

$$P_i = \text{public FigureElement move(float dx2, float dy2),}$$

the predicate generated is $\mathit{bind}(A_{i-1}, P_i) = (\text{dx==dx2 \&\& dy==dy2})$. This causes the α-converted formal parameters of the method specification (dx2 and dy2) to match the formal parameters from the client context (dx and dy).[17]

Initial advice execution. For the first piece of advice on a path, the predicate is generated according to AspectJ's pointcut binding semantics. For example:

$$\mathit{bind}(\text{public FigureElement move(float dx, float dy),}$$
$$\text{FigureElement around(float argX1, float argY1) : args(argX1, argY1)})$$
$$= (\text{argX1==dx \&\& argY1==dy}).$$

[17] This case, where no advice is included, does not appear in the running example.

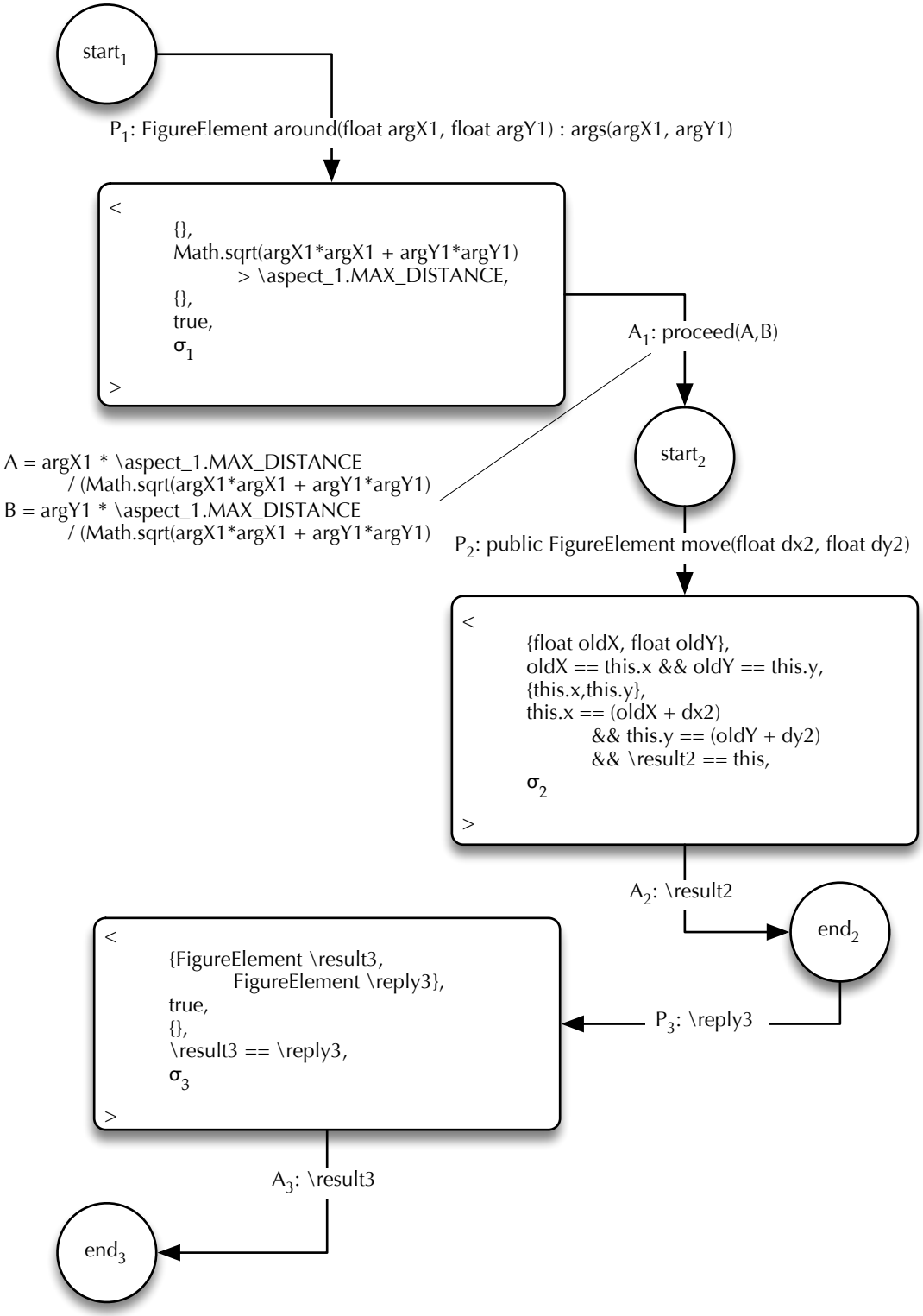

Figure 2.13 Unique Path through Specification Composition Graph, α-converted

This links the formal parameters in the α-converted advice specification (argX1 and argY1) to the formal parameters from the client code (dx and dy).

Proceed. For a parameter-binding node after a piece of advice proceeds, the predicate is generated according to AspectJ's proceed binding semantics. For example:

$$bind\big(\text{proceed(argX1,argY1)},\ \text{public FigureElement move (float dx2, float dy2)}\big) = \big(\text{argX1==dx2 \&\& argY1==dy2}\big).$$

This connects the formal parameters of the α-converted advice specification (argX1 and argY1) to those of the α-converted method specification (dx2 and dy2).

Return value. For an end parameter-binding node, connecting a specification case to subsequent around advice,

$$bind\big(\backslash\text{result2}, \backslash\text{reply3}\big) = \big(\backslash\text{result2}==\backslash\text{reply3}\big).$$

A full treatment of the *bind* predicate-generating function is beyond the scope of this work, though I formalize the parameter passing semantics of Java and a subset of AspectJ in Chapter 3. (For a full formalization of the *bind* function, I would have to include the signature of the relevant advice on each proceed(...)-labeled edge in the sketch presented here. This is because the semantics of proceed in AspectJ depends on the signature of the containing advice. Also, any information at a join point that is not bound by the advice passes unchanged to the called method. Thus a full formalization of *bind* would treat the data from the original join point as available throughout the graph.)

DEALING WITH INTERMEDIATE STATES I next describe the basic intuition for constructing the effective specification of the path. Recall that the path is

$$\langle \text{start}_1, P_1, v_1, A_1, \ldots, \text{start}_m, P_m, v_m, A_m, \text{end}_m, \ldots P_n, v_n, A_n, \text{end}_n \rangle.$$

The predicate generated in this initial construction is not the one that will ultimately be used. For example, it explicitly includes the entire program state at each stage along the path. I deal with that complication, and frame axioms, below.

1. Along a path, all the pre- and postconditions must hold, so conjoin the pre- and postconditions of each specification-case node, v_i, on the path:

$$(r_1(\sigma_1) \wedge \ldots \wedge r_n(\sigma_n)) \wedge (e_1(\sigma_2) \wedge \ldots \wedge e_n(\sigma_{n+1})),$$

   where the postconditions operate on the post-state, σ_{i+1}.

2. Add conjuncts for each parameter-binding node of degree 2 on the path. This connects the proceed arguments to formal parameters, and results to \reply variables:

$$(bind(A_1, P_2) \wedge \ldots \wedge bind(A_{n-1}, P_n)) \wedge (r_1(\sigma_1) \wedge \ldots \wedge r_n(\sigma_n)) \wedge (e_1(\sigma_2) \wedge \ldots \wedge e_n(\sigma_{n+1})).$$

3. Add conjuncts for the start and end nodes of degree 1. This connects the abstract arguments from the client's context to the starting formal parameters of any advice, and connects the ending result to the client's \result variable.:

$$\begin{aligned} &\big(bind(A_0, P_1) \wedge bind\big(A_n, \backslash\text{result}\big)\big) \\ &\wedge (bind(A_1, P_2) \wedge \ldots \wedge bind(A_{n-1}, P_n)) \\ &\wedge (r_1(\sigma_1) \wedge \ldots \wedge r_n(\sigma_n)) \wedge (e_1(\sigma_2) \wedge \ldots \wedge e_n(\sigma_{n+1})), \end{aligned} \tag{2.1}$$

   where A_0 is signature of the called method.

Formula (2.1) gives the basic result of the effective specification, ignoring frame axioms. But, to reason about the effective specification from the client's perspective, I must eliminate the intermediate states from this formula. One way to do this would be to quantify over all the states, like:

$$\forall \sigma_1, \ldots, \sigma_n \cdot X$$

where X stands for formula (2.1). However, in JML entire states are not directly expressible, so this idea has to be used indirectly by quantifying over intermediate values of each of the free variables used in the predicates. These intermediate values will also let me express the frame axioms.

For well-typed specifications, the only free variables are field references. The idea is this: for each field reference *var*, introduce $n+1$ intermediate, quantified variables, $var_1, \ldots, var_{n+1}$, one for each state.[18] So var_i represents the value of the field reference *var* in state σ_i. Then, based on the frame axioms, I include predicates that relate the value of *var* in one state to its value in the next. For example, if the frame axiom, f_3, for the specification case v_3, does not give permission to mutate *var*, then $var_3 == var_4$. (This assumes that *var* has a non-reference type. If *var* has a reference type, then the predicate would be var_3.equals(var_4).)

To formalize this notion, let $\overline{var}$ stand for all the free variables in the path and let $\overline{T}$ be their types. For the sample path in Figure 2.13 on page 39, $\overline{var} = \langle \text{this.x}, \text{this.y} \rangle$ and $\overline{T} = \langle \text{float}, \text{float} \rangle$. I will write $\overline{var}_i$ to represent all the intermediate variables in state σ_i. So for the example $\overline{var}_2 = \langle \text{x}_2, \text{y}_2 \rangle$. (In general, appropriate α-conversion may be needed to avoid collisions for like-named fields in different objects.) Now to remove intermediate state from the pre- and postconditions in formula (2.1), I substitute the intermediate variables for the free variables. I write $\{\overline{var}_i / \overline{var}\}$ for this capture-avoiding substitution. For example, the precondition, r_2, for a specification-case node, v_2, is written $r_2(\sigma_2)$ in formula (2.1). With the notion of intermediate state variables, this precondition becomes $r_2\{\overline{var}_2 / \overline{var}\}$. Because the intermediate state variables are quantified in the effective specification, the resultant precondition contains no free variables. Thus the global state, σ_2, can be dropped. A similar substitution applies for the postconditions, e_i.

ENCODING FRAME AXIOMS I have given enough machinery thus far to describe parameter passing and to connect pre- and post-states of every specification case. I still need to address frame axioms. I also must show how to connect the pre-state of the client to the first intermediate state (the pre-state of the first specification case), and similarly for the post-state. To this end, I define two more auxiliary functions, *equal* and *notmod*. Let $equal\big(var_i, var_j\big)$ be the predicate denoting equality for the intermediate variable *var* in states i and j, using the appropriate notion of equality, either == or .equals(). Let $equal\big(\overline{var}_i, \overline{var}_j\big)$ be the pointwise extension of

[18] The field reference *var* will have the form this.name or aspect_i.name, because of the explicating of field references in the desugaring described in Section 2.4.2.1. The intermediate variables represent the entire reference form.

```
1  forall T var_1; ...; forall T var_{n+1};
2  forall Q'_1; ...; forall Q'_n;
3  requires
4     equal(var, var_1)
5     && bind(A_0, P_1)
6     && bind(A_1, P_2) && ... && bind(A_{n-1}, P_n)
7     && r_1{|var_1/var|} && ... && r_n{|var_n/var|}
8     && e_1{|var_2/var|} && ... && e_{n-1}{|var_n/var|}
9     && notmod(f_1, var, 1, 2) && ... && notmod(f_{n-1}, var, n-1, n);
10 assignable f_1 ∪ ... ∪ f_n;
11 ensures
12    bind(A_0, P_1) && bind(A_n, \result)
13    && bind(A_1, P_2) && ... && bind(A_{n-1}, P_n)
14    && e_1{|var_2/var|} && ... && e_n{|var_{n+1}/var|}
15    && notmod(f_1, var, 1, 2) && ... && notmod(f_n, var, n, n+1)
16    && equal(var_{n+1}, var);
```

Figure 2.14 General Form of the Effective Specification for a Single Path in the Specification Composition Graph

this function. For the running example,

$$equal\left(\overline{var}_1, \overline{var}_2\right) = \left(\mathrm{x}_1{==}\mathrm{x}_2 \;\&\&\; \mathrm{y}_1{==}\mathrm{y}_2\right),$$

indicating that no intermediate state variable changes between states 1 and 2.

Let $notmod\left(f, \overline{var}, i, j\right)$ stand for the predicate that says that all variables in $\overline{var}$ but not in the frame, f, are unchanged between states i and j. That is,

$$notmod\left(f, \overline{var}, i, j\right) = \mathop{\&\&}_{var \in (\overline{var} \setminus f)} equal(var_i, var_j).$$

FORMING THE EFFECTIVE SPECIFICATION This machinery is now sufficient to state the effective specification of a single path through the specification composition graph. Figure 2.14 gives the general form of this single-path specification. A line-by-line description follows:

Line 1 declares all of the quantified, intermediate-state variables used in the pre- and postconditions, as discussed above.

Line 2 declares the quantified variables declared by all of the specification cases on the path.

Lines 3–9 give the preconditions for the effective specification. In general, the preconditions of the effective specification include all of the conjuncts from formula (2.1) on the previous page, except for the final postcondition and the binding of the final result. The logic is this: to reach the final specification case on this path, every precondition on the path must have been satisfied and, by the assumed correctness of the implementation, every postcondition up to the last must also have been satisfied.

Line 4 constrains the first intermediate state, represented by $\overline{var}_1$, to be the initial state of the effective specification.

Line 5 connects the abstract arguments from the client's perspective to the α-converted parameters of the first specification-case node.

Line 6 models the passing of parameters and results all the way through the execution up to the last specification case.

Lines 7 and 8 conjoin all of the preconditions along the path, and all of the postconditions but the last, as discussed above.

Line 9 constrains all of the intermediate states, except the last, to satisfy the frame axioms of the specification cases along the path.

Line 10 gives the frame axiom of the effective specification. Any state that is declared assignable by any specification case along the path must be considerable assignable for the effective specification.

Lines 11–16 give the postconditions for the effective specification. The postconditions include all of the parameter passing constraints and all of the postconditions from formula (2.1) on page 41.

Line 12 maintains the constraint on the abstract arguments from the client's perspective. It also adds a constraint mapping the result from the last specification case to the special \result variable.

Line 13 maintains the constraints on the passing of parameters and results through the execution.

Line 14 asserts that all of the postconditions of the specification cases along the path hold *in the post-state of those cases*. The frame conditions may allow some of those postconditions to no longer hold in the post-state of the effective specification.

Line 15 constrains all of the intermediate states, including the last, to satisfy the frame axioms of the specification cases along the path. Compare this to line 9.

Line 16 constrains the final intermediate state, represented by $\overline{var}_{n+1}$, to be the post-state of the effective specification. Compare this to line 4. Although these predicates constrain the first and last intermediate states to the same variables, $\overline{var}$, those variables in the requires clause represent the pre-state, while in the ensures clause they represent the post-state. Thus, these two similar-looking constraints do not force the pre- and post-states to be equal, as one might naively assume.

Each possible parallel path through the specification composition graph is represented by a single-path specification case like the one in Figure 2.14. To form the effective specification of the entire graph, I simply conjoin these single-path specifications using JML's also operator. This "parallel composition", alluded to earlier, allows the client programmer to abstractly reason about an invocation of a method in the presence of accepted assistance.

2.4.2.3 An Example Effective Specification

Although the general form of effective specifications presented in the previous section appears quite complex, in practice most of the detail cancels out. I demonstrate this for the running example.

Suppose a client program includes the following code:

```
FigureElement fe = ...
fe.move(2,13);
```

and suppose that FigureElement has accepted the MoveLimiting assistant. The specification composition graph for this method call is the one I have been using as a running example. Figure 2.15 on the next page gives the effective specification for one path through this specification composition graph, the path shown in Figure 2.13.

```
1  forall float x1, y1; forall float x2, y2; forall float x3, y3; forall float x4, y4;
2  forall float oldX, oldY; forall FigureElement \result3, \reply3;
3  requires
4     this.x == x1 && this.y == y1
5     && argX1 == dx && argY1 == dy
6     && dx2 == argX1 * \aspect_1.MAX_DISTANCE / (Math.sqrt(argX1*argX1 + argY1*argY1))
         && dy2 == argY1 * \aspect_1.MAX_DISTANCE / (Math.sqrt(argX1*argX1 + argY1*argY1))
         && \result2 == \reply3
7     && Math.sqrt(argX1*argX1 + argY1*argY1) > \aspect_1.MAX_DISTANCE
         && oldX == x2 && oldY == y2 && true
8     && true && x3 == (oldX + dx2) && y3 == (oldY + dy2) && \result2 == this
9     && x1 == x2 && y1 == y2;
10 assignable this.x, this.y;
11 ensures
12    argX1 == dx && argY1 == dy && \result == \result3
13    && dx2 == argX1 * \aspect_1.MAX_DISTANCE / (Math.sqrt(argX1*argX1 + argY1*argY1))
         && dy2 == argY1 * \aspect_1.MAX_DISTANCE / (Math.sqrt(argX1*argX1 + argY1*argY1))
         && \result2 == \reply3
14    && true && x3 == (oldX + dx2) && y3 == (oldY + dy2) && \result2 == this
         && \result3 == \reply3
15    && x1 == x2 && y1 == y2 && x3 == x4 && y3 == y4
16    && this.x == x4 && this.y == y4;
```

Figure 2.15 Effective Specification for the Path Shown in Figure 2.13

The line numbers for this effective specification correspond to those in the earlier figure showing the general form. Thus the description of each of the lines, given in the previous subsection, can also be read in conjunction with this new figure. The reader may find it helpful to do so.

As mentioned above, much of the information in the effective specification can be suppressed. By using transitivity of equality (within clauses); the rules that false is the zero, and true is the identity, of conjunction; and dropping unused intermediate state variables, I reduce the effective specification in Figure 2.15 to the one shown in Figure 2.16 on the facing page. "Resugaring" this specification case, and adding in the simplified and sweetened specification case for the other path in Figure 2.12, yields the effective specification shown in Figure 2.17. With the understanding that \aspect_1 refers to the instance of MoveLimiting, this effective specification matches our intuition.

Thus, my proposed specification constructs for advice, with the specification composition algorithm sketched here, allow clients to abstractly reason about advised code. Furthermore, if the applicable advice can be modularly identified, as with my proposed language features, then this abstract reasoning is also modular.

2.5 Discussion

This section discusses several interesting issues raised by my proposed language features in regards to other features of AspectJ. I also consider some questions raised by my algorithm for generating effective specifications, and touch on issues of tool support

```
  forall float oldX, oldY;
  requires
    Math.sqrt(dx*dx + dy*dy) > \aspect_1.MAX_DISTANCE
    && oldX == this.x && oldY == this.Y
  assignable this.x, this.y;
  ensures
    this.x == (oldX + dx * \aspect_1.MAX_DISTANCE / (Math.sqrt(dx*dx + dy*dy)))
    && this.y == (oldY + dy * \aspect_1.MAX_DISTANCE / (Math.sqrt(dx*dx + dy*dy)))
    && \result == this;
```

Figure 2.16 Simplified Version of Effective Specification from Figure 2.15

```
  requires
    \aspect_1.distance(dx,dy) > \aspect_1.MAX_DISTANCE
  assignable this.x, this.y;
  ensures
    x == (\old(x) + dx * \aspect_1.MAX_DISTANCE / \aspect_1.distance(dx,dy))
    && y == (\old(y) + dy * \aspect_1.MAX_DISTANCE / \aspect_1.distance(dx,dy))
    && \result == this;
also
  requires
    \aspect_1.distance(dx,dy) <= \aspect_1.MAX_DISTANCE
  assignable this.x, this.y;
  ensures
    x == (\old(x) + dx) && y == (\old(y) + dy) && \result == this;
```

Figure 2.17 Effective Specification Derived from the Specification Composition Graph in Figure 2.12

2.5.1 Language Issues

2.5.1.1 Call and Execution Join Points

Explicit acceptance of assistance interacts in interesting ways with call and execution pointcuts. Consider the MoveLimiting example from Figure 2.2 on page 15, but suppose instead of using execution pointcut descriptors, it used call ones. If FigureElement's module in Figure 2.1 on page 13 accepted this hypothetical, call-based MoveLimiting assistant, but no client did, then the advice in the assistant would only apply to the calls to setX and setY within the body of the move method. Calls to any of the three methods from outside FigureElement would not be advised. This is because the hypothetical assistant only uses call join points. Invocations of FigureElement's setX, setY, or move methods from client code would not be advised because no client code accepts the assistance. The actual MoveLimiting assistant uses execution join points, rather than call ones. Thus, FigureElement's module can accept the assistance and the advice applies to invocations of setX, setY, or move from all client modules—MoveLimiting is an implementation utility.

One might suppose that I could change the semantics of call join points, and eliminate execution join points altogether, by relying on the explicit acceptance of assistance to determine when to execute the advice

code. But where should such advice go if an assistant is accepted by both a client and an implementation module? The compiler cannot modularly know where all accept clauses in a program might appear, and so there is no modular answer to the question. Thus both call and execution join points are required in the language.

The call-execution distinction also affects the distinction between client and implementation utilities, discussed in Section 2.3.2. An assistant using call join points is not a viable implementation utility. Conversely, an assistant using execution join points is not a viable client utility. To write an assistant that could fill either role, one would have to write pointcuts that used a combination of call and execution pointcut descriptors, along with the dynamic-context pointcut descriptor cflowbelow to prevent duplicate application of the advice. Something like

$$(\,\mathsf{call}(\,S\,)\;||\;\mathsf{execution}(\,S\,)\,)\;\&\&\;!\mathsf{cflowbelow}(\,\mathsf{call}(\,S\,)\,)$$

might suffice. This pointcut applies to any call or execution of the method matched by S, provided that the method does not already have a frame on the call stack (excepting the top frame). It may be reasonable to define a syntactic sugar for such pointcuts. However, the cflowbelow pointcut requires runtime checks whereas call and execution do not. The technique also suffers from the general AspectJ problem of avoiding binding on the first recursive call versus on all recursive calls, as is the case with cflowbelow. Thus, restricting a given assistant to being exclusively a client utility (using call) or an implementation utility (using execution) is likely to be more efficient, and is more likely to be correct.

2.5.1.2 Other Features of AspectJ

It seems that ordering advice based on the ordering of accept clauses might eliminate the need for AspectJ's precedence declarations. While technically this seems to be the case, I are not claiming that relying on the ordering of accept clauses is any less error prone than relying on precedence declarations to control the order of interacting aspects. On the one hand, it would be quite easy to accidentally misorder the acceptance of two pieces of interacting advice in a concern map or the accept clauses for a module. Compared to precedence declarations, the use of explicit acceptance also spreads out and makes less obvious the kind of decisions that precedence declarations (may) record in one place. On the other hand, when writing an aspect, it is impossible to know all the potential other aspects over which it should have precedence. And effective specifications allow clients to determine that the actual order used might have the wrong behavior.

The current work does not address AspectJ's introduction mechanisms and declare parents construct. An aspect that used introduction to replace an inherited method of a class with an overriding method would clearly change the behavior of that class and would therefore be an assistant. On the other hand, suppose an aspect introduced a new, non-overriding method to a class. Since no other code could have called that new method, this introduction should not change the behavior of existing code. So such an introduction could be allowed in a spectator. (This case is similar to the introduction of external generic functions via MultiJava's "open class" mechanism [38, 43, 46].) However, I leave this decision for future work, because introduction involves subtle modularity issues, particularly for avoiding runtime ambiguities. These issues are made more complex by the possibility that the newly introduced methods might be advised by existing aspects, or that a change in the base program might make a previously "fresh" introduced method into an overriding one.

The current work also does not address AspectJ's declare error and declare warning constructs. But these constructs do not change the behavior of a program in any way. Instead they provide advice to the compiler itself, telling the compiler that if certain join points are detected at compile-time, then an error or warning should be issued. Thus, these constructs can be allowed in spectator aspects.

An aspect that used the declare soft construct, which converts checked exceptions to unchecked ones, would clearly change the control flow of a program to which it was applied. Such an aspect is thus an assistant.

An alternative technique in regular AspectJ for implementing the hyper-cutting pattern (discussed in Section 2.2.1.1) is to use within and withincode pointcut descriptors to statically limit the code to which a particular piece of advice applies. Unlike accepts clauses and concern maps, this approach buries the applicability of advice within pointcut descriptors. The within approach to hyper-cutting makes it even harder to find applicable advice than with the (already non-modular) marker interface approach. The within approach is a handy implementation technique, however. In my prototype implementation of accepts clauses and concern maps, I use the within technique in the intermediate code.

2.5.2 Specification Issues

It seems that the construction of specification composition graphs could lead to a combinatorial explosion of paths, especially when multiple pieces of around advice advise a given join point. This seems unavoidable in general; however, in practice two things would mitigate this problem. First, it is unlikely that a particular piece of advice will proceed more than once. The most common idiom by far is proceeding exactly once, though some pieces of advice for optimizations, for example, do not proceed when a cached result is available. Secondly, the conditions along some paths may be mutually exclusive, meaning that those paths cannot be taken in practice. The problem is also somewhat controlled by my insistence on a total ordering for advice execution.

There are also some interesting interactions between abstract module interfaces and implementation utilities. A module, *M*, accepting assistance from an implementation utility aspect, might give the effective specification of a method in its interface, rather than the specification ignoring advice. Then a facility could be provided to allow *M* to hide the acceptance of this assistance from client modules. Based on the ordering of advice presented in Section 2.2.1.1 and the specification composition graph algorithm given in Section 2.4.2.1, the programmer of a client module could calculate the effective specification of a method in *M*, by composing the specifications of any client-accepted assistance with the effective specification exposed by *M*.

Dynamic-context pointcut descriptors, like cflow, cflowbelow, and if, present challenges for specification composition. It is not possible, in general, to statically determine which join points such a pointcut will match. A first approach to handling dynamic-context pointcuts is to construct an effective specification that non-deterministically assumes that the advice may or may not be executed. A better solution is probably to include the dynamic-context pointcut descriptors as predicates that may be used in preconditions of specification cases. The specification composition algorithm would still include cases assuming that the advice may or may not be executed, but the cases would be guarded with the dynamic-context predicates. Then a verifier, or the programmer, could eliminate some cases if he had knowledge of the dynamic context.

2.5.3 Tool Support

My algorithm for constructing effective specifications naturally raises the possibility of tool support for automatically generating effective specifications. This is somewhat related to the JMLdoc tool, in the JML tools suite [28]. JMLdoc will generate the specification of an overriding method by conjoining the method's specification with that from any overridden superclass methods (through the entire class hierarchy). A documentation tool for AspectJML could similarly generate effective specifications for code that used implementation utilities.

Perhaps more interesting is the possibility of adding automatic generation of effective specifications to an integrated development environment like the AJDT for ECLIPSE. The development environment could

calculate the effective specification for a piece of code "on the fly". In the current AJDT, markers appear in the source code editor, indicating where aspects might apply. But with support for effective specifications, the development environment could provide a pop-up window describing the meaning of a particular piece of code, given the accepted assistance. One might even consider using a theorem-proving toolkit to see if the preconditions of the effective specification are mutually exclusive, indicating that the effective specification cannot be satisfied.

Although my proposed language extensions would make the generation of effective specifications more efficient, AJDT already maintains a global registry of aspect and base program interactions. Essentially the build configuration of the system acts as a root-level concern map. So support for effective-specification-based tools could be added to AJDT without extending the core AspectJ language.

With AJDT, the global interaction registry compensates for the lack of modularity. Such tool support is not, however, a complete solution to the problems with modular reasoning.

Integrated development environments for object-oriented languages also provide support for manipulating and reasoning about polymorphic method calls and overriding methods, but behavioral subtyping is still needed. Behavioral subtyping helps programmers to think about object-oriented code. It also helps them to design code that is easier to understand. This is reinforced by a wealth of articles on trade web sites that discuss the use of the "Liskov Substitution Principle" [103], the popular name for the property provided by behavioral subtyping.[19] I contend that the discipline described here, and the enabling language features, provide similar benefits in guiding programmers' thinking and design efforts. I leave a thorough evaluation of this contention to future work, after the development of tools implementing my proposal.

2.6 Related Work

Kiczales and Mezini [80] argue that the modularity properties of aspect-oriented programs should be understood in terms of "aspect-aware interfaces". These interfaces are based on the global configuration of a system and essentially provide a bi-directional mapping from methods to associated advice, and advice to advised methods. Assuming the existence of such an aspect-aware interface, the authors then argue that reasoning about cross-cutting concerns is simpler in an aspect-oriented implementation than in a purely object-oriented implementation of the same program.

The arguments of the paper are purely rhetorical; the paper presents no formal analysis nor any case studies. A single example is used to support the claims. That example examines the process necessary to refactor a program written in an aspect-oriented style, versus the same program written in a pure object-oriented style. In either program an implementation-level search is needed to understand the design. In the aspect-oriented version, the change necessary to support the refactoring can be localized in a single module. However, it does not seem that the localization of this change is because of the cross-cutting expressiveness of aspect-oriented programming *per se*. Rather, the change can be localized because AspectJ provides predicates over the current call stack: cflow and cflowbelow. The refactoring in their example could be done in a pure object-oriented language that had such predicates.[20]

So essentially, their argument is that reasoning is no less modular in aspect-oriented programs than it is in object-oriented programs where cross-cutting concerns are scattered and tangled. The objective of my work

[19] A Google search for the quoted string "Liskov Substitution Principle" yielded over 7,000 results on June 19, 2005.

[20] The seed for this idea was planted by Christina Lopes in a post to the aosd-discuss mailing list, where she mentioned including call-stack predicates in a non-aspect-oriented language.

is to demonstrate that, given the appropriate design discipline, reasoning in aspect-oriented programs can actually be more modular.

Nevertheless, Kiczales and Mezini's argument that the understanding of cross-cutting concerns requires some understanding of the global system configuration is compelling. For if there is no global system configuration, there is no substrate for concerns to cut across. This idea of lifting concerns into the global configuration was first proposed in my earlier work with Leavens [40] and is refined here. Most modern programming languages offer module hierarchies (e.g., Java's system of classes, nested classes, and packages). It is natural to generalize the acceptance of aspects to any level in such a hierarchy. This is what I have done with concern maps.

The "aspectual collaborations" of Lieberherr et al. [102] are somewhat related to my concern maps (see also Ovlinger's dissertation [130]). With aspectual collaborations, advice is declared within modules using abstract representations of the pointcuts to be matched. Modules must be explicitly composed. This composition reifies the pointcuts, making explicit the ways in which one module's advice might attach to another module's methods. While aspectual collaborations offer some nice modularity properties, they require all composition to be done at the top-level, instead of at any level of the module hierarchy as for concern maps. Aspectual collaborations do not address the problem of reasoning about AspectJ programs, since aspectual collaborations do not use AspectJ. Finally, it is unclear how anything with the flexibility of spectators could be expressed using aspectual collaborations.

Katz and Gil [76] suggest that the body of work on "superimposition", for reasoning about distributed algorithms, might provide a fertile ground for ideas in developing aspect-oriented programming. (Bougé and Francez [24] give an approachable introduction to superimposition.) Katz and Gil briefly sketch three categories of aspects. Their "spectative" category matches my notion of spectators. The other two categories of aspects they mention map to my notion of assistants. However, they do not consider a language design that might help enforce and exploit these distinctions. Because of this they do not address anything like my concern maps and they do not talk about how one might enforce that declared spectators have no observable side effects. Their suggestion regarding mining the work on superimposition in developing aspect-oriented programming seems reasonable. Much of the work on superimposition is concerned with proving properties of distributed algorithms, or adding additional provable properties to distributed algorithms without disturbing other underlying properties. My work can be viewed as extending these more theoretical ideas into practical language designs.

Also attempting to make superimpositions more practical, Sihman and Katz [148] describe SuperJ, a notation and preprocessor for superimpositions. Superimpositions in SuperJ are defined as sets of generic parameterized aspects and singleton classes. When applying a superimposition to a given "basic program", an "activation" is used to bind the superimposition's parameters to classes and methods in the basic program. Superimpositions have specifications, including requirements on the basic program and results assured if those requirements are met. The specification language used is informal, though the authors note that a temporal logic could be used and suggest mapping to model checking systems for Java as a means of verification. Verification conditions and a sketch of a soundness proof are given. The authors note that if a model of all basic programs satisfying the superimposition's requirements can be built, then verification of the superimposition against this model will independently prove the correctness of all applications of the superimposition.

The implementation of SuperJ is as a preprocessor that uses their "activations" to generate regular AspectJ aspects from the generic aspects of a superimposition. Running these aspects and the basic program through the regular AspectJ compiler yields the final program.

Superimpositions may be combined using merging (both merged superimpositions apply to the basic

program but not to each other) or sequencing (one superimposition applies to the basic program, the other to the resultant combination).

SuperJ was designed to meet a different set of goals then the ones that I address. Sihman and Katz are concerned with global properties of the basic program and superimposition, while my work is more about detailed design and specification. The authors claim that AspectJ and other languages do not provide a mechanism for specifying and reasoning about a collection of aspects, but this ignores the possibility of using a single aspect with nested aspects to encapsulate a superimposition-like construct. The authors also neglect to mention that they do not actually have a specification language. The authors do not address whether a closed-world assumption is needed over their basic programs, though it seems to be. Thus, their system is not appropriate for specifying aspects to be applied to components, which themselves must be combined with other code to be useful. My spectators can be applied this way, as can implementation-utility assistants.

Krishnamurthi et al. [85] present an application of model checking to verifying that aspects do not violate properties verified of the base program. The technique is novel in that a base program interface may be calculated from the base program and a fixed set of pointcut descriptions (which may depend on dynamic properties). Once calculated, verification of aspects can be done without access to the base program (assuming that the base program interface and pointcut descriptions don't change). Another novel technique is to use the model checking machinery itself, applied to a "reversed machine", to calculate the aspect-aware interface.

The work assumes a fixed base program and pointcut descriptions, a state machine model of the program, specifications in the temporal logic CTL, and no access to the base program source after interface calculation. Properties to be checked must be known when the aspect-aware interface is constructed, i.e., the technique "is designed to establish the *preservation* of program properties by aspects." Like the work of Sihman and Katz above, this work focuses on global properties rather than detailed design and specification.

Zhao and Rinard [162] implement the specification composition ideas described here, but based on Leaven's and my earlier description of those ideas [39]. Our earlier work does not consider advice that can proceed more than once, nor does it include a mechanism like \reply for referring to the results of proceed in specifications; and so neither does theirs. Their implementation simply takes the specifications of aspect-oriented code, weaves them together based on our composition, and attaches the resulting specifications to woven Java code emitted by the AspectJ compiler. However, the current AspectJ compiler no longer emits Java source code, so there is no longer a plain Java substrate to which woven specifications can be attached. Also, their implementation of our ideas preceded our design of concern maps, so the implementation requires scattering of accepts clauses into every class that uses an assistant aspect. They do not treat spectators at all.

I discuss other related work on reasoning in aspect-oriented languages that is more formal in nature in subsequent chapters, where comparisons to my work can be made more readily [8, 48, 146].

Concern maps, especially at the root of a package hierarchy, bear a strong resemblance to configuration files in a build management system like ANT or MAVEN [69, 106]. One consequence of my proposal is to elevate a portion of the build configuration into the implementation language. This standardizes the description of aspect and base code composition, allowing a variety of tools to use the same composition semantics. Using concern maps to specify this composition does not preclude the use of a build management system for other tasks in the build process.

2.7 Conclusion

In this chapter, I have presented the MAO discipline for modular aspect-oriented reasoning. The MAO discipline addresses the twin problems of modular reasoning in aspect-oriented languages: (1) unseen aspects

may apply to the code, and (2) aspects may be developed without complete knowledge of the code that will be advised. The discipline addresses these problems by separating aspects into two sorts: benign spectators and surprising assistants. The discipline also requires that the aspect author and the programmer of advised code share the burden of ensuring modular reasoning.

I have argued that a few additional language features are sufficient to support the MAO discipline in a language like AspectJ. My proposed features statically separate aspects into assistants and spectators. Assistants have the full power of AspectJ's aspects, but to maintain modular reasoning I require that assistants be explicitly accepted. Spectators are constrained to not modify the behavior of the modules that they view. This allows modular reasoning about the advised code, even if spectators remain unseen.

My proposal introduces concern maps to allow acceptance of assistance, while avoiding the scattering of duplicate accept clauses throughout a program.

I have described an evaluation of the practical effect of my proposed language features. My evaluation looked at AspectJ style guidelines and three sets of AspectJ examples. The ATLAS case study identifies style rules that are equivalent to my definition of spectators. I studied the examples from the *AspectJ Programming Guide*, and Kiselev's and Laddad's books. For the AspectJ constructs considered in the current work, my language features can handle their examples with no changes in most cases, and minor changes otherwise. The ready identification of places to accept assistance from client or implementation utilities in these examples supports my contention that experienced aspect-oriented programmers are already using disciplines, like the MAO discipline, that enable modular reasoning.

I have described extensions to the Java Modeling Language that allow one to write specifications for advice. I have given a formal model for advice composition that allows one to determine the effective specification of a method in the presence of accepted assistance. This model also illustrates the reasoning a programmer must undertake even in the absence of formal specifications.

The major technical challenge for my proposal is checking that aspects declared as spectators meet my definition, as discussed in Section 2.2.2. I have specified constraints on spectators that allow modular reasoning about their (lack of) effect on control flow. A type system that restricts aliasing and mutation allows modular reasoning about spectators (lack of) effect on the relevant state of the modules they view. The remainder of this dissertation presents:

— an aspect-oriented calculus for investigating these ideas in a formal setting, and

— a sound type-system for the calculus that statically enforces my proposed restrictions on spectators.

This formal foundation is essential to demonstrating the soundness of the reasoning techniques sketched here. AspectJ possesses a rich semantics of advice binding and pointcuts. This means that even a core language representing a small subset of the full language presents formidable obstacles to formalization. Through a series of increasingly rich formalisms, I demonstrate that a firm foundation for studying reasoning in AspectJ can be built. I use this foundation to prove that my proposed restrictions on spectators are indeed statically verifiable.

CHAPTER 3. MINIMAO$_1$: INVESTIGATING THE SEMANTICS OF PROCEED

This chapter describes a core aspect-oriented language, *MiniMAO$_1$*. MiniMAO$_1$ is intended to provide a formal foundation for studying the spectators and assistants proposed in the preceding chapter. It is designed to formalize core features of AspectJ. In particular, MiniMAO$_1$ models the ability of advice in AspectJ to:

— change the target object of an advised operation, possibly affecting dynamic method selection;

— change or capture the arguments to, or results from, an advised operation; and

— affect control flow to an advised operation, causing it to be executed once, multiple times, or not at all.

These abilities are central to advice that introduces "surprising" behavior into advised code. Thus, MiniMAO$_1$ provides the desired foundation for further study.[1]

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

For clarity, I begin with a core object-oriented calculus with classes. I then extend this object-oriented calculus with aspects and advice binding.

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

In this section I introduce *MiniMAO$_0$*, a core object-oriented calculus with classes. MiniMAO$_0$ is an imperative calculus derived from Classic Java [61]. But, following the lightweight philosophy of Featherweight Java [73], I eliminate interfaces, super calls, method overloading, and let expressions. Since eliminating let expressions eliminates implicit sequencing [1], I introduce explicit expression sequencing. I adopt Featherweight Java's technique of treating the current program and its declarations as global constants. This avoids burdening the formal semantics with excess notation—when MiniMAO is fully developed the notation is quite heavy enough.

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

3.1.1 Syntax of MiniMAO$_0$

The syntax for MiniMAO$_0$ is given in Figure 3.1 on the next page. 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.

[1] Portions of this chapter appeared in a paper at Foundations of Aspect-Oriented Languages (FOAL) 2005, co-authored with Gary Leavens [42].

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

$c, d \in \mathscr{C}$, the set of 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 3.1 Syntax of MiniMAO$_0$

In MiniMAO$_0$ the declarations are all of classes; later calculi will add other sorts of 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 my 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$_0$, I define Object to contain 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 I will take it to be the set of all valid Java identifiers. I 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. I omit field initializers from the calculus.

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 I 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, I follow Classic Java in using the syntax cast $t\ e$ to represent the Java cast $(\ t\ )\ e$. Finally, I 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 I choose the direct approach.

3.1.2 Operational Semantics of MiniMAO$_0$

I describe the dynamic semantics of MiniMAO$_0$ using a structured operational semantics [58, 137, 160]. The semantics is given in Figure 3.2 on the following 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.

I add two expressions for the operational semantics of MiniMAO$_0$ that do not appear in the static syntax. To model state, I 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 calculus I just require that $\mathscr{L}$ is some countable set. To model method execution independently from method calls, I 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, mapping the 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 subsequent subsection.

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

The evaluation context rules, denoted by $\mathbb{E}$, serve as implicit congruence rules and give a non-constructive definition of 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 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 $\mathbb{E}.f$ and cast t $\mathbb{E}$ 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 I leave the set *Stack* undefined for now; the aspect-oriented calculus 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 $[t \bullet \{f \mapsto v \cdot f \in dom(fieldsOf(t))\}]$. 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

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

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.

I 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

Syntax extensions:

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

Objects:

$$o ::= [t.F]$$
$$F : \mathscr{F} \to \mathscr{V}$$

Evaluation contexts:

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

Evaluation relation:

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3.2 Operational Semantics of MiniMAO$_0$

closure of the $\hookrightarrow$ relation restricted to the range $\mathscr{E} \times \mathit{Stack} \times \mathit{Store}$.)

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, I briefly describe the intuition behind each of the evaluation rules, and I give a small example program and trace its evaluation.

3.1.2.1 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 3.3 on the next 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 $e\{\!|e'/\mathit{var}|\!\}$ denotes the standard capture-avoiding substitution of e' for *var* in e.) 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 at the bottom of Figure 3.2 on the facing page.)

The GET and SET rules both lookup 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. (This rule takes advantage of the definition of $\oplus$, which lets the right-hand argument replace bindings in the left-hand map.) 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 3.4 on page 59 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.

3.1.2.2 Sample Evaluation

In this section I illustrate several of the evaluation rules with an example. Figure 3.5 on page 59 gives the example program, which models the natural numbers. The program uses two classes: a general natural number class, Natural, and a special class to model Zero.

The figure includes javadoc-style comments describing all the methods, though a couple of these warrant further explanation.

— The Zero class overrides the pred method to just return this, because zero is considered to be its own predecessor in this model of the natural numbers.

Map update:

$$\oplus : \mathscr{P}(\Phi \rightharpoonup \Psi) \times (\Phi \rightharpoonup \Psi) \to \mathscr{P}(\Phi \rightharpoonup \Psi)\text{, polymorphic in sets } \Phi \text{ and } \Psi$$

$$A \oplus (\phi \mapsto \psi) = \{\phi' \mapsto \psi' \cdot (\phi' \neq \phi \wedge A(\phi') = \psi') \vee (\phi' = \phi \wedge \psi' = \psi)\}\text{, for } \phi, \phi' \in \Phi \text{ and } \psi, \psi' \in \Psi$$

Field lookup:

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

Method lookup:

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

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

Method type lookup:

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

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

Valid method overriding:

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

$$\frac{methodType(d, m) = t_1 \times \ldots \times t_n \to t}{override(m, d, t_1 \times \ldots \times t_n \to t)} \qquad \frac{}{override(m, \text{Object}, t_1 \times \ldots \times t_n \to t)}$$

Valid class:

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

Figure 3.3 Auxiliary Functions for MiniMAO$_0$

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

Figure 3.4 Subtyping in 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 3.5 Sample MiniMAO$_0$ Program

— The add method in Natural calculates the sum by adding the predecessor of the current number and the successor of the argument (since $t + n = (t - 1) + (n + 1)$). The Zero class overrides the add method to just return the argument, so the addition terminates.

The interpretation of instances of these classes is that the value of an instance of Zero is 0, and the value of an instance of Natural is 1 plus the value of its predecessor.

The last line in the sample program uses this model of the natural numbers to calculate $1 + 2$. The listing below traces the evaluation of this expression in MiniMAO$_0$. The *current redex*—the term to be evaluated next—is italicized at each stage. I omit type information on fun terms, because it is not used by the evaluation rules.

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

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

where $S_0 = \{\text{loc0} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}]\}$

$\hookrightarrow \langle$*(fun succ⟨this⟩.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)

$\hookrightarrow \langle$*loc1.setPred(loc0)*.add(new Zero().succ().succ()), $J, S_1\rangle$ (NEW)

where $S_1 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}\,.\,\{\text{pred} \mapsto \text{null}\}] \end{array} \right\}$

$\hookrightarrow \langle$*(fun setPred⟨this,pred⟩.(this.pred = pred);this (loc1,loc0))*.add(new Zero().succ().succ()), $J, S_1\rangle$ (CALL)

$\hookrightarrow \langle$*((loc1.pred = loc0);* loc1).add(new Zero().succ().succ()), $J, S_1\rangle$ (EXEC)

$\hookrightarrow \langle$*(loc0; loc1)*.add(new Zero().succ().succ()), $J, S_2\rangle$ (SET)

where $S_2 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}\,.\,\{\text{pred} \mapsto \text{loc0}\}] \end{array} \right\}$

$\hookrightarrow \langle$loc1.add(*new Zero()*.succ().succ()), $J, S_2\rangle$ (SKIP)

$\hookrightarrow \langle$loc1.add(*loc2.succ()*.succ()), $J, S_3\rangle$ (NEW)

where $S_3 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}\,.\,\{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}] \end{array} \right\}$

$\hookrightarrow \langle$loc1.add(*(fun succ⟨this⟩.new Natural().setPred(this) (loc2))*.succ()), $J, S_3\rangle$ (CALL)

$\hookrightarrow \langle$loc1.add(*new Natural()*.setPred(loc2).succ()), $J, S_3\rangle$ (EXEC)

$\hookrightarrow \langle$loc1.add(*loc3.setPred(loc2)*.succ()), $J, S_4\rangle$ (NEW)

where $S_4 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}\,.\,\{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero}\,.\,\{\text{pred} \mapsto \text{null}\}], \\ \text{loc3} \mapsto [\text{Natural}\,.\,\{\text{pred} \mapsto \text{null}\}] \end{array} \right\}$

$\hookrightarrow \langle$loc1.add(*(fun setPred⟨this,pred⟩.(this.pred = pred);this (loc3,loc2))*.succ()), $J, S_4\rangle$ (CALL)

$\hookrightarrow \langle$loc1.add(*((loc3.pred = loc2);* loc3).succ()), $J, S_4\rangle$ (EXEC)

$\hookrightarrow \langle \text{loc1.add}(\textit{(loc2; loc3)}.\text{succ}()), J, S_5 \rangle$ (SET)

$$\text{where } S_5 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc3} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc2}\}] \end{array} \right\}$$

$\hookrightarrow \langle \text{loc1.add}(\textit{loc3.succ()}), J, S_5 \rangle$ (SKIP)

$\hookrightarrow \langle \text{loc1.add}(\textit{(fun succ⟨this⟩.new Natural().setPred(this) (loc3))}), J, S_5 \rangle$ (CALL)

$\hookrightarrow \langle \text{loc1.add}(\textit{new Natural()}.\text{setPred}(\text{loc3})), J, S_5 \rangle$ (EXEC)

$\hookrightarrow \langle \text{loc1.add}(\textit{loc4.setPred(loc3)}), J, S_6 \rangle$ (NEW)

$$\text{where } S_6 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc3} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc2}\}], \\ \text{loc4} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{null}\}] \end{array} \right\}$$

$\hookrightarrow \langle \text{loc1.add}(\textit{(fun setPred⟨this,pred⟩.(this.pred = pred);this (loc4,loc3))}), J, S_6 \rangle$ (CALL)

$\hookrightarrow \langle \text{loc1.add}(\textit{(loc4.pred = loc3)}; \text{loc4}), J, S_6 \rangle$ (EXEC)

$\hookrightarrow \langle \text{loc1.add}(\textit{loc3; loc4}), J, S_7 \rangle$ (SET)

$$\text{where } S_7 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc3} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc2}\}], \\ \text{loc4} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc3}\}] \end{array} \right\}$$

$\hookrightarrow \langle \textit{loc1.add(loc4)}, J, S_7 \rangle$ (SKIP)

$\hookrightarrow \langle \textit{(fun add⟨this,n⟩.this.pred().add(n.succ) (loc1,loc4))}, J, S_7 \rangle$ (CALL)

$\hookrightarrow \langle \textit{loc1.pred()}.\text{add}(\text{loc4.succ}()), J, S_7 \rangle$ (EXEC)

$\hookrightarrow \langle \textit{(fun pred⟨this⟩.this.pred (loc1))}.\text{add}(\text{loc4.succ}()), J, S_7 \rangle$ (CALL)

$\hookrightarrow \langle \textit{loc1.pred}.\text{add}(\text{loc4.succ}()), J, S_7 \rangle$ (EXEC)

$\hookrightarrow \langle \text{loc0.add}(\textit{loc4.succ()}), J, S_7 \rangle$ (GET)

$\hookrightarrow \langle \text{loc0.add}(\textit{(fun succ⟨this⟩.new Natural().setPred(this) (loc4))}), J, S_7 \rangle$ (CALL)

$\hookrightarrow \langle \text{loc0.add}(\textit{new Natural()}.\text{setPred}(\text{loc4})), J, S_7 \rangle$ (EXEC)

$\hookrightarrow \langle \text{loc0.add}(\textit{loc5.setPred(loc4)}), J, S_8 \rangle$ (NEW)

$$\text{where } S_8 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero}.\{\text{pred} \mapsto \text{null}\}], \\ \text{loc3} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc2}\}], \\ \text{loc4} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{loc3}\}], \\ \text{loc5} \mapsto [\text{Natural}.\{\text{pred} \mapsto \text{null}\}] \end{array} \right\}$$

$\hookrightarrow \langle \text{loc0.add}(\textit{(fun setPred⟨this,pred⟩.(this.pred = pred);this (loc5,loc4))}), J, S_8 \rangle$ (CALL)

$\hookrightarrow \langle \text{loc0.add}(\textit{(loc5.pred = loc4)};\ \text{loc5}), J, S_8 \rangle$ (EXEC)

$\hookrightarrow \langle \text{loc0.add}(\textit{loc4; loc5}), J, S_9 \rangle$ (SET)

$$\text{where } S_9 = \left\{ \begin{array}{l} \text{loc0} \mapsto [\text{Zero} \centerdot \{\text{pred} \mapsto \text{null}\}], \\ \text{loc1} \mapsto [\text{Natural} \centerdot \{\text{pred} \mapsto \text{loc0}\}], \\ \text{loc2} \mapsto [\text{Zero} \centerdot \{\text{pred} \mapsto \text{null}\}], \\ \text{loc3} \mapsto [\text{Natural} \centerdot \{\text{pred} \mapsto \text{loc2}\}], \\ \text{loc4} \mapsto [\text{Natural} \centerdot \{\text{pred} \mapsto \text{loc3}\}], \\ \text{loc5} \mapsto [\text{Natural} \centerdot \{\text{pred} \mapsto \text{loc4}\}] \end{array} \right\}$$

$\hookrightarrow \langle \textit{loc0.add(loc5)}, J, S_9 \rangle$ (SKIP)

$\hookrightarrow \langle \textit{(fun add⟨this,n⟩.n (loc0,loc5))}, J, S_9 \rangle$ (CALL)

$\hookrightarrow \langle \text{loc5}, J, S_9 \rangle$ (EXEC)

To interpret this result, we count the predecessors of loc5 in S_9. From loc5, we must follow the pred field three times (first to loc4 then to loc3 then to loc2) to arrive at an instance of Zero. Thus, we see that $1 + 2 = 3$.

3.1.3 Static Semantics of MiniMAO$_0$

Figure 3.6 on the next page gives the static semantics for MiniMAO$_0$. To avoid overburdening the typing rules, I make the following simplifying assumptions:

— 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 its method body is well typed. That is, the type of its method body is a subtype of the declared return type, 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 3.3 on page 58) 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 static 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

Program typing:

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

Class typing:

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

Method typing:

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

Expression typing:

$$\textsc{T-New}\ \frac{c \in dom(CT)}{\Gamma \vdash \text{new } c() : c} \qquad \textsc{T-Obj}\ \frac{}{\Gamma \vdash \text{new Object}() : \text{Object}} \qquad \textsc{T-Var}\ \frac{\Gamma(var) = t}{\Gamma \vdash var : t} \qquad \textsc{T-Loc}\ \frac{\Gamma(loc) = t}{\Gamma \vdash loc : t} \qquad \textsc{T-Null}\ \frac{t \in \mathcal{T}}{\Gamma \vdash \text{null} : t}$$

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

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

$$\textsc{T-Set}\ \frac{\Gamma \vdash e_1 : u \qquad fieldsOf(u)(f) = t \qquad \Gamma \vdash e_2 : s \qquad s \preccurlyeq t}{\Gamma \vdash e_1.f = e_2 : s} \qquad \textsc{T-Cast}\ \frac{\Gamma \vdash e : s}{\Gamma \vdash \text{cast } t\ e : t} \qquad \textsc{T-Seq}\ \frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1;\ e_2 : t}$$

Figure 3.6 Static Semantics of MiniMAO$_0$

the type environment is extended to include locations. The T-NULL rule says that null can be treated as having any type.

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 fun application form 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.

I 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. [73], 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. [73] 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 my 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.) I choose to use the simpler single cast rule, since the precise correspondence to Java's cast typing rules is not needed for my work and a type safety 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.

3.1.4 Meta-theory of MiniMAO$_0$

The key property of MiniMAO$_0$ is that it is type safe: a well-typed MiniMAO$_0$ program either converges to a value or exception, or else it diverges. I prove this using the usual subject reduction and progress theorems. The proofs closely follow those of Flatt et al. [61].

Before stating and proving a subject reduction theorem, we first need a notion of consistency between a type environment and a store [58, 61]. For the meta-theory, the type environment maps variables and store locations to types, $\Gamma : (\mathcal{V}_{\text{this}} \cup \mathcal{L}) \to \mathcal{T}$.

Definition 3.1 (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. $\forall loc \in \mathcal{L} \cdot S(loc) = [t \,.\, F] \implies$
   (a) $\Gamma(loc) = t$ and
   (b) $dom(F) = dom\big(fieldsOf(t)\big)$ and

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

(c) $rng(F) \subseteq dom(S) \cup \{\mathsf{null}\}$ and

(d) $\forall f \in dom(F) \cdot \big(F\big(f\big) = loc'$ and $fieldsOf(t)(f) = u$ and $S\big(loc'\big) = \big[t' \,.\, F'\big] \implies t' \preccurlyeq u\big)$

2. $\forall loc \in \mathscr{L} \cdot (loc \in dom(\Gamma) \implies loc \in dom(S))$

3. $dom(S) \subseteq dom(\Gamma)$

The following standard substitution lemma will also be useful.

Lemma 3.2 (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 simplify the notation, let $\Gamma' = \Gamma, var_1 : t_1, \ldots, var_n : t_n$ and write $\{\!|\bar{e} / \overline{var}|\!\}$ for $\{\!|e_1 / var_1, \ldots, e_n / var_n|\!\}$. 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:

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

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 $methodType(u''_0, m) = methodType(u'_0, m)$ 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 $e = \left(e'_1.f = e'_2\right)$ and the last step in the type derivation is:

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

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

$$\mathit{fieldsOf}\left(u''_1\right)(f) = \mathit{fieldsOf}\left(u'_1\right)(f) = u.$$

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, $e = \text{cast } t\ e'$. Here the last derivation step is:

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

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 $t' \preccurlyeq t$. □

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 3.3 (Environment Extension). *If* $\Gamma \vdash e : t$ *and* $a \notin \mathit{dom}(\Gamma)$, *then* $\Gamma, a : t' \vdash e : t$.

Proof. The proof is by a straightforward structural induction on the derivation of $\Gamma \vdash e : t$.

For the base case, the last step in the derivation is T-NEW, T-OBJ, T-NULL, T-VAR, or T-LOC. In the first three cases, the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the T-VAR case, $e = var$ and $\Gamma(var) = t$. But $a \notin dom(\Gamma)$, so $var \neq a$. Therefore $\left(\Gamma, a : t'\right)(var) = t$ and the claim holds for this case. The T-LOC case is similar.

The remaining typing rules cover the induction step. By the induction hypothesis, changing the type environment to $\Gamma, a : t'$ does not change the types assigned by any hypotheses. Therefore, the types assigned by each rule are also unchanged and the claim holds. □

Lemma 3.4 (Environment Contraction). *If* $\Gamma, a : t' \vdash e : t$ *and* a *is not free in* e*, then* $\Gamma \vdash e : t$.

Proof. The proof is by a straightforward structural induction on the derivation of $\Gamma, a : t' \vdash e : t$.

For the base case, the last step in the derivation is T-NEW, T-OBJ, T-NULL, T-VAR, or T-LOC. In the first three cases, the type environment does not appear in the hypotheses of the judgment, so the claim holds. For the T-VAR case, $e = var$ and $\left(\Gamma, a : t'\right)(var) = t$. But a is not free in e, so $var \neq a$. Therefore $\Gamma(var) = t$ and the claim holds for this case. The T-LOC case is similar.

The remaining typing rules cover the induction step. By the induction hypothesis, changing the type environment to Γ does not change the types assigned by any hypotheses. Therefore, the types assigned by each rule are also unchanged and the claim holds. □

Lemma 3.5 (Replacement). *If* $\Gamma \vdash \mathbb{E}[e] : t$, $\Gamma \vdash e : t'$, *and* $\Gamma \vdash e' : t'$, *then* $\Gamma \vdash \mathbb{E}[e'] : t$.

Proof. By examining the evaluation context rules and corresponding typing rules, we see that $\Gamma \vdash e : t'$ must be a sub-derivation of $\Gamma \vdash \mathbb{E}[e] : t$. Now the typing derivation for $\Gamma \vdash \mathbb{E}[e'] : t''$ must have the same shape as that for $\mathbb{E}[e] : t$, except for the sub-derivation for $\Gamma \vdash e' : t'$. However, because this sub-derivation yields the same type as the sub-derivation it replaces, it must be the case that $t'' = t$. □

Lemma 3.6 (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* $t' \preccurlyeq t$.

Proof. The proof is by induction on the size of the evaluation context $\mathbb{E}$, where the size is the number of recursive applications of the syntactic rules necessary to build $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $t' = u' \preccurlyeq u = t$.

For the induction step we divide the evaluation context into two parts so that $\mathbb{E}[-] = \mathbb{E}_1[\mathbb{E}_2[-]]$, where $\mathbb{E}_2$ has size one. The induction hypothesis is that the claim of the lemma holds for all evaluation contexts smaller than the one considered in the induction step. We use a case analysis on the rule used to generate $\mathbb{E}_2$. In each case we show that $\Gamma \vdash \mathbb{E}_2[e] : s$ implies that $\Gamma \vdash \mathbb{E}_2[e'] : s'$, for some $s' \preccurlyeq s$, and therefore the claim holds by the induction hypothesis.

Case 1—$\mathbb{E}_2 = -.m(\, e_1, \ldots, e_n\,)$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-CALL:

$$\frac{\Gamma \vdash e : u \qquad \forall i \in \{1..n\} \cdot \Gamma \vdash e_i : u_i \qquad methodType(u, m) = s_1 \times \ldots \times s_n \to s \qquad \forall i \in \{1..n\} \cdot u_i \preccurlyeq s_i}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

By the definitions of *override* and *methodType*, $\mathit{methodType}(u', m) = \mathit{methodType}(u, m)$, so T-CALL gives $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 2—$\mathbb{E}_2 = v_0.m(\ v_1, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )$ *where* $p \in \{1..n\}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-CALL:

$$\frac{\begin{array}{c}\Gamma \vdash v_0 : u_0 \qquad \forall i \in \{1..(p-1)\} \cdot \Gamma \vdash v_i : u_i \qquad \Gamma \vdash e : u \qquad \forall i \in \{(p+1)..n\} \cdot \Gamma \vdash e_i : u_i \\ \mathit{methodType}(u_0, m) = s_1 \times \ldots \times s_n \to s \qquad \forall i \in \{1..n\} \setminus \{p\} \cdot u_i \preccurlyeq s_i \qquad u \preccurlyeq s_p\end{array}}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

Now $u' \preccurlyeq u \preccurlyeq s_p$, so by T-CALL $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 3—$\mathbb{E}_2 = (\ l\ (\ v_0, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )\ )$ *where* $p \in \{0..n\}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-EXEC:

$$\frac{\begin{array}{c}\Gamma, \mathit{var}_0 : s_0, \ldots, \mathit{var}_n : s_n \vdash e'' : u'' \qquad u'' \preccurlyeq s \\ \forall i \in \{0..(p-1)\} \cdot \Gamma \vdash v_i : u_i \qquad \Gamma \vdash e : u \qquad \forall i \in \{(p+1)..n\} \cdot \Gamma \vdash e_i : u_i \\ \forall i \in \{0..n\} \setminus \{p\} \cdot u_i \preccurlyeq s_i \qquad u \preccurlyeq s_p\end{array}}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

where $l = \mathsf{fun}\ m\langle \mathit{var}_0, \ldots, \mathit{var}_n \rangle . e'' : (s_0 \times \ldots \times s_n \to s)$. Now $u' \preccurlyeq u \preccurlyeq s_p$, so by T-EXEC $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 4—$\mathbb{E}_2 = -.f$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-GET:

$$\frac{\Gamma \vdash e : u \qquad \mathit{fieldsOf}(u)(f) = s}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

By the first hypothesis of T-CLASS and the definition of field lookup, $\mathit{fieldsOf}(u')(f) = \mathit{fieldsOf}(u)(f)$. Thus, by T-GET, $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 5—$\mathbb{E}_2 = \mathit{cast}\ s\ -$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-CAST:

$$\frac{\Gamma \vdash e : u}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

Because $\Gamma \vdash e' : u'$, $\Gamma \vdash \mathbb{E}_2[e'] : s$ by T-CAST.

Case 6—$\mathbb{E}_2 = -;\ e''$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-SEQ:

$$\frac{\Gamma \vdash e : u \qquad \Gamma \vdash e'' : s}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

Thus, also by T-SEQ, $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 7—$\mathbb{E}_2 = (-.f = e'')$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-SET:

$$\frac{\Gamma \vdash e : u \qquad \mathit{fieldsOf}(u)(f) = u'' \qquad \Gamma \vdash e'' : s \qquad s \preccurlyeq u''}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

As in Case 4 on the preceding page, $\mathit{fieldsOf}(u')(f) = \mathit{fieldsOf}(u)(f)$. Thus, by T-SET, $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 8—$\mathbb{E}_2 = (v_0.f = -)$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-SET, letting $s = u$:

$$\frac{\Gamma \vdash v_0 : u_0 \qquad \mathit{fieldsOf}(u_0)(f) = u'' \qquad \Gamma \vdash e : u \qquad u \preccurlyeq u''}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

Now $u' \preccurlyeq u \preccurlyeq u''$, so let $s' = u'$ and $\Gamma \vdash \mathbb{E}_2[e'] : s'$. □

Theorem 3.7 (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 $e = \mathbb{E}[\text{new } c()]$, $e' = \mathbb{E}[loc]$, $loc \notin dom(S)$, and $S' = S \oplus (loc \mapsto [c \,.\, F])$ where $F = \{f \mapsto \text{null} \cdot f \in dom(\mathit{fieldsOf}(c))\}$.

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

We now show that $\Gamma' \approx S'$. Because $loc \notin dom(S)$, $(\Gamma \approx S) \implies loc \notin dom(\Gamma)$ by part 2 of Definition 3.1 (Environment-Store Consistency) on page 64. Thus part 1 of the definition for $\Gamma' \approx S'$ holds for all $loc' \in \mathscr{L}$, $loc' \neq loc$. Now $S'(loc) = [c \,.\, F]$, $\Gamma'(loc) = c$, $dom(F) = dom(\mathit{fieldsOf}(c))$, $rng(F) = \{\text{null}\} \subseteq dom(S) \cup \{\text{null}\}$, 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 now show that $\Gamma' \vdash \mathbb{E}[loc] : t$. By Lemma 3.3 (Environment Extension) on page 66 and $loc \notin dom(\Gamma)$, we have $\Gamma' \vdash \mathbb{E}[\text{new } c()] : t$. Now $\Gamma' \vdash \text{new } c() : c$ and $\Gamma' \vdash loc : c$, so by Lemma 3.5 (Replacement) on page 67, $\Gamma' \vdash \mathbb{E}[loc] : t$.

Case 2—CALL. Here $e = \mathbb{E}[loc.m(\ v_1, \ldots, v_n\ )]$, $e' = \mathbb{E}[(\ \text{fun } m\langle \text{this}, var_1, \ldots, var_n\rangle.e'' : \tau\ (\ loc, v_1, \ldots, v_n\ )\ )]$ (where $S(loc) = [u \,.\, F]$, $\mathit{methodBody}(u, m) = \text{fun } m\langle \text{this}, var_1, \ldots, var_n\rangle.e'' : \tau$, 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 now show 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 $\mathit{methodType}(u, m) = t_1 \times \ldots \times t_n \to t_m$ 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 $(var_1 : t_1, \ldots, var_n : t_n, \text{this} : u') \vdash e'' : u'_m$ where $u_m \preccurlyeq u'_m = t_m$. By Lemma 3.3 (Environment Extension) on page 66 (and appropriate alpha conversion of free variables in e''), $\Gamma, var_1 : t_1, \ldots, var_n : t_n, \text{this} : u' \vdash e'' : u'_m$.

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 3.6 (Replacement with Subtyping) on page 67 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 now show 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 3.2 (Substitution) on page 65, $\Gamma \vdash e''\{\!|v_0/\,var_0, \ldots, v_n/\,var_n|\!\} : u''$ for some $u'' \preccurlyeq u' \preccurlyeq u$. Finally, by Lemma 3.6 (Replacement with Subtyping) on page 67 $\Gamma \vdash e' : t'$ 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 now show 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 $\Gamma \approx S$, $S(v) = \left[u'\,.\,F'\right]$ where $u' \preccurlyeq s$ and $\Gamma(v) = u'$.

Thus, $\Gamma \vdash v : u'$ and, by Lemma 3.6 (Replacement with Subtyping) on page 67, $\Gamma \vdash \mathbb{E}[v] : t'$ where $t' \preccurlyeq t$.

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

Let $\Gamma' = \Gamma$.

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

For part 1(c), $rng\left(F \oplus \left(f \mapsto v\right)\right) = rng(F) \cup \{v\}$. 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, $\left(F \oplus \left(f \mapsto v\right)\right)\left(f\right) = v$ 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 3.5 (Replacement) on page 67, $\Gamma \vdash \mathbb{E}[v] : t$.

Case 6—CAST. Here $e = \mathbb{E}[\mathsf{cast}\ t''\ loc]$, $e' = \mathbb{E}[loc]$, $S' = S$, $S(loc) = [u \bullet F]$, 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 3.6 (Replacement with Subtyping) on page 67 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 3.5 (Replacement) on page 67, $\Gamma \vdash \mathbb{E}[\mathsf{null}] : t$.

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 3.5 (Replacement) on page 67, $\Gamma \vdash \mathbb{E}[e''] : t$.

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

Theorem 3.8 (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:*

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

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 $S(loc) = [s' \centerdot F]$. 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 definition of *methodType* and that of *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 $S(loc) = [s \centerdot F]$ 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 $S(loc) = [s \centerdot F]$, 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 $\langle e, J, S\rangle \hookrightarrow \langle \text{ClassCastException}, J, S\rangle$ by XCAST. □

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

Theorem 3.9 (Type Safety). *Given a program $P = decl_1 \ldots decl_n\ e$, if $\vdash P$ OK then either the evaluation of e diverges or else $\langle e, \bullet, \emptyset\rangle \overset{*}{\hookrightarrow} \langle x, J, S\rangle$ where one of the following holds for x:*

— *$x = loc$ and $loc \in dom(S)$,*

— *$x = null$,*

— *$x = NullPointerException$, or*

— *$x = ClassCastException$*

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 3.7 (Subject Reduction) on page 69 and Theorem 3.8 (Progress) on the preceding page. □

3.2 MiniMAO$_1$: Adding Aspects

In this section I add advice binding to MiniMAO$_0$, producing the aspect-oriented core calculus MiniMAO$_1$. Continuing with the minimalist philosophy, the join point model of MiniMAO$_1$ is quite simple. The model only includes call and execution pointcut descriptors, the parameter binding forms this, target, and args, and the operators for pointcut union, intersection, and negation. The omission of the dynamic-context pointcut descriptors, such as cflow, is an intentional decision. The techniques for dealing semantically with such descriptors are well understood [157], and such dynamic-context pointcut descriptors do not substantially affect the typing rules for aspects.

MiniMAO$_1$ accurately models AspectJ's semantics for around advice [83], 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 at a call join point 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. These differences are discussed more 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. My approach increases the syntactic complexity of the calculus, but I 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. My calculus is motivated by the insight of Walker et al. [156] 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, I 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, I 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 [48].) 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. My type system is motivated by that of Jagadeesan et al. [74]. I discuss this and other related work in more detail in Section 3.3.

3.2.1 Syntax of MiniMAO$_1$

Figure 3.7 on the next page gives the additional syntax for MiniMAO$_1$. To the declarations of MiniMAO$_0$ I add aspects, with a ranging over the set, $\mathscr{A}$, of aspect names. As for identifiers in MiniMAO$_0$, I 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.

I 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. [74], typing around advice is more challenging than typing before and after advice, since formal parameters in around advice appear in both co- and contravariant positions [31].)

An advice declaration in MiniMAO$_1$ consists of a return type, followed by the keyword around and a sequence of formal parameters. A pointcut description comes next. The pointcut description 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 description 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. I restrict method patterns to a concrete

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

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

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

Figure 3.7 Syntax Extensions for MiniMAO$_1$

return type plus an identifier pattern that is matched against the name of the called method. I choose not to include matching against target or parameter types here because that is just syntactic sugar for the target and args pointcut descriptors.

I leave the set $\mathscr{I}$ of identifier patterns underspecified. Generally, we can think of $\mathscr{I}$ as a regular expression language such that all members of $\mathscr{M}$ are elements of regular expressions in $\mathscr{I}$. For examples, I will treat $\mathscr{I}$ as the set of all legal Java identifiers, but treating 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$_1$ (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. I do 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 $pcd_1 \mid\mid pcd_2$ matches a join point, then the bindings defined by pcd_1 are used and pcd_2 is ignored.

MiniMAO$_1$ also includes proceed expressions, which are only valid within advice. An expression such as e_0.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

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

Figure 3.8 Join Point Stack

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 my intuition for how proceed should work; it has little effect on expressiveness in a language with type-safe around advice. My design also precludes changing the this object at call join points. Such changes would only be visible from other aspects, not the base program. Precluding these changes eliminates some possibilities for aspect interference, a useful property for my work on aspect-oriented reasoning. I am not aware of any use cases demonstrating a need to allow changing the this object.

3.2.2 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.

3.2.2.1 The Join Point Stack

The stack in MiniMAO$_1$ is a list of *join point abstractions*, each of which is a five-tuple denoted by half-moon brackets, $(\!| \ldots |\!)$, as shown in Figure 3.8. A join point abstraction records all the information in a join point that is needed for advice matching and advice parameter bindings, together referred to as *advice binding*. A join point abstraction also includes all the information necessary to proceed from advice to the original code that triggered the join point. A join point abstraction consists of the following parts (most of which are optional and are replaced with "−" when omitted):

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

— an optional value indicating the self object at the join point, used for parameter binding by this pointcut descriptors;

— an optional name indicating 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

$e ::= \ldots \mid \text{joinpt } j(\, e^* \,) \mid \text{under } e \mid \text{chain } \bar{B}, j(\, e^* \,)$

$\bar{B} ::= B + \bar{B} \mid \bullet$

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

$b ::= \langle \alpha, \beta, \beta^* \rangle$

$\alpha ::= var \mapsto loc \mid -$

$\beta ::= var \mid -$

$b \in \mathscr{B}$, the set of advice parameter bindings

Figure 3.9 Additional Expression Forms for the Operational Semantics of Mini-MAO$_1$

— 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.

3.2.2.2 New Expression Forms

The operational semantics relies on three additional expression forms, as shown in Figure 3.9. The first, joinpt, reifies join points of a program evaluation into the expression syntax. A joinpt expression consists of a join point abstraction followed by a sequence of expressions representing the actual arguments to the code under the join point.

The second expression form that I 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 calculus 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.

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

The *binding term*, b, describes how the values of actual arguments should be substituted for formals in the advice body. This substitution is somewhat complex to account for the special binding of the this pointcut descriptor, which takes its data from the original join point, and the target and args pointcut descriptors, which take their data from the invocation or proceed expression immediately preceding the evaluation of the advice body. (No previous formalization of AspectJ has faithfully modeled this binding semantics for target and args.)

I give examples of binding terms in Section 3.2.2.5.

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. This 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 is the declared type of the advice, a function type 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. I include the type information in body tuples to simplify the subject-reduction proof; the type information is not needed for the evaluation rules.

3.2.2.3 Evaluation Rules for MiniMAO$_1$

Next I give an intuitive description of the new evaluation rules in MiniMAO$_1$. These rules are given in Figure 3.10 on the following page. The example evaluations in Section 3.2.2.5 illustrate the rules.

I add new evaluation contexts to handle the joinpt, under, and chain expressions. The semantics replaces proceed expressions with chain expressions, so I do not need an additional context for proceed.

I 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. I 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, my 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 popping join point abstractions from the stack when appropriate.

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.[3] The locations of these instances are recorded in the global *advice table*, *AT*, which is a set of 5-tuples. Each 5-tuple represents one piece of advice. The 5-tuple for the advice t around(t_1 $var_1, \ldots, t_n$ var_n): pcd { e }, declared in aspect a, is $\langle loc, pcd, e, (t_1 \times \ldots \times t_n \to t), \tau \rangle$; 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. I extend the subtyping rules with a rule that all aspects are subtypes of Object, as shown in Figure 3.11 on the next page. 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. I also extend the field lookup function, *fieldsOf*, with an additional rule for aspects as shown in Figure 3.12 on page 79.

Next I describe the new evaluation rules in more detail.

[3] Because of the lack of constructors, there is no obvious mechanism in MiniMAO$_1$ for initializing the state of these implicitly instantiated aspects. Section 3.4 address this issue.

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):

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

$$\langle \mathbb{E}[(\, l\, (\, v_0, \ldots, v_n\,)\,)], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{joinpt } (\!|\text{exec}, v_0, m, l, \tau|\!)(\, v_0, \ldots, v_n\,)], J, S\rangle \quad \textsc{Exec}_{\text{A}}$$
$$\text{where } l = \text{fun } m\langle \mathit{var}_0, \ldots, \mathit{var}_n\rangle.e : \tau$$
$$\langle \mathbb{E}[\text{chain } \bullet, (\!|\text{exec}, v, m, l, \tau|\!)(\, v_0, \ldots, v_n\,)], J, S\rangle$$
$$\hookrightarrow \langle \mathbb{E}[\text{under } e\{\!|\, v_0 / \mathit{var}_0, \ldots, v_n / \mathit{var}_n\,|\!\}], j + J, S\rangle \quad \textsc{Exec}_{\text{B}}$$
$$\text{where } l = \text{fun } m\langle \mathit{var}_0, \ldots, \mathit{var}_n\rangle.e : \tau \text{ and } j = (\!|\text{this}, v_0, -, -, -|\!)$$

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

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

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

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

Figure 3.10 Changes to the Operational Semantics for MiniMAO$_1$

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

Figure 3.11 Additional Subtyping Rule for MiniMAO$_1$

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

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

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 3.12 Auxiliary Functions for MiniMAO$_1$ Operational Semantics

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. 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 method, creates an application form

A similar division of labor is used for EXEC. I 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 where the join point abstraction carries the information about the call necessary to bind advice and to proceed with the original call. This information is: the call kind, the method name, and a function type, τ, for the method. 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 3.12 on the previous 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. I 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.

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

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 3.2.2.4, to find the matching advice in the global advice table. (I leave *adviceBind* underspecified. In particular, I 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 my 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. The general procedure is to substitute for this in the advice body with the location, *loc*, of the advice's aspect and substitute for the advice's formal parameters according to the binding term, b. I describe below how the binding term is used for the substitution. However, before the substitution occurs 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. [75], 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 3.2.2.5.

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. I overload notation to define this substitution for binding terms (see Figure 3.12 on page 79). 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:

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

The $\mathsf{x} \mapsto \mathsf{loc2}$ in the binding term does not use data from the arguments $\langle \mathsf{loc0,loc1}\rangle$; 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—analogous 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 calculus into a pair of rules, with intervening advice binding and execution, is general. The first rule in the new pair replaces the original expression with a joinpt expression, ready for advice binding. The second rule in the pair takes a chain expression, exhausted of advice, and maps it to a new expression like the result expression of the rule from MiniMAO_0. This is how the two new EXEC rules are generated.

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

The EXEC_B rule takes a chain expression that has been exhausted of its advice. It applies the fun term from the chain's join point abstraction to the argument sequence, substituting the arguments for the variables in

the body of the fun term. Like ADVISE, the EXEC_B rule pushes a this join point abstraction onto the stack and wraps its result expression in an under expression.

It would be straightforward to add pointcut descriptors and join points for any of the primitive operations in the original calculus. One 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 my study, I choose not to include join points for the other primitive operations. To do so would just introduce additional notation and bookkeeping.

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

3.2.2.4 Pointcut Matching

Following Wand et al. [157], I use a boolean algebra over binding terms to define a *matchPCD* function, for matching pointcut descriptors to join points. My binding terms, as described in Section 3.2.2.2 above, are somewhat more complex than theirs, since I 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.

Figure 3.13 on the next page gives the boolean algebra. The terms of the algebra are drawn from the set $\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\}$, where binding terms can be thought of as "true" and $\perp$ as "false". The operators in the algebra are conjunction ($\wedge$), disjunction ($\vee$), and complement ($\neg$). The complement of the complement of an element is not necessarily the original element, unless we consider all binding terms to be isomorphic; this effect of this detail on advice binding is discussed below. The binary operators are short circuiting; for example, $b \vee r = b$, ignoring the value of r. One difference in my algebra, versus Wand et al. [157], is in the conjunction of two non-$\perp$ terms. My calculus must consider the bindings from both terms, because I 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* in Figure 3.14 on page 84 are straightforward. If the pointcut descriptor matches the join point stack, then the rules construct the appropriate binding term; otherwise they evaluate to $\perp$. The only 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 where the optional self-object location is provided in the join point abstraction. Once found, if the

Boolean algebra of bindings (adapted from Wand et al. [157]):

$$\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\} \qquad b \in \mathscr{B} \qquad r \in \mathscr{B}_\perp \qquad b \vee r = b \qquad \perp \vee r = r \qquad \perp \wedge r = \perp \qquad b \wedge \perp = \perp \qquad b \wedge b' = b \sqcup b'$$
$$\neg\perp = \langle -, - \rangle \qquad\qquad \neg b = \perp$$

Join of bindings:

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

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

Figure 3.13 Boolean Algebra over Binding Terms

object record in that location is a subtype of the formal parameter type, then the formal named by the pointcut descriptor is mapped to the location; otherwise the result is $\perp$.

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 static type safety, as noted by Jagadeesan et al. [74]. 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.[4] Thus, for static type safety 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 CALL$_A$ rule generates the target type for its join point abstraction, it uses the type of the class declaring the top-most method in the method overriding hierarchy. Thus, the actual target object for a matched call may be a subtype of the target type that was matched exactly. Using the declaring class of this top-most method also means that advice can be written to match a call to any method in a family of overriding methods. Unlike the CALL$_A$ rule, the EXEC$_A$ rule creates a join point abstraction using the actual target type. Again, this is necessary for type safety. 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.

I am also interested in investigating whether a more elaborate type system might permit more expressive pointcut matching while maintaining soundness of the static type system. However, this is orthogonal to my concerns with modular reasoning and so I leave it for future work.

[4] 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.

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

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

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

$$matchPCD(\llparenthesis \_, -, \_, \_, \_ \rrparenthesis + J, \text{this}(\ t\ var\ ), S) = matchPCD(J, \text{this}(\ t\ var\ ), S)$$

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

$$matchPCD(\llparenthesis \_, \_, \_, \_, - \rrparenthesis + J, \text{target}(\ t\ var\ ), S) = matchPCD(J, \text{target}(\ t\ var\ ), S)$$

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

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

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

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

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

Figure 3.14 Pointcut Descriptor Matching for MiniMAO$_1$

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. Like the target rule, the args rule uses exact type matching.

The rules for pointcut descriptor operators simply appeal to the corresponding operators in the binding algebra: union to disjunction, intersection to conjunction, and negation to complement. The definition of complement implies that $\neg\neg pcd \neq pcd$. 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).

3.2.2.5 Example Evaluations in MiniMAO$_1$

This section gives several example MiniMAO$_1$ programs and their evaluations.

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 3.15 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 3.16 on page 87. Below is the evaluation in MiniMAO$_1$. In the evaluation, the initial store is

$$S_0 = \{\mathsf{locA} \mapsto [\mathsf{Asp} \centerdot \{\mathsf{f1} \mapsto \mathsf{null}, \mathsf{f2} \mapsto \mathsf{null}\}]\}.$$

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 $\langle -, \mathsf{s}, \mathsf{arg1}\rangle$, indicating that the target object will be bound to the formal parameter s and the argument to arg1. Figure 3.17 on page 88 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

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⟨this,arg⟩.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⟨this,arg⟩.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 $S_0 = \{\text{loc0} \mapsto [\text{Simple}.\{\text{f} \mapsto \text{null}\}]\}$,

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

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

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

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

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

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

Figure 3.15 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 3.16 Sample Program Showing Advice Binding

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.

$$\langle \textit{new Simple()}.\text{m(new Object())}, \bullet, S_0 \rangle$$

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

$$\text{where } S_1 = \left\{ \begin{array}{l} \text{locA} \mapsto [\text{Asp}.\{\text{f1} \mapsto \text{null}, \text{f2} \mapsto \text{null}\}], \\ \text{loc0} \mapsto [\text{Simple}.\{\text{f} \mapsto \text{null}\}] \end{array} \right\}$$

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

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

$$\hookrightarrow \langle \textit{joinpt } (\!|\textit{call, –, m, –, Simple} \times \textit{Object} \rightarrow \textit{Object}|\!) \textit{ (loc0,loc1)}, \bullet, S_2 \rangle \qquad \text{(CALL}_\text{A}\text{)}$$

$$\hookrightarrow \langle \text{under } \textit{chain} \qquad \text{(BIND)}$$
$$[\![\textit{b, locA, this.f1=arg1, Object} \times \textit{Simple} \rightarrow \textit{Object, Simple} \times \textit{Object} \rightarrow \textit{Object}]\!],$$
$$(\!|\textit{call, –, m, –, Simple} \times \textit{Object} \rightarrow \textit{Object}|\!) \textit{ (loc0,loc1)}, J_1, S_2 \rangle$$

$$\text{where } \; b = \langle -, \text{s}, \text{arg1} \rangle$$
$$J_1 = (\!|\text{call, –, m, –, –, Simple} \times \text{Object} \rightarrow \text{Object}|\!)$$

$$\hookrightarrow \langle \text{under under } \textit{locA.f1=loc1}, J_2, S_2 \rangle \qquad \text{(ADVISE)}$$

$$\text{where } J_2 = (\!|\text{this, locA, –, –, –}|\!) + J_1$$

$$\hookrightarrow \ldots$$

I omit the remaining steps of the evaluation because similar steps have been shown already.

$$
\begin{aligned}
&\mathit{matchPCD}(\llparenthesis \text{call, } -\text{, m, } -\text{, } -\text{, Simple} \times \text{Object} \to \text{Object} \rrparenthesis,\\
&\qquad\qquad \text{call(Object m(..))} \;\&\&\; \text{args(Object arg1)} \;\&\&\; \text{target(Simple s)}, S_2)\\
&= \;\mathit{matchPCD}(\llparenthesis \text{call, } -\text{, m, } -\text{, } -\text{, Simple} \times \text{Object} \to \text{Object} \rrparenthesis, \text{call(Object m(..))}, S_2)\\
&\quad \wedge \mathit{matchPCD}(\llparenthesis \text{call, } -\text{, m, } -\text{, } -\text{, Simple} \times \text{Object} \to \text{Object} \rrparenthesis, \text{args(Object arg1)}, S_2)\\
&\quad \wedge \mathit{matchPCD}(\llparenthesis \text{call, } -\text{, m, } -\text{, } -\text{, Simple} \times \text{Object} \to \text{Object} \rrparenthesis, \text{target(Simple s)}, S_2)\\
&= \langle -, - \rangle \sqcup \langle -, -, \text{arg1} \rangle \sqcup \langle -, \text{s} \rangle\\
&= \langle -, -, \text{arg1} \rangle \sqcup \langle -, \text{s} \rangle\\
&= \langle -, \text{s}, \text{arg1} \rangle
\end{aligned}
$$

Figure 3.17 Sample Derivation of Pointcut Descriptor Matching

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. I give the full program listing in Figure 3.18 on the facing 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:

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

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

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

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

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

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

$$
\begin{aligned}
&\text{under locA.f1 =}\\
&\qquad \text{chain } [\![\langle -\text{,s2,arg2}\rangle, \text{ locA, this.f2=s2.proceed(arg2)}, \tau, \tau 2]\!], \llparenthesis \text{call, } -\text{, m, } -\text{, } \tau 2 \rrparenthesis \;\; (\text{loc0, loc1})
\end{aligned}
$$

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

$$
\begin{aligned}
&\text{under locA.f1 =}\\
&\qquad (\text{under locA.f2 = chain } \bullet, \llparenthesis \text{call, } -\text{, m, } -\text{, } \tau 2 \rrparenthesis \;\; (\text{loc0, loc1}))
\end{aligned}
$$

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

```
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 3.18 Sample Program Showing Advice Chaining

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 My final example illustrates the differences between parameter binding for this and target pointcut descriptors in MiniMAO_1. Recall that my 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, my 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 3.19 on the next page gives the sample program.

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

$\langle \textit{new Super()}.\text{run}(), \bullet, S_0 \rangle$
$\hookrightarrow \langle \textit{loc0.run()}, \bullet, S_1 \rangle$ (NEW)

```
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 3.19 Sample Program Contrasting this vs. target Binding and call vs. execution Advice

$\hookrightarrow \langle \mathit{joinpt}\ (\!|\mathit{call},-,\mathit{run},-,\tau 0|\!)\ (\mathit{loc0}), \bullet, S_1 \rangle$ (CALL$_A$)

where $\tau 0 = \mathsf{Super}\to\mathsf{Object}$

$\hookrightarrow \langle \mathsf{under}\ \mathit{chain}\ \bullet,\ (\!|\mathit{call},-,\mathit{run},-,\tau 0|\!)\ (\mathit{loc0}), J_0, S_1 \rangle$ (BIND)

where $J_0 = (\!|\mathsf{call},-,\mathsf{run},-,\tau 0|\!)$

$\hookrightarrow \langle \mathsf{under}\ (\mathit{fun\ run}\langle \mathit{this}\rangle.\mathit{this.m(new\ Super()):}\tau 0\ (\mathit{loc0})), J_0, S_1 \rangle$ (CALL$_B$)

$\hookrightarrow \langle \mathsf{under}\ \mathit{joinpt}\ (\!|\mathit{exec,loc0,run,fun\ run}\langle \mathit{this}\rangle\mathit{.this.m(new\ Super()):}\tau 0,\tau 0|\!)\ (\mathit{loc0}), J_0, S_1 \rangle$ (EXEC$_A$)

$\hookrightarrow \langle \mathsf{under}^2\ \mathit{chain}\ \bullet,\ (\!|\mathit{exec,loc0,run,fun\ run}\langle \mathit{this}\rangle\mathit{.this.m(new\ Super()):}\tau 0,\tau 0|\!)\ (\mathit{loc0}), J_1, S_1 \rangle$ (BIND)

where $J_1 = (\!|\mathsf{exec,loc0,run,fun\ run}\langle\mathsf{this}\rangle\mathsf{.this.m(new\ Super()):}\tau 0,\tau 0|\!) + J_0$

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

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

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

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

where $\tau 1 = \mathsf{Super}\times\mathsf{Super}\to\mathsf{Object}$

$\hookrightarrow \langle \mathsf{under}^4$ (BIND)

$\mathit{chain}\ [\![\langle \mathit{caller}\mapsto \mathit{loc0,callee,arg}\rangle, \mathit{locA}, (\mathit{caller;callee;new\ Sub().proceed(arg)}), \tau 2, \tau 1]\!]$

$(\!|\mathit{call},-,m,-,\tau 1|\!)\ (\mathit{loc0,loc1}), J_3, S_2 \rangle$

where $\tau 2 = \mathsf{Super}\times\mathsf{Super}\times\mathsf{Super}\to\mathsf{Object}$

$J_3 = (\!|\mathsf{call},-,\mathsf{m},-,\tau 1|\!) + J_2$

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.

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

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

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.

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

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

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

where $\tau 3 = \mathsf{Sub}\times\mathsf{Super}\to\mathsf{Object}$

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

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

$\hookrightarrow \langle \mathsf{under}^6$

$\mathit{chain}\ [\![\langle \mathit{caller}\mapsto \mathit{loc2,callee,arg}\rangle, \mathit{locA}, (\mathit{caller;\ callee;\ new\ SubSub().proceed(arg)}), \tau 4, \tau 3]\!],$

$(\!|\mathit{exec,loc2,m,fun\ m}\langle \mathit{this,arg}\rangle\mathit{.(arg;this):}\tau 3,\tau 3|\!)\ (\mathit{loc2,\ loc1}), J_5, S_3 \rangle$

(BIND)

where $\tau 4 = \mathsf{Super}\times\mathsf{Sub}\times\mathsf{Super}\to\mathsf{Object}$

$J_5 = (\!|\mathsf{exec,\ loc2,\ m,\ fun\ m}\langle\mathsf{this,arg}\rangle\mathsf{.(arg;this):}\tau 3,\ \tau 3|\!) + J_4$

$\hookrightarrow \langle \mathsf{under}^7$

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

(ADVISE)

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

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.

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

(SKIP×2)

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

$\hookrightarrow \langle \text{under}^8\ \textit{(loc1;loc3)}, J_7, S_4\rangle$ (EXEC_B)

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

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.

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

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

3.2.3 Static Semantics of MiniMAO$_1$

Figure 3.20 on the facing page and Figure 3.22 on page 96 give the additional rules for the static semantics of MiniMAO$_1$. All of the rules from MiniMAO$_0$ are used unchanged.

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

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

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.

3.2.3.1 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. Advice is well typed, as defined by the T-ADV rule, 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 "␣" 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, $pcd : \text{␣} \,.\, u_0 \,.\, \langle u_1, \ldots, u_p \rangle \,.\, u \,.\, V \,.\, V$, which is discussed in Section 3.2.3.2 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. This means that advice like:

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

Aspect typing:

$$\textsc{T-Asp}\ \frac{\forall i \in \{1..p\} \cdot \vdash adv_i \text{ OK in } a}{\vdash \text{aspect } a \ \{\ field_1 \ldots field_n\ adv_1 \ldots adv_p\ \}\ \text{OK}}$$

Advice typing:

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

Expression typing:

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

$$\textsc{T-Under}\ \frac{\Gamma \vdash e : t}{\Gamma \vdash \text{under } e : t}$$

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

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

Binding typing:

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

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

Figure 3.20 Additions to the Static Semantics for MiniMAO$_1$

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

Figure 3.21 Binding for Type Environments

is not well typed if A is a proper subtype of B: the proceed expression has type B, which is not a subtype of the declared return type of the advice. Wand et al. [157, §5.3] argue that this advice should be typable, but I 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.

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. My 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 3.21, 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 3.12 on page 79).

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.

3.2.3.2 Pointcut Descriptor Typing Rules

The rules for typing pointcut descriptors are shown in Figure 3.22 on page 96. These rules make use of a simple algebra over $\mathcal{T} \cup \{\bot\}$, 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, $\Gamma \vdash pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, \hat{u}'' \,.\, V_1 \,.\, V_2$, 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_1, the set of variables that would definitely be bound by the pointcut descriptor at a matched join point; and

— V_2, the set of variables that might be bound by the pointcut descriptor at a matched join point.

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).

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.

3.2.4 Meta-theory of MiniMAO$_1$

The meta-theory of MiniMAO$_1$ is essentially the same as for MiniMAO$_0$. One 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. The key technical innovation is a Binding Soundness lemma that relates the type of a pointcut description to the type of any code that it matches.

The statement of the Substitution lemma is unchanged. For clarity, I repeat it here with the updated proof.

Pointcut typing:

$$U ::= \langle t^* \rangle \mid \bot \qquad \hat{u} ::= t \mid \bot \qquad V \in \mathscr{P}(\mathcal{V})$$

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

T-CallPCD

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

T-ExecPCD

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

T-ThisPCD

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

T-TargPCD

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

T-ArgsPCD

$$\frac{\forall i \in \{1..n\} \cdot (\Gamma(\mathit{var}_i) = t_i) \qquad \forall i \in \{1..n\} \cdot \big(\forall j \in \{1..n\} \setminus \{i\} \cdot \big(\mathit{var}_i \neq \mathit{var}_j\big)\big)}{\Gamma \vdash \mathsf{args}(\ t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n\ ) : \bot.\bot.\langle t_1, \ldots, t_n \rangle.\bot.\{\mathit{var}_1, \ldots, \mathit{var}_n\}.\{\mathit{var}_1, \ldots, \mathit{var}_n\}}$$

T-UnionPCD

$$\frac{\Gamma \vdash \mathit{pcd}_1 : \hat{u}.\hat{u}'.U.\hat{u}''.V_1.V_1' \qquad \Gamma \vdash \mathit{pcd}_2 : \hat{u}.\hat{u}'.U.\hat{u}''.V_2.V_2' \qquad V = V_1 \cap V_2 \qquad V' = V_1' \cup V_2'}{\Gamma \vdash \mathit{pcd}_1 \mid\mid \mathit{pcd}_2 : \hat{u}.\hat{u}'.U.\hat{u}''.V.V'}$$

T-NegPCD

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

T-IntPCD

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

Figure 3.22 Static Semantics of Pointcuts in MiniMAO$_1$

Lemma 3.10 (Substitution). *If* $\Gamma, \mathit{var}_1 : t_1, \ldots, \mathit{var}_n : t_n \vdash e : t$ *and* $\forall i \in \{1..n\} \cdot \Gamma \vdash e_i : s_i$ *where* $s_i \preccurlyeq t_i$ *then* $\Gamma \vdash e\{\!|e_1 / \mathit{var}_1, \ldots, e_n / \mathit{var}_n|\!\} : s$ *for some* $s \preccurlyeq t$.

Proof. Let $\Gamma' = \Gamma, \mathit{var}_1 : t_1, \ldots, \mathit{var}_n : t_n$ and let $\{\!|\bar{e} / \overline{\mathit{var}}|\!\}$ represent $\{\!|e_1 / \mathit{var}_1, \ldots, e_n / \mathit{var}_n|\!\}$. 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, $e = \mathit{var}$, and there are two subcases. If $\mathit{var} \notin \{\mathit{var}_1, \ldots, \mathit{var}_n\}$ then $\Gamma'(\mathit{var}) = \Gamma(\mathit{var}) = t$ and the claim holds. Otherwise, without loss of generality, let $\mathit{var} = \mathit{var}_1$. Then $e\{\!|\bar{e} / \overline{\mathit{var}}|\!\} = e_1$, $\Gamma \vdash e\{\!|\bar{e} / \overline{\mathit{var}}|\!\} : s_1$, 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 3.2 (Substitution) on page 65.

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. I 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 $u \neq \text{Object}$ 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 3.2 (Substitution) on page 65, 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

$$\forall i \in \{0..p\} \cdot \left(\Gamma \vdash e''_i : u''_i, \text{ where } u''_i \preccurlyeq u'_i \preccurlyeq u_i\right).$$

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. □

The Environment Extension, Environment Contraction, and Replacement lemmas (Lemma 3.3 (Environment Extension), Lemma 3.4 (Environment Contraction), and Lemma 3.5 (Replacement), respectively) apply to MiniMAO$_1$ without change. The proof of Lemma 3.6 (Replacement with Subtyping) on page 67 needs two additional cases in the induction step to account for the new evaluation context rules. I restate it here.

Lemma 3.11 (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* $t' \preccurlyeq t$.

Proof. The proof is by induction on the size of the evaluation context $\mathbb{E}$, where the size is the number of recursive applications of the syntactic rules necessary to build $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, and $t' = u' \preccurlyeq u = t$.

For the induction step we divide the evaluation context into two parts so that $\mathbb{E}[-] = \mathbb{E}_1[\mathbb{E}_2[-]]$, where $\mathbb{E}_2$ has size one. The induction hypothesis is that the claim of the lemma holds for all evaluation contexts smaller than the one considered in the induction step, and therefore holds for $\mathbb{E}_1$. We use a case analysis on the rule used to generate $\mathbb{E}_2$. In each case we show that if $\Gamma \vdash \mathbb{E}_2[e] : s$ then $\Gamma \vdash \mathbb{E}_2[e'] : s'$ where $s' \preccurlyeq s$, and therefore the claim holds by the induction hypothesis.

Case 1—$\mathbb{E}_2 = -.m(\ e_1, \ldots, e_n\ )$. Unchanged from original proof of Lemma 3.6 (Replacement with Subtyping) on page 67.

Case 2—$\mathbb{E}_2 = v_0.m(\ v_1, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )$ *where* $p \in \{1..n\}$. Unchanged from original proof.

Case 3—$\mathbb{E}_2 = (\ l\ (\ v_0, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )\ )$ *where* $p \in \{0..n\}$. Unchanged from original proof.

Case 4—$\mathbb{E}_2 = -.f$. Unchanged from original proof.

Case 5—$\mathbb{E}_2 = cast\ s\ -$. Unchanged from original proof.

Case 6—$\mathbb{E}_2 = -;\ e''$. Unchanged from original proof.

Case 7—$\mathbb{E}_2 = (-.f = e'')$. Unchanged from original proof.

Case 8—$\mathbb{E}_2 = (v.f = -)$. Unchanged from original proof.

Case 9—$\mathbb{E}_2 = \mathit{joinpt}\ (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )$ *where* $p \in \{0..n\}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-JOIN:

$$\frac{\begin{array}{l}\forall i \in \{0..(p-1)\} \cdot \Gamma \vdash v_i : u_i \qquad \Gamma \vdash e : u \qquad \forall i \in \{(p+1)..n\} \cdot \Gamma \vdash e_i : u_i \\ \forall i \in \{0..n\} \setminus \{p\} \cdot u_i \preccurlyeq t_i \qquad u \preccurlyeq s_p \qquad (v_{opt} = loc) \implies (loc \in dom(\Gamma))\end{array}}{\Gamma \vdash \mathbb{E}_2[e] : s}$$

Now $u' \preccurlyeq u \preccurlyeq s_p$. So, also by T-JOIN, $\Gamma \vdash \mathbb{E}_2[e'] : s$.

Case 10—$\mathbb{E}_2 = \mathit{under}\ -$. The proof for this case is immediate from T-UNDER with $s = u$ and $s' = u'$.

Case 11—$\mathbb{E}_2 = \mathit{chain}\ \bar{B}, j(\ v_0, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )$ *where* $p \in \{0..n\}$. The proof is like that for Case 9, but using T-CHAIN instead of T-JOIN. The additional hypotheses of T-CHAIN, beyond those of T-JOIN, are unchanged in the type derivations for $\mathbb{E}_2[e]$ and $\mathbb{E}_2[e']$. □

Before stating the Subject Reduction theorem for MiniMAO$_1$, I give a few necessary definitions and lemmas. One simple lemma is analogous to substitution but changes the environment instead of the expression.

Lemma 3.12 (Environment Subtyping). *Let* $\Gamma, var : t \vdash e : s$. *Then for all* $t' \preccurlyeq t$, *there exists some* $s' \preccurlyeq s$ *such that,* $\Gamma, var : t' \vdash e : s'$.

Proof. Let var' be a variable reference such that $var' \notin dom(\Gamma)$, $var' \neq var$, and var' is not free in e. Then by the assumption of the lemma and Lemma 3.3 (Environment Extension) on page 66, $\Gamma, var' : t', var : t \vdash e : s$. By Lemma 3.10 (Substitution) on page 96, $\Gamma, var' : t' \vdash e\{var' / var\} : s'$ for some $s' \preccurlyeq s$. Finally, by α-converting var' to var (relying on the correspondence of α-conversion with capture avoiding substitution of one variable reference for another), we have $\Gamma, var : t' \vdash e : s'$ for some $s' \preccurlyeq s$. □

I 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 3.13 (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 3.14 (Store Validity). Given a program P, we say that a store S is *valid* if both of the following hold:

1. $\forall$aspect a { … } $\in CT \cdot (\exists loc \in \mathscr{L} \cdot S(loc) = [a.F])$
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.

Advice declaration: s around(s_1 var_1,…,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 3.23 Meta-variables Used in the Proof of Lemma 3.15

Lemma 3.15 (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. I 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

$$\frac{\begin{array}{cc} & var_1 : s_1, \ldots, var_p : s_p \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, u \,.\, V \,.\, V \\ V = \{var_1, \ldots, var_p\} & var_1 : s_1, \ldots, var_p : s_p, \text{this} : a, \text{proceed} : (u_0 \times \ldots \times u_q \to u) \vdash e : s' \qquad s' \preccurlyeq s \preccurlyeq u \end{array}}{\vdash s \text{ around}(\ s_1\ var_1, \ldots, s_p\ var_p\ ) : pcd\ \{\ e\ \} \text{ OK in } a} \tag{3.1}$$

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 3.23 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, I 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 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.

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

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

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

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

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

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

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UnionPCD, $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, u_0 \,.\, U_1 \,.\, \hat{u}'_1 \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, u_0 \,.\, U_2 \,.\, \hat{u}'_2 \,.\, V_2 \,.\, V'_2$. 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:

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

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

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

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

Subclaim-□

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

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

Proof of subclaim.

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

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

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

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

— $pcd = \mathsf{args}(\ t_1''\ var_1'', \ldots, t_w''\ var_w''\ )$. 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, $\Gamma' \vdash pcd_1 : \hat{u}_1 \centerdot \hat{u}_1' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}_1'' \centerdot V_1 \centerdot V_1'$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \centerdot \hat{u}_2' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}_2'' \centerdot V_2 \centerdot V_2'$. 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:

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

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 $\Gamma' \vdash pcd : \hat{u} \centerdot \hat{u}' \centerdot U \centerdot u \centerdot V' \centerdot V''$ (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 100. The first hypothesis of T-ADV (see (3.1) on page 100) 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:

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

We next turn to consequent 2 on page 100. 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 3.20 on page 93).

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:

$$\emptyset \vdash b \text{ OK} \tag{3.2a}$$

$$V' \subseteq var(b) \subseteq V'' \tag{3.2b}$$

$$\hat{u} = \bot \iff \alpha = - \tag{3.2c}$$

$$\hat{u}' = \bot \iff \beta_0 = - \tag{3.2d}$$

$$U = \bot \implies x = 0 \tag{3.2e}$$

$$U \neq \bot \implies x = n \tag{3.2f}$$

$$U = \bot \iff \forall i \in \{1..x\} \cdot \beta_i = - \tag{3.2g}$$

Proof of subclaim.

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

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, - \rangle \\ &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' = \emptyset \subseteq var(b) \subseteq \emptyset = V'' \\ &\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (3.2c) holds} \\ &\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{so (3.2d) holds} \\ &\quad U = \bot \text{ and } x = 0 \text{ so (3.2e) holds} \\ &\quad U = \bot \text{ so (3.2f) holds} \\ &\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (3.2g) holds} \end{aligned}$$

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

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

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle var'' \mapsto v, - \rangle \text{ for some } v \in \mathcal{V} \\ &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' = \{var''\} \subseteq var(b) \subseteq \{var''\} = V'' \\ &\quad \hat{u} \neq \bot \text{ and } \alpha \neq - \text{ so (3.2c) holds} \\ &\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{ so (3.2d) holds} \\ &\quad U = \bot \text{ and } x = 0 \text{ so (3.2e) holds} \\ &\quad U = \bot \text{ so (3.2f) holds} \\ &\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (3.2g) holds} \end{aligned}$$

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

matchPCD,

$$\begin{aligned}
\mathit{matchPCD}(J, \mathit{pcd}, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies & b = \langle -, \mathit{var}'' \rangle \\
\implies & \varnothing \vdash b \text{ OK} \\
& V' = \{\mathit{var}''\} \subseteq \mathit{var}(b) \subseteq \{\mathit{var}''\} = V'' \\
& \hat{u} = \bot \text{ and } \alpha = - \text{ so (3.2c) holds} \\
& \hat{u}' \neq \bot \text{ and } \beta_0 \neq - \text{ so (3.2d) holds} \\
& U = \bot \text{ and } x = 0 \text{ so (3.2e) holds} \\
& U = \bot \text{ so (3.2f) holds} \\
& U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (3.2g) holds}
\end{aligned}$$

— $\mathit{pcd} = \text{args}(\ t_1''\ \mathit{var}_1'', \ldots, t_w''\ \mathit{var}_w''\ )$. By T-ARGSPCD, $\Gamma' \vdash \mathit{pcd} : \bot \centerdot \bot \centerdot \langle t_1'', \ldots, t_w'' \rangle \centerdot \bot \centerdot V' \centerdot V''$ where $V' = V'' = \{\mathit{var}_1'', \ldots, \mathit{var}_w''\}$, and all var_i'' are unique. By the definition of *matchPCD*,

$$\begin{aligned}
\mathit{matchPCD}(J, \mathit{pcd}, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle \implies & b = \langle -, -, \mathit{var}_1'', \ldots, \mathit{var}_w'' \rangle \\
\implies & \varnothing \vdash b \text{ OK} \\
& V' \subseteq \mathit{var}(b) \subseteq V'' \\
& \hat{u} = \bot \text{ and } \alpha = - \text{ so (3.2c) holds} \\
& \hat{u}' = \bot \text{ and } \beta_0 = - \text{ so (3.2d) holds} \\
& U \neq \bot \text{ so (3.2e) holds} \\
& U \neq \bot \text{ and } x = w = n \text{ by Subclaim 2, so (3.2f) holds} \\
& U \neq \bot \text{ and } \exists i \in \{1..0\} \cdot \beta_i \neq - \text{ so (3.2g) holds}
\end{aligned}$$

— $\mathit{pcd} = \mathit{pcd}_1 \ ||\ \mathit{pcd}_2$. By T-UNIONPCD, let

$$\begin{aligned}
\Gamma' \vdash \mathit{pcd}_1 &: \hat{u}_1 \centerdot \hat{u}_1' \centerdot U_1 \centerdot \hat{u}_1'' \centerdot V_1 \centerdot V_1' \\
\Gamma' \vdash \mathit{pcd}_2 &: \hat{u}_2 \centerdot \hat{u}_2' \centerdot U_2 \centerdot \hat{u}_2'' \centerdot V_2 \centerdot V_2'
\end{aligned}$$

Also let $\mathit{matchPCD}(J, \mathit{pcd}_1, S) = r_1$ and $\mathit{matchPCD}(J, \mathit{pcd}_2, S) = r_2$.

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*,

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

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

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

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

$$\Gamma' \vdash pcd_1 : \hat{u}_1 \centerdot \hat{u}'_1 \centerdot U_1 \centerdot \hat{u}''_1 \centerdot V_1 \centerdot V'_1$$
$$\Gamma' \vdash pcd_2 : \hat{u}_2 \centerdot \hat{u}'_2 \centerdot U_2 \centerdot \hat{u}''_2 \centerdot V_2 \centerdot V'_2$$

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 (3.2c) holds.

– Similarly, $\hat{u}' = - \iff \beta_0 = -$, and (3.2d) 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 $\big(U = - \iff \forall i \in \{1..x\} \cdot \beta_i = -\big)$. Thus, (3.2e), (3.2f), and (3.2g) 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, (3.2a) 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 (3.2b) holds.

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

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, - \rangle \\ &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' = \emptyset \subseteq var(b) \subseteq \emptyset = V'' \\ &\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (3.2c) holds} \\ &\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{so (3.2d) holds} \\ &\quad U = \bot \text{ and } x = 0 \text{ so (3.2e) holds} \\ &\quad U = \bot \text{ so (3.2f) holds} \\ &\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (3.2g) holds} \end{aligned}$$

Subclaim-□

By T-Adv, the assumption of the subclaim holds. Therefore, consequent 2 on page 100 holds by (3.2a).

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

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s_1', \ldots, var_p : s_p' \text{ where } \forall i \in \{1..p\} \cdot s_i' \preccurlyeq s_i \tag{3.3}$$

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

$$\begin{aligned} &matchPCD(J, pcd, S) = b \neq \bot \\ &\implies \forall var \in var(b) \cdot \left(\exists i \in \{1..p\}, s'_i \in \mathcal{T} \cdot \left(var = var_i,\ typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_i) = s'_i, \text{ and } s'_i \preccurlyeq s_i\right)\right) \end{aligned}$$

Proof of subclaim. The assumption of this subclaim implies the assumption for Subclaim 4 on page 103; we will make free use of the earlier result.

— $pcd = \mathsf{call}(\ \ldots\ )$. By T-CALLPCD, $V' = V'' = \emptyset$. By (3.2b) on page 104, $matchPCD(J, pcd, S) = b \neq \bot$ 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

$$\begin{aligned} &loc_1 \in dom(S) \text{ by } J \approx S, \\ &S(loc_1) = \left[s'_1 \,.\, F\right], s'_1 \preccurlyeq s_1, \text{ by definition of } matchPCD, \text{ and} \\ &\Gamma(loc_1) = s'_1 \text{ by } \Gamma \approx S. \end{aligned}$$

Thus,

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s'_1 \text{ where } s'_1 \preccurlyeq s_1.$$

— $pcd = \mathsf{target}(\ t''\ var''\ )$. 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

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

— $pcd = \mathsf{args}(\ t''_1\ var''_1, \ldots, t''_w\ var''_w\ )$. 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,

$$\forall i \in \{1..w\} \cdot \left(\exists! j \in \{1..p\} \cdot \left(t''_i = s_j \text{ and } var''_i = var_j\right)\right) \tag{3.4}$$

The definition of *matchPCD* gives

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = \left\langle -, -, var''_1, \ldots, var''_w \right\rangle$$

where $n = w$ and $\forall i \in \{1..w\} \cdot \left(t''_i = t_i\right)$. 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

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)\left(var''_1\right) = t''_1.$$

By (3.4) on the facing page, 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 (3.1) on page 100) are:

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

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 (3.3) from page 107.

$$\begin{array}{lll} V = \{var_1, \ldots, var_p\} & & \text{by T-ADV} \\ \quad \Longrightarrow var(b) = \{var_1, \ldots, var_p\} & & \text{by (3.2b)} \\ \quad \Longrightarrow \forall i \in \{1..p\} \cdot \exists s'_i \in \mathscr{T} & & \\ \qquad \big(typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_i) = s'_i, s'_i \preccurlyeq s_i\big) & & \text{by Subclaim 5} \end{array}$$

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

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

Thus, no additional variables are bound and (3.3) on page 107 holds:

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s'_1, \ldots, var_p : s'_p \text{ where } \forall i \in \{1..p\} \cdot s'_i \preccurlyeq s_i$$

The third hypothesis of T-ADV gives

$$\begin{array}{ll} var_1 : s_1, \ldots, var_p : s_p, \text{this} : a, \text{proceed} : \tau' \vdash e : s' & \\ \quad \Longrightarrow var_1 : s'_1, \ldots, var_p : s'_p, \text{this} : a, \text{proceed} : \tau' \vdash e : s'' & \text{by Lemma 3.12} \\ \qquad \text{where } s'' \preccurlyeq s \text{ and } \forall i \in \{1..p\} \cdot s'_i \preccurlyeq s_i & \\ \quad \Longrightarrow \text{this} : a, \text{proceed} : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e : s'' & \text{by (3.3)} \\ \quad \Longrightarrow \Gamma, \text{this} : a, \text{proceed} : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash e : s'' & \end{array}$$

where the last implication is by Lemma 3.3 (Environment Extension), with appropriate α-conversion of b and e. Finally, the last hypothesis of T-ADV gives $s' \preccurlyeq s \preccurlyeq u$. By transitivity of subtyping, and $u = t$, $s'' \preccurlyeq t$. Thus the final consequent holds. □

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

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

— $\Gamma \vdash b$ *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 3.4 (Environment Contraction) on page 67.

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

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

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 and the induction hypothesis, we have:

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

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

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

Lemma 3.17 (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 $j = (\!|\text{exec}, v, m, l, \tau|\!)$ and $l = \text{fun } m\langle var_0, \ldots, var_n \rangle . e : \tau$, or else*
2. *$j = (\!|\text{call}, -, m, -, (t_0 \times \ldots \times t_n \to t)|\!)$ and $\text{methodType}(t_0, m) = t_1 \times \ldots \times t_n \to t$.*

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. The proof is extended to account for the new evaluation rules.

Theorem 3.18 (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 3.14 (Store Validity) on page 99 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 3.7 (Subject Reduction) on page 69.

Case 1—CALL$_A$. Here $e = \mathbb{E}[loc.m(\ v_1,\ldots,v_n\ )]$, $e' = \mathbb{E}[\mathsf{joinpt}\ (\!|\mathsf{call}, -, m, -, (s_0 \times \ldots \times s_n \to s)|\!)(\ loc, v_1, \ldots, v_n\ )]$ (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 now show 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]

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

Therefore, Lemma 3.5 (Replacement) on page 67 gives $\Gamma \vdash e' : t$.

Case 2—CALL$_B$. Here $e = \mathbb{E}[\mathsf{chain}\ \bullet, (\!|\mathsf{call}, -, m, -, \tau|\!)(\ loc, v_1, \ldots, v_n\ )]$, $e' = \mathbb{E}[(\ l\ (\ loc, v_1, \ldots, v_n\ )\ )]$ (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 now show that $\Gamma \vdash e' : t$. Let $e_{\mathsf{left}} = \mathsf{chain}\ \bullet, (\!|\mathsf{call}, -, m, -, \tau|\!)(\ loc, v_1, \ldots, v_n\ )$. 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 $\Gamma \vdash e_{\mathsf{left}} : s$. 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 3.17 (Join Point Abstractions) on the previous page, 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

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

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

Thus, T-EXEC gives

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

and Lemma 3.5 (Replacement) on page 67 gives $\Gamma \vdash e' : t$.

[5] I omit the v_{opt} hypothesis because "$-$" is not a location.

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 now show 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 3.5 (Replacement) on page 67 gives $\Gamma \vdash e' : t$.

Case 4—EXEC$_B$. Here

$$
\begin{aligned}
e &= \mathbb{E}[\mathsf{chain}\ \bullet, (\!|\mathsf{exec}, v, m, l, (s_0\times\ldots\times s_n\to s)|\!)(\ v_0,\ldots,v_n\ )]\\
l &= \mathsf{fun}\ m\langle var_0,\ldots,var_n\rangle.e'' : (s_0\times\ldots\times s_n\to s)\\
e' &= \mathbb{E}[\mathsf{under}\ e''\{\!|v_0/\ var_0,\ldots,v_n/\ var_n|\!\}]\\
J' &= (\!|\mathsf{this}, v_0, -, -, -|\!) + J\\
S' &= S
\end{aligned}
$$

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

We now show 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 the 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 $J' \approx S$.

To complete the case, we will next 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 3.3 (Environment Extension) on page 66 we have $\Gamma, var_0 : s_0,\ldots,var_n : s_n \vdash e'' : u$. Thus, by Lemma 3.10 (Substitution) on page 96, $\Gamma \vdash e''\{\!|v_0/\ var_0,\ldots,v_n/\ var_n|\!\} : u'$ where $u' \preccurlyeq u \preccurlyeq s$. So Lemma 3.11 (Replacement with Subtyping) on page 98 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 $e_{\text{left}} = \text{joinpt}\ (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )$. 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 with $\Gamma \vdash e_{\text{left}} : s$. 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 $e_{\text{right}} = \text{chain}\ \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s)|\!)(\ v_0, \ldots, v_n\ )$. (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 3.15 (Binding Soundness) on page 100 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 3.5 (Replacement) on page 67, $\Gamma \vdash e' : t$.

Case 6—ADVISE. Here

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

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

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

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

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

where $s \preccurlyeq t_c$. The constraints on $\bar{B}$ and j imposed by T-CHAIN satisfy the conditions of Lemma 3.16 (Advice Chaining) on page 110, so we have

$$\Gamma, \mathsf{this} : \Gamma(loc), typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \vdash \langle\!\langle e'' \rangle\!\rangle_{\bar{B}, j} : s \tag{3.5}$$

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 (3.5) expands to

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

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

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

Finally, by the hypotheses of T-CHAIN in the typing of e_{left} we have $\forall i \in \{0..n\} \cdot \left(\Gamma \vdash v_i : u_i' \text{ where } u_i' \preccurlyeq t_i\right)$. Thus, Lemma 3.10 (Substitution) gives $\Gamma \vdash e_{\mathsf{right}} : s'$ where $s' \preccurlyeq s \preccurlyeq t_c$. By T-UNDER and Lemma 3.11 (Replacement with Subtyping) on page 98, $\Gamma \vdash e' : t'$ 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 now show 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 3.5 (Replacement) on page 67, we have $\Gamma \vdash e' : t$.

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

The Progress theorem is slightly modified for MiniMAO$_1$, to include the validity of the store. Additional proof cases are added for the new and modified evaluation rules.

Theorem 3.19 (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:*

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

[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.

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

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

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 3.8 (Progress) on page 71.

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

Because $\mathit{loc}.m(\ v_1, \ldots, v_n\ )$ is well typed, we know by the hypotheses of T-CALL that $\mathit{methodType}(s, m)$ yields an n-arity method type. By the definition of *origType*, we know that $\mathit{origType}(s, m) = t_0$, where $s \preccurlyeq t_0$. By T-CLASS, T-MET, and *override*, we know that $\mathit{methodType}(t_0, m)$ also 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 3.17 (Join Point Abstractions) on page 111, there are two cases:

— $j = (\!|\mathsf{exec}, v, m, l, \tau|\!)$: By Lemma 3.17, $l = \mathsf{fun}\ m\langle \mathit{var}_0, \ldots, \mathit{var}_n\rangle.e : \tau$. 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 $v_0 \in \mathit{dom}(\Gamma)$. By $\Gamma \approx S$, $S(v_0) = \left[u_0' \centerdot F\right]$. 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 3.17, $\mathit{methodType}(t_0, m) = t_1 \times \ldots \times t_n \to t$. Also by T-CHAIN, $u_0' \preccurlyeq t_0$. By the correspondence between *methodType*'s definition and that of *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 $\mathit{methodBody}(u_0', m) = l$. 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. □

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

Theorem 3.20 (Type Safety). *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 $\langle e, \bullet, S_0\rangle \overset{*}{\hookrightarrow} \langle x, J, S\rangle$ and one of the following hold for x:*

— *$x = loc$ and $loc \in dom(S)$,*

— *$x = null$,*

— *$x = NullPointerException$, or*

— *$x = ClassCastException$*

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 3.18 (Subject Reduction) on page 111 and Theorem 3.19 (Progress) on page 115. □

3.3 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. [157] present a denotational semantics for an aspect-oriented language that includes dynamic-context pointcut descriptors. My 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. [53] present a system for reasoning about dynamic-context pointcut matching. They do not formalize advice parameter binding and do not include proceed in their language.

Jagadeesan et al. [75] present a calculus for a multithreaded, class-based, aspect-oriented language. They omit methods, using advice for all code abstraction. The lack of separate methods simplifies their semantics, but makes their calculus a poor fit for my study of reasoning in an AspectJ-like language. Also, their calculus does not include the ability of advice to change the target object of an invocation. In an unpublished paper Jagadeesan et al. [74] add a sound, static type system to their calculus. My 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 [108] 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.

Orleans [128] also presents a Scheme-based language, Fred, that includes some aspect-oriented features. Essentially Fred allows programmers to write reflective predicates which are evaluated at every method call. If true, these predicates trigger the execution of associated code. Fred does not try to model AspectJ *per se*, but only advice-like constructs. Lämmel [89] also presents a core aspect-oriented calculus that models advice execution, but not AspectJ in particular. He uses "method call interception" to trigger advice at method call sites in the operational semantics. Neither of these studies considers changing target objects or the affect of that on method dispatch. Neither study considers execution join points.

Aldrich [8] presents a system called "open modules" that includes advice and dynamic-context pointcut descriptors 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 [48] present a calculus for "harmless advice", based on an extension of the typed lambda calculus plus Abadi-Cardelli-style objects. They use a type system with "protection levels" to keep aspects from altering the data of the base program. In keeping with this non-interference property, they do not allow advice to change values when proceeding to the base program. I discuss this more in Section 4.5.

Bruns et al. [27] 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 in my 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. [156] 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 (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.

3.4 Discussion

As noted in Section 3.2.2.3, because of the lack of constructors, there is no obvious mechanism in MiniMAO$_1$ for initializing the state of the implicitly instantiated aspects.

The meta-theory for MiniMAO$_1$ only relies on having a valid store. Thus, one can reason about the language by assuming a store where aspects have already been initialized. Because of this, I choose not to complicate the calculus further by adding a mechanism for aspect initialization. For the reader's edification, I sketch here how such a mechanism might be added. The basic idea is to lazily initialize an aspect instance at the start of every advice body. A full-scale language like AspectJ has constructors, so this mechanism would not be necessary there, but could still be used.

The problem for lazy initialization in MiniMAO$_1$ is that there is no way to check whether an aspect is already initialized. Polymorphic method dispatch is the only branching mechanism in the language. An uninitialized aspect has null-valued fields, so there are no objects on which to dispatch. My proposed solution would be to add a simple if expression to the calculus for branching based on whether or not a value is null. The expression would have the form

$$\text{if } (\ e_0 == \text{null}\ )\ \{\ e_1\ \}\ \text{else}\ \{\ e_2\ \}\ .$$

A new evaluation context, evaluation rule, and typing rule would also be needed. These would be:

$$\mathbb{E} ::= \ldots \mid \text{if } (\ \mathbb{E} == \text{null}\ )\ \{\ e\ \}\ \text{else}\ \{\ e\ \}$$

$$\left\langle \mathbb{E}[\text{if } (\ v == \text{null}\ )\ \{\ e_1\ \}\ \text{else}\ \{\ e_2\ \}\ ], S, J \right\rangle \hookrightarrow \left\langle \mathbb{E}[e'], S, J \right\rangle, \text{where } e' = \begin{cases} e_1 & \text{if } v = \text{null} \\ e_2 & \text{otherwise} \end{cases}$$

$$\frac{\forall i \in \{0..2\} \cdot \Gamma \vdash e_i : s_i \qquad s_1 \preccurlyeq t \qquad s_2 \preccurlyeq t}{\Gamma \vdash \text{if } (\ e_0 == \text{null}\ )\ \{\ e_1\ \}\ \text{else}\ \{\ e_2\ \} : t}$$

The updates to the meta-theory to add this expression form would be straightforward. I leave them as an

exercise for the reader.

3.5 Conclusion

In this chapter I introduced MiniMAO$_1$, a core calculus for AspectJ. MiniMAO$_1$ faithfully explains the semantics of AspectJ's around advice at method call and execution join points. In particular, MiniMAO$_1$ is the first aspect-oriented formalism to model the possibility that advice can change the target object at a join point and affect method dispatch. MiniMAO$_1$ models the fact that in AspectJ, advice that changes the target object at a call join point may change the method dispatched to, while advice that changes the target object at an execution join point will not affect the dispatched method. 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 not used to determine the method called until step iv, a piece of advice can change the target by passing a different object in a proceed expression. Such a change affects method dispatch by potentially changing the application form created.

The application form created in step iv of the method call sequence is processed through a similar four-step sequence modeling method execution. In the fourth step of this sequence, arguments provided by the last piece of advice are substituted for formal parameters in the application form generated by the method call sequence. A new target object provided by execution advice will replace any this expressions in the application form. In this way, execution advice may change the "self" object used, but does not affect method dispatch.

This four-step sequence, used for method call and execution in MiniMAO$_1$, is a general technique. It separates advice binding and advice execution from the primitive operations in the base language. This simplifies the modeling of join points for any primitive operation.

MiniMAO$_1$ faithfully models the binding of formal parameters in advice to the target, self, and argument objects at a join point. It uses the notion of a binding term, derived from a pointcut description, to perform this binding. This modeling of binding, plus the imperative nature of the calculus, provides the foundation necessary to investigate both the power of my proposed assistant aspects and my proposed restrictions on spectator aspects.

AspectJ is not statically type safe [74]. With MiniMAO$_1$, I demonstrate that the type safety problems extend to, and are exacerbated by, the ability to change target objects in advice. To provide a solid foundation for formalizing the reasoning issues that I am concerned with, MiniMAO$_1$ changes advice matching and pointcut typing to provide static type safety. The concept of *binding soundness*, introduced here, is instrumental in proving the soundness of my static type system. MiniMAO$_1$'s sound static type system is a first for a language with such powerful around advice.

MiniMAO$_1$ uses a different semantics for advice binding than AspectJ, using exact type matching rather than subtype matching in many cases. The semantics in MiniMAO$_1$ causes pointcut descriptions to match a subset of the join points matched using the AspectJ semantics. MiniMAO$_1$'s more limited matching semantics is necessary for static type safety.

The typing of proceed and the various pointcut descriptors in MiniMAO$_1$ also differs from AspectJ. The typing of proceed expressions 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.

In the next chapter I build on MiniMAO$_1$, introducing new type system features that help to distinguish, and reason about, spectators and assistants.

CHAPTER 4. MINIMAO$_2$: PARTITIONING THE HEAP BY CROSS-CUTTING CONCERNS

In this chapter, I extend MiniMAO$_1$ with "concern domains" and read-only pointers. I call the new calculus *MiniMAO$_2$* .

Informally, concern domains represent a partitioning of the heap into sets representing orthogonal, or cross-cutting, concerns. Concern domains in MiniMAO$_2$ allow cross-cutting concerns to be represented in the type system. MiniMAO$_2$ enables efficient static detection of tangled code by lifting cross-cutting concerns from the program implementation into the type system.

A global configuration declares the concern domains that may be used to partition the heap. Thus, the programmer controls which actual concerns are expressed in the type system. The signatures of declarations in the calculus, along with object instantiation expressions, determine the actual partitioning. The type system enforces a non-interference property so that a global, signature-level search can identify all the code that might mutate a particular concern domain. By "signature-level", I mean that only method and advice headers, and not their bodies, must be considered. This global search is related to the global configuration informally argued for by Kiczales and Mezini [80]. As discussed in Chapter 2, in a language with concern maps and explicit acceptance of advice, the search scope could be further narrowed.

MiniMAO$_2$'s type system statically detects code tangling, based on a separation of concerns defined by the programmer. Aspects in MiniMAO$_2$ are assistants; they may interfere with the concerns of the base program. However, in MiniMAO$_2$ this interference must be declared in the advice, and so is easily identified. The subsequent chapter describes how we can formally define spectator aspects that are statically known to not affect the concerns of other code.

In addition to concern domains, MiniMAO$_2$ also has read-only pointers. These serve two purposes: practically, they provide a mechanism for formalizing spectators in the subsequent chapter; theoretically, they serve as a proxy for the reasoning issues involved in combining more general alias-control type systems with an aspect-oriented language.

The type system for MiniMAO$_2$ is inspired by the various ownership type systems for object-oriented languages [9, 10, 25, 35, 116, 117, 118, 121]. It is also similar to the "Harmless Advice" system described by Dantas and Walker [48], though the type system of MiniMAO$_2$ provides more fine-grained control to the programmer. For example, unlike harmless advice, aspects in MiniMAO$_2$ may be given permission to mutate data from the base program.

4.1 Intuition

Perhaps the best way to develop an intuition for MiniMAO$_2$ is to consider the store of the calculus as representing words in memory. The concern domains declared in a MiniMAO$_2$ program partition these words into sets. This partitioning is formalized in Definition 4.20 (Concern Domain) on page 187. Figure 4.1 gives a

schematic view of this intuition. The cloud-shaped outlines in the figure represent two concern domains, one for Products and one for People.

An object record in the store can be thought of as a contiguous block of words, all of which must appear in the same domain. Each of the rounded rectangles in the figure represent an object record. As in MiniMAO$_1$, each object record describes the object's type and its fields. Object types in MiniMAO$_2$ include a type like that in MiniMAO$_1$, naming the class or aspect of which the object is an instance. The object type in MiniMAO$_2$ also includes a sequence of concern domain names. The first name in this sequence is the *home* domain of the object (see Definition 4.28 (Home Domain) on page 191). The remaining names say which domains the object may access, either through method calls or field accesses. In Figure 4.1 on the facing page, the Book object is in the Products domain, and may also access the People domain. The remaining objects in the figure may not access any objects in other than their home domains.

The *loc* values stored in an object's fields can be thought of as pointers to other blocks of memory representing other object records. For example, the locT pointer for the title field in the Book object points to a StringBuffer object, also in the Products concern domain. It is also possible for a pointed-to object to be in a different domain than the field itself. The author1 field in the Book object demonstrates this. While the field itself is in the Products domain—the home domain of its object—the value stored in the author1 field points to an Author object in the People domain. Such *interdomain pointers* are only allowed to domains named in the object's type. For example, the Book object in the figure could only store pointers to objects in its home Products domain or to objects in the People domain.

As for object records, each field in MiniMAO$_2$ has a type like that in MiniMAO$_1$ and a sequence of concern domain names. The first name in this sequence specifies the domain of the object *to which the field points*. For example, the author1 field in Figure 4.1 has the type Author⟨People⟩.

I will occasionally find it useful in this chapter to refer to *public* concern domains. In MiniMAO$_2$, all concern domains are public. But in the subsequent chapter, I introduce *private* concern domains. A private concern domain can only can be named within the aspect that declares it. Such concern domains provide an encapsulated region in the heap. No pointers into a private concern domain may escape the declaring aspect. But again, all of the concern domains dealt with in the present chapter are public.

Besides concern domains, MiniMAO$_2$ also includes a notion of read-only fields. Each field in MiniMAO$_2$ may contain either a read-only pointer or a write-enabled pointer. If a field is marked as readonly, then code is not allowed to dereference the field and mutate the object to which that field points. This restriction does not prohibit the field itself from being changed to point to a different object. For example, consider the author1 field of the Book object in Figure 4.1. If this field were read-only, then code could change the value stored in the field within the Book object record. This is allowed because that would just involve changing the Book object record. However, code could not dereference the field and mutate the Author object to which the field points.

A key property enforced in MiniMAO$_2$ is that all interdomain pointers, such as those from the Book object to the Author objects in the figure, must be read-only. The static type system of MiniMAO$_2$ ensures that this property holds for any valid store occurring in the evaluation of a well-typed program. Thus, I was a bit misleading when I said above that the author1 field has the type Author⟨People⟩; it must actually have the type *readonly* Author⟨People⟩. In the subsequent chapter, this property of interdomain pointers will be central to my argument that unseen spectator aspects may not affect the behavior of an operation with respect to the writable domains of that operation.

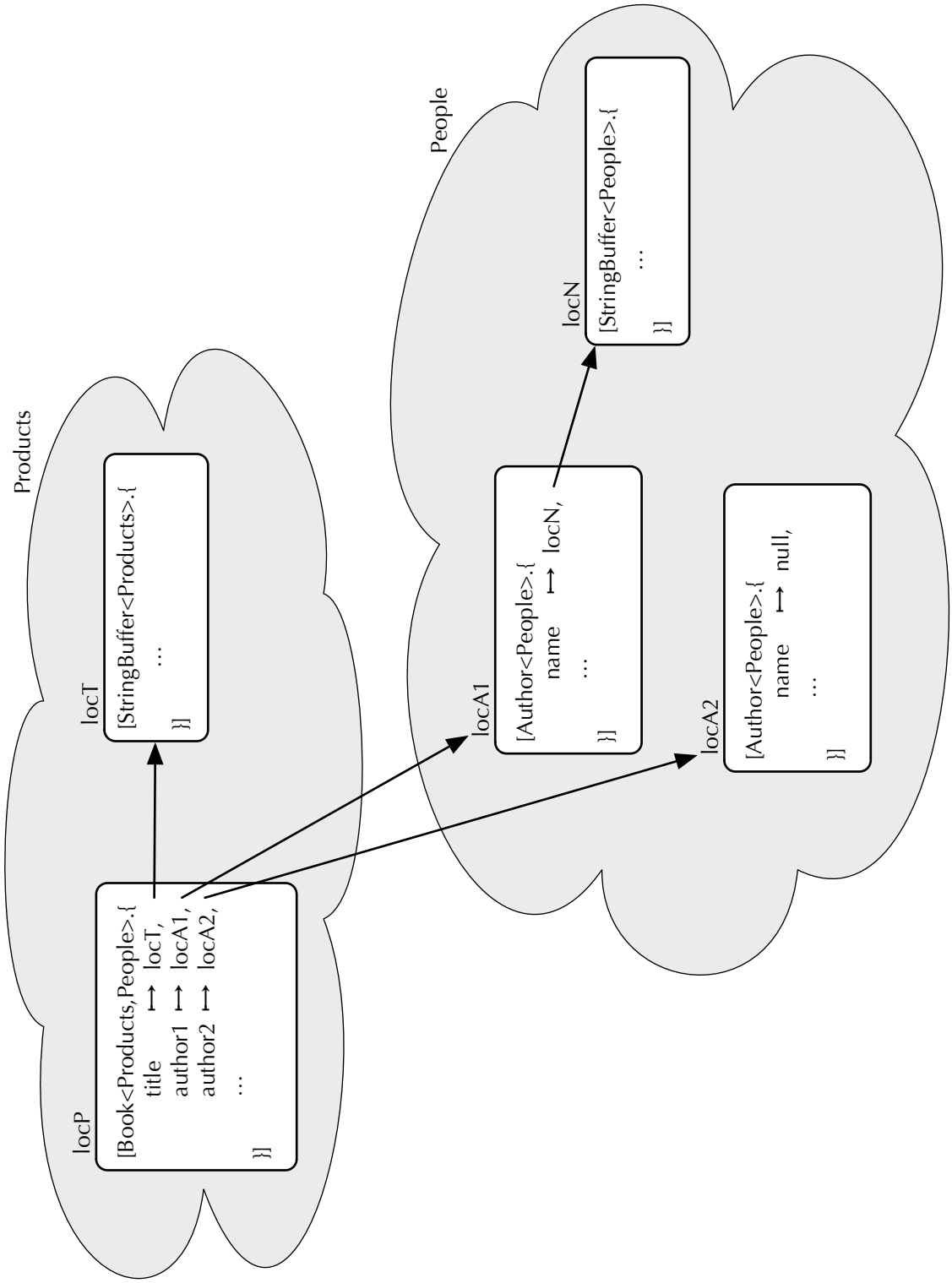

Figure 4.1 Schematic View of a Store in MiniMAO$_2$

4.2 Syntax

The syntax of MiniMAO$_2$ extends that of MiniMAO$_1$. The essential changes are the introduction of:

— *public concern domain declarations*, naming the concerns present in a program;

— *concern domain arguments on class instantiations and aspect instantiation instructions*, placing the instances in particular domains and allowing multiple instances of a single aspect;

— *refined types*, adding concern domain parameters and read-only annotations that make the cross-cutting concerns and concern interference of the program explicit;

— *effects clauses on methods and advice*, describing the domains that may be modified if the body of the method or advice is evaluated;

— *writes pointcut descriptors*, allowing advice to bind to a method based on the domains that the method might modify; and

— *domain dependency declarations in aspects*, making the interference of aspects and the base program explicit.

In this section, and the subsequent ones on semantics and meta-theory, I include the full calculus in figures and proofs, instead of just showing the changes versus MiniMAO$_1$. Hopefully, complete figures make the reader's task easier, reducing page flipping. I provide full versions of the proofs because of the subtle interaction of the new and old features. To avoid too much tedium of detail in the discussion, I will focus on the changes versus MiniMAO$_1$, and the interaction of new and old features where appropriate.

Figure 4.2 on the next page gives the syntax of MiniMAO$_2$. The following subsections describe the six essential changes. Figure 4.3 on page 126 gives a fragment of a MiniMAO$_2$ program illustrating the new syntax.[1]

4.2.1 Public Concern Domain Declarations

MiniMAO$_2$ extends program declarations, denoted by the meta-variable P, with public concern domain declarations and aspect instantiation instructions. The latter are described in Section 4.2.2. Public concern domain declarations have the simple form domain g;, where the meta-variable g ranges over the set $\mathcal{G}$ of valid concern domain names. I leave this set unspecified, but use legal Java identifiers in examples. A program may have zero or more public concern domain declarations. The sample program in Figure 4.3 declares five public concern domains, beginning on line 39: a domain for the main (driver) class of the program, a domain each for objects related to products and people, and two "logger" domains that are used for aspects.

4.2.2 Class and Aspect Instantiation

Class and aspect declarations in MiniMAO$_2$ are polymorphic with respect to concern domains.

A class declaration in MiniMAO$_2$ lists, following the class name, the concern domain variables that are *in scope* for the remainder of the declaration (see line 1). The first concern domain variable listed, home in the example, represents the home domain for instances of the class (as described in Section 4.1, and formalized in Definition 4.28 (Home Domain) on page 191). The remaining concern domain variables (what and who in the example) are used to endow instances of the class with permission to access objects in other domains. The

[1] For conciseness, I take some liberties in the sample program. In particular, I treat integers, strings, and string concatenation as built-in features of the language.

$$
\begin{aligned}
P &::= decl^*\ \{\ domain^*\ asp^*\ e\ \} \\
decl &::= \text{class } c\langle G^*\rangle \text{ extends } c\langle G^*\rangle\ \{\ field^*\ meth^*\ \} \mid \\
&\quad\ \ \text{aspect } a\langle G^*\rangle\ \{\ dep^*\ field^*\ adv^*\ \} \\
field &::= t\ f; \\
meth &::= t\ m(\ form^*\ )\ eff\ \{\ e\ \} \\
dep &::= \gamma \text{ varies with } \gamma; \\
adv &::= t\ \text{around}(\ form^*\ )\ eff\ :\ pcd\ \{\ e\ \} \\
eff &::= \text{writes } \langle\gamma^*\rangle \\
pcd &::= \text{call}(\ pat\ ) \mid \text{execution}(\ pat\ ) \mid \text{writes}(\ \gamma^*\ ) \mid \\
&\quad\ \ \text{this}(\ form\ ) \mid \text{target}(\ form\ ) \mid \text{args}(\ form^*\ ) \mid \\
&\quad\ \ pcd\ \&\&\ pcd \mid !\ pcd \mid pcd\ ||\ pcd \\
pat &::= t\ idPat(..) \\
form &::= t\ var, \text{ where } var \neq \text{this} \\
e &::= \text{new } c\langle\gamma^*\rangle() \mid var \mid \text{null} \mid e.m(\ e^*\ ) \mid \\
&\quad\ \ e.f \mid e.f = e \mid \text{cast } t\ e \mid e;\ e \mid e.\text{proceed}(\ e^*\ ) \\
t, s, u &::= \delta^*\ T\langle\gamma^*\rangle \\
\delta &::= \varepsilon \mid \text{readonly, where } \varepsilon \text{ represents the empty string} \\
T, S &::= c \mid a \\
\gamma &::= g \mid G \\
domain &::= \text{domain } g; \\
asp &::= \text{use } a\langle g^*\rangle;
\end{aligned}
$$

$G \in \mathcal{G}_{var}$, the set of concern domain variable names

$g \in \mathcal{G}$, the set of concern domain names

$c, d \in \mathcal{C}$, the set of class names

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

$f \in \mathcal{F}$, the set of field names

$m \in \mathcal{M}$, the set of method names

$var \in \{\text{this}\} \cup \mathcal{V}$, where $\mathcal{V}$ is the set of variable names

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

Figure 4.2 Syntax of MiniMAO$_2$

```
1  class Main⟨home, what, who⟩ extends Object⟨home⟩ {
2     Book<what, who> book;
3
4     readonly Object⟨home⟩ run() writes ⟨home, what, who⟩ {
5        this.book = new Book⟨what, who⟩();      this.book.init();
6        this.book.setTitle(''The Long Dark Tea Time of the Soul'');
7        ...
8     }
9  }
10 class Book⟨home,author⟩ extends Object⟨home⟩ {
11    StringBuffer⟨home⟩ title;
12    readonly Author⟨author⟩ author1;
13    readonly Author⟨author⟩ author2;
14    ...
15    readonly Object⟨home⟩ setTitle(String⟨home⟩ newTitle) writes ⟨home⟩ {
16       this.title.setLength(0);       this.title.append(newTitle)
17    }
18 }
19 class Author⟨home⟩ extends Object⟨home⟩ {
20    StringBuffer⟨home⟩ name;
21    ...
22    readonly Object⟨home⟩ setName(String⟨home⟩ newName) writes ⟨home⟩ {
23       this.name.setLength(0);     this.name.append(newName)
24    }
25 }
26 aspect Logger⟨logger, loggee⟩ {
27    logger varies with loggee;
28    StringBuffer⟨logger⟩ log;
29    readonly Object⟨loggee⟩
30       around(String⟨loggee⟩ newVal, StringBuffer⟨loggee⟩ targ)
31       writes ⟨logger, loggee⟩ :
32          call(readonly Object⟨loggee⟩ append(..)) && args(String⟨loggee⟩ newVal)
33          && target(StringBuffer⟨loggee⟩ targ) && writes(loggee) {
34       this.log.append(''Setting '' + targ + '' to '' + newVal);
35       targ.proceed(newVal)
36    }
37 }
38 {
39    domain Main;
40    domain Products; domain People;
41    domain ProductLog; domain PeopleLog;
42    use Logger⟨ProductLog, Products⟩;
43    use Logger⟨PeopleLog, People⟩;
44
45    new Main⟨Main,Products,People⟩().run();
46 }
```

Figure 4.3 Fragment of a MiniMAO$_2$ Program Illustrating New Syntax

extends clause of a class declaration specifies the mapping of the concern domain variables to those of the superclass. For simplicity, the sequence of concern domain variables for the superclass must be a prefix of the sequence for the subclass. This could be relaxed in a practical language, but the strict correspondence avoids some unnecessary complexity in the core calculus. Aspect are declared similarly to classes (see line 26).

I extend the object instantiation instruction, new, in MiniMAO$_2$ to include concern domain arguments (see line 45). The static type system ensures that new expressions within the main expression of a program only use the names of declared, public concern domains. Furthermore, new expressions within a method or advice declaration must only use concern domain variables that are in scope. When the body expression of the method or advice is evaluated, the concern domain variables will be replaced with the concern domain names used to instantiate the self object of the evaluation. For example, the new expression in line 5 uses the concern domain variables what and who, which are in scope from the declaration of Main. When the run method is called on the instance of Main created in line 45, these concern domain variables will be replaced with the concern domain names Products and People.

As in MiniMAO$_1$, new expressions are syntactically restricted to creating class instances, not aspects. So, what concern domains should the semantics use for creating aspect instances? My answer is to add *aspect instantiation instructions* to MiniMAO$_2$. These instructions, denoted *asp* in Figure 4.2 on page 125 and written use $a\langle g,\ldots\rangle$, appear in the program declaration, following the concern domain declarations and preceding the main expression of the program. The sample program has two aspect instantiation instructions, beginning on line 42. Aspect instantiation instructions are like new expressions, in that they create objects and specify the concern domains to be used. However, aspect instantiation instructions are *not* part of the expression syntax. MiniMAO$_2$ uses the aspect instantiation instructions only to generate the initial store for a program evaluation. Additional aspects cannot be instantiated during program evaluation. (This ensures that all aspects that might affect a calculation may be statically identified.)

MiniMAO$_2$ no longer restricts programmers to a single instance of each aspect. Instead, they can explicitly instantiate aspects and assign them to particular domains. Because writes pointcut descriptors in advice declarations, described below, use concern domain variables, a programmer could instantiate an aspect for monitoring changes to a particular domain. The sample program illustrates this. The program creates two instances of the Logger aspect. By virtue of substitution of concern domains names for concern domain variables, one of these instances will bind advice to StringBuffer updates in the Products domain (see the advice declaration beginning on line 29). The other will bind advice to such updates in the People domain.

Although MiniMAO$_2$ does not have a module system to provide scoping, the aspect instantiation instructions can be thought of as a degenerate form of the concern maps introduced in Chapter 2. The instructions specify all the aspects that one must consider when reasoning about the program. It would be technically straightforward, though notationally complex, to extend the operational semantics to support modules and the scoping of aspect instantiation. I say that this would be technically straightforward, because the current type system already checks each class and aspect separately, without relying on the aspect instantiation instructions. Thus, the only technical issue would be to design the operational semantics to match advice based on just the aspect instances which are applicable according to the concern maps.

4.2.3 Refined Types

As discussed in Section 4.1, MiniMAO$_2$ adds read-only status and concern domain information to types. As in MiniMAO$_1$, the meta-variables t, s, and u range over types. But in a MiniMAO$_2$ program the set of types is

$$\mathcal{T} = \{\delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle \cdot q \geq 1\},$$

where the meta-variables δ, T, and γ are such that:

— δ is either readonly or the empty string (denoted ε),

— T ranges over the set of valid class and aspect names ($\mathscr{C} \cup \mathscr{A}$), and

— γ is either g, which ranges over the set of valid concern domain names, or else G, which ranges over the set of valid concern domain variables.

The sample program includes several of the new types. For example, line 2 declares a field named book. An object stored in this field must be an instance of the Book class (see line 10). Furthermore, the concern domains of the Book instance must match the second and third concern domains of the Main instance. To be concrete, the instance of Main created in line 45 could hold in its book field an instance of Book with concern domains Products and People.

As shown in Figure 4.2 on page 125, a type may include zero or more read-only annotations, δ. I treat zero annotations as equivalent to a single ε annotation. For multiple annotations, if any one of them is readonly I treat this as equivalent to a single readonly annotation. Otherwise, I treat multiple ε annotations as a single one. Allowing multiple δ annotations on types is something of a hack. The readonly annotation is idempotent—one readonly is as good as a dozen. The hack is useful in the static semantics where I can write "readonly t" to confer read-only status to the type t regardless of whether or not it is already read-only. (See the *fieldsOf* auxiliary function, in Figure 4.9 on page 136, for an example of this.)

4.2.4 Effects Clauses

In most languages, the side effects of a method or advice body on the store can only be determined by analyzing the code of the method or advice, and that of any methods called, or advice triggered, by that code. Some languages have added support for "modifies" clauses, which describe the state that may be changed by a method [22, 97, 99, 100, 104, 118, 159].

MiniMAO$_2$ adds *effects clauses*, written writes $\langle\gamma_1,\ldots,\gamma_n\rangle$, to the declarations of methods and advice (see line 15 and line 29 of Figure 4.3 for examples). These effects clauses indicate all the concern domains that might be modified when the code of the method or advice is evaluated. The static type system of MiniMAO$_2$ ensures that no other domains may be modified at evaluation time (modulo domain dependencies, described below).

Effects clauses are written using the concern domain variables of the host class or aspect. The actual concern domains that may be modified at evaluation time are thus a function of the concern domains used to instantiate the class or aspect that is the "self" object of the method or aspect evaluation.

4.2.5 New Pointcut Descriptor

The presence of effects clauses gives another mechanism for matching advice in MiniMAO$_2$. The new writes pointcut descriptor allows advice to match any method whose effects clause lists a particular set of concern domains. Consider the writes pointcut on line 33 of the sample program. The pointcut uses the loggee concern domain variable. Based on the aspect instantiation instructions in the example, one instance of the Logger aspect will monitor matched methods that may write to the Products concern domain, while the other instance will monitor matched methods that may write to People.

For soundness of the static type system, we will see that the domains listed in the writes pointcut descriptor must exactly match the writable domains of the advised code. This restriction, and other matching restrictions

are relaxed for spectators in the subsequent chapter. These relaxations make "concern-domain generic" aspects, like Logger, more practical.

As far as I know, MiniMAO$_2$ is the first language to propose a pointcut descriptor based on the statically verified side effects of the matched join point.

4.2.6 Concern Domain Dependencies

Finally, MiniMAO$_2$ adds concern domain dependency declarations to aspects. These declarations allow an aspect to declare that one concern domain may be modified when code that is declared to modify another domain is executed. These dependency declarations allow aspects to modify other domains besides those written by some advised code. They also allow—thanks to the aspect instantiation instructions—a static analysis of what other domains might be modified.

For example, line 27 declares that the home domain of the aspect may vary with the second domain. So for the aspect instance with type Logger⟨ProductLog,Products⟩, the advice can mutate its own state (in the ProductLog concern domain) when advising a method that is supposed to mutate the Products domain. In a full language with concern maps, this dependency declaration would also be useful. It would tell any classes accepting the aspect that a method that mutates Products might also trigger aspect code that mutates ProductLog.

4.3 Semantics

4.3.1 Operational Semantics

The main changes to the operational semantics of MiniMAO$_1$ versus MiniMAO$_2$ are for tracking concern domains in the store, read-only status of values, and the writable concerns for methods and advice. Most of the changes are only to simplify the type safety proof by letting the operational semantics do some of the symbol shuffling needed for the subject reduction proof. Other changes are an important part of the evaluation, because they are used for matching the new writes pointcut descriptor.

4.3.1.1 Extensions to the Syntax

Figure 4.4 on the next page gives the extensions beyond the user syntax that MiniMAO$_2$ uses for maintaining the machine state. Again, I just discuss the differences from MiniMAO$_1$.

The value expressions in MiniMAO$_2$, *loc* and null, bear subscripts indicating whether the pointer is read-only. (The subscript on null is just for consistency.) The read-only status of a value is used in the evaluation for type casts and for type matching in advice. The status is also important for some proofs of the meta-theory (see Section 4.4.3.2 in particular). As with the types in Figure 4.2 on page 125, readonly subscripts on *loc* and null in the operational semantics may appear multiple times and are idempotent. In the sequel, when I write "*pointer*", I mean a *loc* value.

MiniMAO$_2$ adds an entirely new expression form, called a *tagged expression*, of the form $\langle e \rangle_{\delta,\hat{\gamma}}$. This expression is used in the evaluation rules for method and advice bodies. The δ subscript indicates the read-only status to be given to the value that results from evaluating e (assuming the evaluation does not diverge). The $\hat{\gamma}$ subscript gives the set of domains that are writable during the evaluation of e. The meta-variable $\hat{\gamma}$ ranges over all possible sets of concern domain names and concern domain variables. It is used solely for the subject reduction proof. MiniMAO$_2$ also uses an additional evaluation context for tagged expressions.

Syntax extensions:

$$
\begin{array}{ll}
e ::= \ldots \mid loc_{\delta^*} \mid \mathsf{null}_{\delta^*} \mid (\, l\,(\, e^*\,)\,) \mid \langle e \rangle_{\delta,\hat{\gamma}} \mid & \\
\quad \mathsf{joinpt}\ j(\, e^*\,) \mid \mathsf{under}\ e \mid \mathsf{chain}\ \bar{B}, j(\, e^*\,) & \\
l ::= \mathsf{fun}\ m\langle var^*\rangle.e : \tau \,.\, \hat{\gamma} & \\
\bar{B} ::= B + \bar{B} \mid \bullet & loc \in \mathscr{L}\text{, the set of store locations} \\
B ::= [\![b, loc, e, \hat{\gamma}, \tau, \tau]\!] & b \in \mathscr{B}\text{, the set of advice parameter bindings} \\
b ::= \langle \alpha, \beta, \beta^* \rangle & \hat{g} \in \mathscr{P}(\mathscr{G}) \\
\alpha ::= var \rightarrowtail loc_{\delta^*} \mid - & \hat{\gamma} \in \mathscr{P}(\mathscr{G} \cup \mathscr{G}_{var}) \\
\beta ::= var \mid - & \\
\tau ::= t \times \ldots \times t \rightarrow t & \\
v ::= loc_{\delta^*} \mid \mathsf{null}_{\delta^*} &
\end{array}
$$

Evaluation contexts:

$$
\begin{array}{l}
\mathbb{E} ::= - \mid \mathbb{E}.m(\, e \ldots\,) \mid v.m(\, v \ldots \mathbb{E} e \ldots\,) \mid (\, l\,(\, v \ldots \mathbb{E} e \ldots\,)\,) \mid \\
\quad \mathsf{cast}\ t\ \mathbb{E} \mid \mathbb{E}.f \mid \mathbb{E};\, e \mid \mathbb{E}.f = e \mid v.f = \mathbb{E} \mid \langle \mathbb{E} \rangle_{\delta,\hat{\gamma}} \mid \\
\quad \mathsf{joinpt}\ j(\, v \ldots \mathbb{E} e \ldots\,) \mid \mathsf{under}\ \mathbb{E} \mid \mathsf{chain}\ \bar{B}, j(\, v \ldots \mathbb{E} e \ldots\,)
\end{array}
$$

Objects:

$$
\begin{array}{l}
o ::= [t \,.\, F] \\
F : \mathscr{F} \rightarrow \mathscr{V}
\end{array}
$$

Figure 4.4 Syntax Extensions for the Operational Semantics of MiniMAO$_2$

The writable domains meta-variable, $\hat{\gamma}$, also appears in the fun form, l, and the advice body tuple, B, where it represents the writable domains of the method or advice respectively. Join point abstractions in MiniMAO$_2$ also get an optional writable domains entry (see Figure 4.5 on the next page); optional, because not every kind of join point abstraction includes this information.

Objects in the store in MiniMAO$_2$ have the same general form as in MiniMAO$_1$: $[t \,.\, F]$. But, as discussed in Section 4.2.3, the type t carries concern domain information. A property of the evaluation relation is that no type, t, in an object record is marked read-only. Read-only status is a property of a pointer, not of the object in the store. Multiple pointers to an object may exist in an evaluation, only some of which are read-only.

4.3.1.2 Evaluation in MiniMAO$_2$

Program Evaluation Program evaluation in MiniMAO$_2$ begins with the triple $\langle e, \bullet, S_0 \rangle$, where e is the main expression of the program and S_0 is a *valid* initial store containing aspect instances formed according to the aspect instantiation instructions. The notion of a valid store is formalized in Definition 4.5 (Store Validity) on page 152. I also use a global class table, CT, and a global advice table, AT, as in MiniMAO$_1$.

The advice table in MiniMAO$_2$ is constructed differently than in MiniMAO$_1$ to account for the aspect instantiation instructions. As in MiniMAO$_1$, AT consists of n-tuples recording information for each piece of

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

Figure 4.5 Join Point Stack in MiniMAO$_2$

advice, but MiniMAO$_2$ replaces the 5-tuples from MiniMAO$_1$ with 6-tuples. Figure 4.6 on the next page shows a sample aspect declaration, along with two aspect instantiation instructions and the corresponding advice table entries. As in MiniMAO$_1$, the recorded information includes the location of the advice's aspect instance in the store, the pointcut descriptor of the advice, the advice body expression, and two function types representing the formal parameter types and return type of (1) the advice and (2) any proceed expressions appearing within the advice body expression. In MiniMAO$_2$, any concern domain variables appearing in this information are replaced with the appropriate concern domain names from the aspect instantiation instruction. Thus in Figure 4.6, the first advice table entry refers to the ProductLog and Products concern domains, while the second refers to PeopleLog and People. The additional piece of information, new in MiniMAO$_2$, is the set of writable domains declared for the advice, again with concern domain variables reified.

EVALUATION RULES Figure 4.7 on page 133 gives the normal evaluation rules for MiniMAO$_2$, i.e., the rules that do not lead to an exception state. Figure 4.8 on page 134 gives the exceptional rules. Nearly every normal rule requires some modifications from its MiniMAO$_1$ version, whether to record concern domain information or to shunt about read-only annotations and writable domains sets. Only the SKIP, BIND, and UNDER rules are untouched. (The pointcut matching function, *matchPCD*—indirectly used by BIND through its call to *adviceBind*—does change, however. This change is described in Section 4.3.1.2.) Below I describe the sorts of changes needed for the other rules and note the rules where those changes are made.

Adding concern domains to objects in the store The NEW rule uses the concern domain names from an object instantiation expression when creating an object record in the store. As discussed above, object types in the store, which can only be added by the NEW rule, are never read-only. Because the object types in the store carry concern domain information, when an object type from the store is passed to an auxiliary function, the domain information is available.

As in MiniMAO$_1$, the semantics of MiniMAO$_2$ allows advice to change the target of a method, either before method lookup or else after method lookup but before method execution. In the latter case, the CALL$_B$ rule constructs the execution expression that represents the looked-up method. Examining the definition of the *methodBody* auxiliary function, in Figure 4.9 on page 136, one notes that the domain variables in the method body are replaced with the concern domain names from the target of the call operation. Does this create problems if advice later changes the target object? Thankfully, the answer is no. This is because the type rules

Example aspect to be Instantiated:

```
aspect Logger⟨logger, loggee⟩ {
   logger varies with loggee;
   StringBuffer⟨logger⟩ log;

   readonly Object⟨loggee⟩
      around(String⟨loggee⟩ newVal, StringBuffer⟨loggee⟩ targ)
      writes ⟨logger, loggee⟩ :
         call(readonly Object⟨loggee⟩ append(..)) && args(String⟨loggee⟩ newVal)
         && target(StringBuffer⟨loggee⟩ targ) && writes(loggee) {
      this.log.append(''Setting '' + targ + '' to '' + newVal);
      targ.proceed(newVal)
   }
}
```

Advice table entries:

Aspect Instantiation Instruction	Advice Table Entry, $\langle loc, pcd, e, \hat{\gamma}, \tau, \tau \rangle$
use Logger⟨ProductLog, Products⟩	⟨ loc1, call(readonly Object⟨Products⟩ append(..)) && args(String⟨Products⟩ newVal) && target(StringBuffer⟨Products⟩ targ) && writes(Products), (this.log.append(''Setting '' + targ + '' to '' + newVal); targ.proceed(newVal)), {ProductLog,Products}, String⟨Products⟩ × StringBuffer⟨Products⟩ → readonly Object⟨Products⟩, StringBuffer⟨Products⟩ → readonly Object⟨Products⟩ ⟩
use Logger⟨PeopleLog, People⟩	⟨ loc2, call(readonly Object⟨People⟩ append(..)) && args(String⟨People⟩ newVal) && target(StringBuffer⟨People⟩ targ) && writes(People), (this.log.append(''Setting '' + targ + '' to '' + newVal); targ.proceed(newVal)), {PeopleLog,People}, String⟨People⟩ × StringBuffer⟨People⟩ → readonly Object⟨People⟩, StringBuffer⟨People⟩ → readonly Object⟨People⟩ ⟩

Figure 4.6 Example of Advice Table Construction

$\langle \mathbb{E}[\text{new } c\langle g_1,\ldots,g_n\rangle()], J, S\rangle$
$\quad \hookrightarrow \langle \mathbb{E}[loc], J, S \oplus \left(loc \mapsto \left[c\langle g_1,\ldots,g_n\rangle \centerdot \{f \mapsto \text{null} \cdot f \in dom\left(fieldsOf\left(c\langle g_1,\ldots,g_n\rangle\right)\right)\}\right]\right)\rangle$ NEW
where $loc \notin dom(S)$

$\langle \mathbb{E}[loc_\delta.m(\ v_1,\ldots,v_n\ )], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{joinpt } (\!|\text{call}, -, m, -, \tau, \hat{\gamma}|\!)(\ loc_\delta, v_1,\ldots,v_n\ )], J, S\rangle$ CALL_A
where $S(loc) = [t \centerdot F]$, $methodType(t_0, m) = t_1 \times \ldots \times t_n \to t'$, $writable(t_0, m) = \hat{\gamma}$
$\quad origType(\delta\ t, m) = t_0$, and $\tau = t_0 \times \ldots \times t_n \to t'$
$\langle \mathbb{E}[\text{chain } \bullet, (\!|\text{call}, -, m, -, \tau, \hat{\gamma}|\!)(\ loc_\delta, v_1,\ldots,v_n\ )], J, S\rangle$
$\quad \hookrightarrow \langle \mathbb{E}[(\ l\ (\ loc_\delta, v_1,\ldots,v_n\ )\ )], J, S\rangle$ CALL_B
where $S(loc) = [t \centerdot F]$ and $methodBody(\delta\ t, m) = l$

$\langle \mathbb{E}[(\ l\ (\ v_0,\ldots,v_n\ )\ )], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{joinpt } (\!|\text{exec}, v_0, m, l, \tau, \hat{\gamma}|\!)(\ v_0,\ldots,v_n\ )], J, S\rangle$ EXEC_A
where $l = \text{fun } m\langle var_0,\ldots,var_n\rangle.e : \tau \centerdot \hat{\gamma}$
$\langle \mathbb{E}[\text{chain } \bullet, (\!|\text{exec}, v, m, l, \tau, \hat{\gamma}|\!)(\ v_0,\ldots,v_n\ )], J, S\rangle$
$\quad \hookrightarrow \left\langle \mathbb{E}[\text{under } \langle e\{\!|v_0/\ var_0,\ldots,v_n/\ var_n|\!\}\rangle_{\delta',\hat{\gamma}}], j + J, S\right\rangle$ EXEC_B
where $l = \text{fun } m\langle var_0,\ldots,var_n\rangle.e : \tau \centerdot \hat{\gamma}$, $readonly(\tau) = \delta'$, and
$\quad j = (\!|\text{this}, v_0, -, -, -, -|\!)$

$\langle \mathbb{E}[loc_\delta.f], J, S\rangle \hookrightarrow \langle \mathbb{E}[v_{\delta'}], J, S\rangle$ GET
where $S(loc) = \left[T\langle \gamma_1,\ldots,\gamma_n\rangle \centerdot F\right]$, $readonly\left(fieldsOf\left(\delta\ T\langle \gamma_1,\ldots,\gamma_n\rangle\right)(f)\right) = \delta'$, and $F(f) = v$
$\langle \mathbb{E}[loc_\delta.f = v], J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J, S \oplus \left(loc \mapsto \left[t \centerdot F \oplus (f \mapsto v')\right]\right)\rangle$ SET
where $S(loc) = [t \centerdot F]$ and $v' = \begin{cases} loc' & \text{if } v = loc'_{\delta'} \\ \text{null} & \text{otherwise} \end{cases}$

$\langle \mathbb{E}[\text{cast } t\ loc_\delta], J, S\rangle \hookrightarrow \langle \mathbb{E}[loc_\delta], J, S\rangle$ CAST
where $S(loc) = [s \centerdot F]$ and $\delta\ s \preccurlyeq t$
$\langle \mathbb{E}[\text{cast } \delta\ T\langle \gamma_1,\ldots,\gamma_n\rangle\ \text{null}_{\delta'}], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{null}_{\delta'}], J, S\rangle$ NCAST
if $\delta = \text{readonly}$ or $\delta' = \varepsilon$

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

$\langle \mathbb{E}[\text{joinpt } j(\ v_0,\ldots,v_n\ )], J, S\rangle \hookrightarrow \langle \mathbb{E}[\text{under chain } \bar{B}, j(\ v_0,\ldots,v_n\ )], j + J, S\rangle$ BIND
where $adviceBind(j + J, S) = \bar{B}$
$\langle \mathbb{E}[\text{chain } [\![b, loc, e, \hat{\gamma}, \tau, \_]\!] + \bar{B}, j(\ v_0,\ldots,v_n\ )], J, S\rangle$
$\quad \hookrightarrow \langle \mathbb{E}[\text{under } \langle e'\{\!|loc/\text{this}|\!\}\{\!|(v_0,\ldots,v_n)/\ b|\!\}\rangle_{\delta,\hat{\gamma}}], j' + J, S\rangle$ ADVISE
where $readonly(\tau) = \delta$, $e' = \langle\!\langle e \rangle\!\rangle_{\bar{B},j}$, and $j' = (\!|\text{this}, loc, -, -, -, -|\!)$
$\langle \mathbb{E}[\text{under } v], J, S\rangle \hookrightarrow \langle \mathbb{E}[v], J', S\rangle$ UNDER
where $J = j + J'$, for some j

$\langle \mathbb{E}[\langle v\rangle_{\delta,\hat{\gamma}}], J, S\rangle \hookrightarrow \langle \mathbb{E}[v_\delta], J, S\rangle$ TAG

Figure 4.7 Evaluation Relation for the Operational Semantics of MiniMAO$_2$

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

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

$$\langle \mathbb{E}[\mathsf{null}_\delta.f], J, S\rangle \hookrightarrow \langle \mathsf{NullPointerException}, J, S\rangle \qquad \textsc{NGet}$$

$$\langle \mathbb{E}[\mathsf{null}_\delta.f = v], J, S\rangle \hookrightarrow \langle \mathsf{NullPointerException}, J, S\rangle \qquad \textsc{NSet}$$

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

$$\langle \mathbb{E}[\mathsf{cast}\ \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle\ \mathsf{null}_{\delta'}], J, S\rangle \hookrightarrow \langle \mathsf{ClassCastException}, J, S\rangle \qquad \textsc{NXCast}$$
if $\delta' = \mathsf{readonly}$ and $\delta = \varepsilon$

Figure 4.8 Evaluation Relation for the Operational Semantics of MiniMAO$_2$ (Exceptional Rules)

only allow a subtype to be substituted for the target object, and the subtyping relation is "positionally invariant" in concern domains (see Figure 4.11 on page 138).

Tracking read-only status Several rules in MiniMAO$_1$ use the type of an object from the store during evaluation. Because read-only status, added in MiniMAO$_2$, is an attribute of a pointer, rather than the referenced object, several rules change to combine the read-only status of a location in the evaluation with the type of the referenced object from the store. For example, consider the call to the *origType* auxiliary function in the CALL$_\text{A}$ rule. Here the read-only status, δ, of the target object location, *loc*, is combined with the type, t, of the object record at $S(loc)$. By the definition of the *origType* auxiliary function, if δ here is readonly, then t_0 must also be read-only (see Figure 4.9 on page 136). So the read-only status of the target object flows through to the function type, τ, stored in the join point abstraction.

A similar change, combining a location's read-only status with a type from the store, is made in CALL$_\text{B}$, GET, CAST, and NCAST.

Also, the CAST rule, and the newly added NCAST rule, do not permit a readonly annotation to be "cast away"—any read-only annotation on the target of the cast survives in the result expression. The new NXCAST exceptional rule handles the cases not matched by the NCAST rule.

The EXEC$_\text{B}$ and ADVISE rules initiate the "execution" of a method or advice body expression. In MiniMAO$_1$, this involved instantiating the expression (by substituting actual arguments for formal parameters), pushing a join point abstraction onto the call stack, and wrapping the instantiated expression in an under expression to record that the call stack must later be popped. In MiniMAO$_2$, these rules additionally place the instantiated expression inside a tagged expression. The tagged expression records the read-only status that is to be assigned to the result of evaluating the instantiated method or advice body expression. This read-only status is extracted from the type recorded in either the method's fun term or the advice's body tuple. For a method, this status will be readonly if either the method is declared to have a read-only return type or else the target object of the call was marked readonly (by the *methodBody* auxiliary function used in the CALL$_\text{B}$ rule). Tagging a method body as read-only when the target object pointer is read-only prevents accessor methods from being used to gain a write-enabled pointer into an object's representation via a read-only target object pointer. For advice, the self object is always writable, so the read-only status of the tagged expression is readonly only if the advice is

declared to have a read-only return type.

The TAG rule takes the read-only status from the tagged expression, introduced in EXEC_B or ADVISE, and adds it to the result value. It drops the tagged expression wrapper from the result.

Similar to the EXEC_B rule, the GET rule combines the read-only status, δ, of the target object pointer, with the declared read-only status of the field type. This is done through the *fieldsOf* auxiliary function (see Figure 4.9 on the next page). If either the target pointer or the field type is readonly, then the result value is also marked as readonly.

Finally, the SET rule changes to accommodate read-only annotations. In particular, if the value to be assigned to the field is marked read-only, this marking is dropped when the store is updated. This ensures that all pointers in the store are free of annotations. No information is lost, however. The static type system, described in Section 4.3.2, ensures that the field itself is read-only in this case; otherwise the assignment would be ill typed. The type system also ensures that the target object pointer is not read-only.

Tracking writable domains Besides adding concern domain information to object records in the store, MiniMAO_2 also tracks writable domains, from effects clauses, through the CALL, EXEC, and ADVISE rules. The CALL_A rule uses the *writable* auxiliary function (see Figure 4.9 on the following page) to find the set of writable domains, $\hat{\gamma}$, for the called method, given the concern domains of the target object. This set is recorded in the call join point abstraction. In the CALL_B rule, the *methodBody* auxiliary function returns a fun term, l, that records the set of writable domains. This information flows through the EXEC_A rule and into the exec join point abstraction. The EXEC_B rule adds the set of writable domains to the tagged method body expression.

For advice, the set of writable domains comes from the advice table, via a body tuple. The ADVISE rule adds the set of writable domains for the advice to the tagged advice body expression.

Finally, the TAG rule drops the set of writable domains.

The pointcut matching function, *matchPCD*, described in Section 4.3.1.2, uses the writable domains information recorded in the join point abstractions. But otherwise the other threading of this information through the evaluation is just for the benefit of the subject reduction proof, and the meta-theory of effects clauses.

AUXILIARY FUNCTIONS Figure 4.9 on the next page and Figure 4.10 on page 137 give the auxiliary functions used in the operational semantics of MiniMAO_2.

Several of the rules use a notion of domain variable substitution, defined in the obvious way. The auxiliary functions also take advantage of the "width subtyping" of concern domain variables; that is, $q \geq r$ in a well-typed declaration like:

$$\text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ \ldots\ \}\,.$$

Some auxiliary functions are changed from MiniMAO_1 to track read-only status. In particular, if the target object of an auxiliary function is read-only, then this information is carried over to the result. This can be seen in *fieldsOf*, *methodType*, and *methodBody*.

SUBTYPING Figure 4.11 on page 138 gives the subtyping relation for MiniMAO_2. The two main changes are to handle concern domains and read-only types. Following Aldrich and Chambers [9],the subtyping relation allows things like $\mathsf{IterImpl}\langle H,E,D\rangle \preccurlyeq \mathsf{Iterator}\langle H,E\rangle$, so that downcasts may introduce (and dynamically verify) domain annotations. I say that concern domains in subtyping are *positionally invariant* and use *width subtyping*. The subtyping scheme also allows a write-enabled pointer to be used where a read-only one is

Field lookup:

$$\frac{CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ t_1\ f_1 \ldots t_n\ f_n\ meth^*\ \} \quad \mathit{fieldsOf}\big(\delta\ d\langle\gamma_1,\ldots,\gamma_r\rangle\big) = F' \qquad \forall i \in \{1..n\} \cdot s_i = \delta\ t_i\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\}}{\mathit{fieldsOf}\big(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle\big) = \big\{f_i \mapsto s_i \cdot i \in \{1..n\}\big\} \cup F'}$$

$$\frac{CT(a) = \text{aspect } a\langle G_1,\ldots,G_q\rangle \ \{\ dep^*\ t_1\ f_1;\ \ldots;\ t_n\ f_n;\ adv^*\ \} \qquad \forall i \in \{1..n\} \cdot s_i = \delta\ t_i\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\}}{\mathit{fieldsOf}\big(\delta\ a\langle\gamma_1,\ldots,\gamma_q\rangle\big) = \big\{f_i \mapsto s_i \cdot i \in \{1..n\}\big\}}$$

$$\mathit{fieldsOf}\big(\delta\ \text{Object}\langle\gamma\rangle\big) = \emptyset$$

Method type lookup:

$$\frac{\begin{array}{c}CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \exists i \in \{1..p\} \cdot meth_i = t\ m(\ t_1\ var_1,\ldots,t_n\ var_n\ )\ \mathit{eff}\ \{\ e\ \} \\ \forall i \in \{1..n\} \cdot s_i = t_i\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\} \qquad s = \delta\ t\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\}\end{array}}{\mathit{methodType}(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle, m) = s_1 \times \ldots \times s_n \to s}$$

$$\frac{\begin{array}{c}CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \nexists i \in \{1..p\} \cdot meth_i = t\ m(\ t_1\ var_1,\ldots,t_n\ var_n\ )\ \mathit{eff}\ \{\ e\ \} \qquad \mathit{methodType}(\delta\ d\langle\gamma_1,\ldots,\gamma_r\rangle, m) = \tau\end{array}}{\mathit{methodType}(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle, m) = \tau}$$

Writable domains:

$$\frac{\begin{array}{c}CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \exists i \in \{1..p\} \cdot meth_i = t\ m(\ \ldots\ ) \text{ writes } \langle\gamma'_1,\ldots,\gamma'_n\rangle\ \{\ e\ \} \qquad \forall i \in \{1..n\} \cdot \gamma''_i = \gamma'_i\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\}\end{array}}{\mathit{writable}(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle, m) = \{\gamma''_1,\ldots,\gamma''_n\}}$$

$$\frac{\begin{array}{c}CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ field^*\ meth_1 \ldots meth_p\ \} \\ \nexists i \in \{1..p\} \cdot meth_i = t\ m(\ \ldots\ )\ \mathit{eff}\ \{\ e\ \} \qquad \mathit{writable}(\delta\ d\langle\gamma_1,\ldots,\gamma_r\rangle, m) = \hat{\gamma}\end{array}}{\mathit{writable}(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle, m) = \hat{\gamma}}$$

Original declaration lookup:

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

Figure 4.9 Auxiliary Functions for Operational Semantics of MiniMAO$_2$

Method lookup:

$$\frac{\begin{array}{c} CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ \mathit{field}^*\ \mathit{meth}_1 \ldots \mathit{meth}_p\ \} \\ \exists i \in \{1..p\} \cdot \mathit{meth}_i = t\ m(\ t_1\ \mathit{var}_1,\ldots,t_n\ \mathit{var}_n\ ) \text{ writes } \langle \gamma'_1,\ldots,\gamma'_p\rangle\ \{\ e'\ \} \\ e = e'\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\} \qquad \hat{\gamma} = \left\{\gamma'_1,\ldots,\gamma'_p\right\}\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\} \\ \tau = \delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle \times s_1 \times \ldots \times s_n \to s \qquad \forall i \in \{1..n\} \cdot s_i = t_i\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\} \qquad s = \delta\ t\{\!|\gamma_1/G_1,\ldots,\gamma_q/G_q|\!\} \end{array}}{\mathit{methodBody}(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle, m) = \text{fun } m\langle\text{this}, \mathit{var}_1,\ldots,\mathit{var}_n\rangle.e : \tau \bullet \hat{\gamma}}$$

$$\frac{\begin{array}{c} CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ \mathit{field}^*\ \mathit{meth}_1 \ldots \mathit{meth}_p\ \} \\ \nexists i \in \{1..p\} \cdot \mathit{meth}_i = t\ m(\ t_1\ \mathit{var}_1,\ldots,t_n\ \mathit{var}_n\ )\ \mathit{eff}\ \{\ e\ \} \qquad \mathit{methodBody}(\delta\ d\langle\gamma_1,\ldots,\gamma_r\rangle, m) = l \end{array}}{\mathit{methodBody}(\delta\ c\langle\gamma_1,\ldots,\gamma_q\rangle, m) = l}$$

Read-only annotation:

$$\mathit{readonly}\big(\delta\ T\langle\gamma_1,\ldots,\gamma_p\rangle\big) = \delta \qquad\qquad \mathit{readonly}\big(t_0 \times \ldots \times t_n \to \delta\ T\langle\gamma_1,\ldots,\gamma_p\rangle\big) = \delta$$

Advice binding:

$$\mathit{adviceBind} : \mathit{Stack} \times \mathit{Store} \to \big\langle \mathscr{B} \times \mathscr{L} \times \mathscr{E} \times \mathscr{P}(\mathscr{G} \cup \mathscr{G}_{var}) \times (\mathscr{T}^* \to \mathscr{T}) \times (\mathscr{T}^* \to \mathscr{T}) \big\rangle$$

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

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

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:

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

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

In all other cases, binding substitution is undefined.

Figure 4.10 More Auxiliary Functions for MiniMAO$_2$ Operational Semantics

$$t \preccurlyeq t \qquad \frac{t \preccurlyeq s \qquad s \preccurlyeq u}{t \preccurlyeq u} \qquad t \preccurlyeq \text{readonly } t \qquad \frac{t \preccurlyeq s}{\text{readonly } t \preccurlyeq \text{readonly } s}$$

$$\frac{CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \ \{\ \ldots\ \}}{c\langle \gamma_1,\ldots,\gamma_q\rangle \preccurlyeq d\langle \gamma_1,\ldots,\gamma_r\rangle} \qquad \frac{CT(a) = \text{aspect } a\langle G_1,\ldots,G_q\rangle \ \{\ \ldots\ \}}{a\langle \gamma_1,\ldots,\gamma_q\rangle \preccurlyeq \text{Object}\langle \gamma_1\rangle}$$

Figure 4.11 Subtyping in MiniMAO$_2$

expected, but not vice versa. Formally, $\big(u \preccurlyeq t \text{ and } \mathit{readonly}(u) = \text{readonly}\big) \implies \big(\mathit{readonly}(t) = \text{readonly}\big)$. I make liberal use of this property in the proofs of the meta-theory in Section 4.4.

POINTCUT MATCHING Figure 4.12 on the facing page gives the definition of the pointcut matching function in MiniMAO$_2$. MiniMAO$_2$ makes just three changes to the function. (1) The cases carried over from MiniMAO$_1$ account for, but ignore, the writable concern domains information in the join point abstraction. (2) The this rule gives the bound value, v, a read-only annotation if the formal parameter type is read-only but the bound value is not. (This change is actually immaterial for MiniMAO$_2$ but increases the consistency between it and MiniMAO$_3$ in the subsequent chapter.) (3) A new case handles the new writes pointcut descriptor. For static type safety, the case requires exact matching between the set of writable domains in the join point abstraction and the set in the pointcut descriptor. Since the writes pointcut descriptor does not carry any parameter binding information, a match is signaled by the empty binding, $\langle -, - \rangle$.

The algebra of bindings for MiniMAO$_2$ is like that for MiniMAO$_1$, except for the shunting about of read-only subscripts on locations. Figure 4.13 on page 140 shows this.

4.3.2 Static Semantics

Although the changes to the operational semantics from MiniMAO$_1$ to MiniMAO$_2$ are substantial, they pale in comparison to the changes in the static semantics. The operational semantics must handle the new concern domain and read-only information to describe the new writes pointcut descriptor, and to help with the proofs of the meta-theory. The static semantics must do this, plus enforce the desired properties of concern domains, effects clauses, and read-only annotations. It does this while maintaining separate static typechecking of class and aspect declarations.

I describe the static semantics by first giving some background on a couple of notions used throughout the discussion. I follow that with a description of the specific changes to the declaration, expression, and pointcut typing rules.

4.3.2.1 Dependency Tables and Writable Concern Domains

DEPENDENCY TABLES The static semantics must track the writable concern domains for each method and piece of advice. It also must account for the "varies with" dependency declarations in aspects, which extend the set of writable concern domains. To this end, the static semantics for MiniMAO$_2$ uses *dependency tables*, denoted *DT*, that record the information from a program's domain dependency declarations, reified according to the aspect instantiation instructions. Dependency tables are used in the meta-theory for reasoning about the effects of aspects on a program. Because this reasoning is based on the aspect instantiation instructions,

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

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

$$matchPCD(\langle\!| \_, \_, \_, \_, \_, \hat{\gamma} |\!\rangle + J, \mathsf{writes}(\ \hat{\gamma}'\ ), S) = \begin{cases} \langle -, - \rangle & \text{if } \hat{\gamma} = \hat{\gamma}' \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD(\langle\!| \_, v, \_, \_, \_, \_ |\!\rangle + J, \mathsf{this}(\ t\ var\ ), S)$$
$$= \begin{cases} \langle var \mapsto v_{\delta'}, - \rangle & \text{if } v = loc_\delta,\ S(loc) = [s \centerdot F], \text{ and } \delta\ s \preccurlyeq t \quad (\text{where } readonly(t) = \delta') \\ \bot & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

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

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

Figure 4.12 Pointcut Descriptor Matching for MiniMAO$_2$

Boolean algebra of bindings (adapted from Wand et al. [157]):

$$\mathscr{B}_\perp = \mathscr{B} \cup \{\perp\} \qquad b \in \mathscr{B} \qquad r \in \mathscr{B}_\perp \qquad b \vee r = b \qquad \perp \vee r = r \qquad \perp \wedge r = \perp \qquad b \wedge \perp = \perp \qquad b \wedge b' = b \sqcup b'$$
$$\neg\perp = \langle -, - \rangle \qquad\qquad \neg b = \perp$$

Join of bindings:

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

$$(var \mapsto loc_\delta) \sqcup \left(var' \mapsto loc'_{\delta'}\right) = var \mapsto loc_\delta \qquad\qquad (var \mapsto loc_\delta) \sqcup - = var \mapsto loc_\delta$$

$$- \sqcup \left(var' \mapsto loc'_{\delta'}\right) = var' \mapsto loc'_{\delta'} \qquad var \sqcup var' = var \qquad var \sqcup - = var \qquad - \sqcup var' = var' \qquad - \sqcup - = -$$

Figure 4.13 Bindings in MiniMAO$_2$

which are like a degenerate form of concern map (as discussed in Chapter 2), the reasoning is analogous to the sort a programmer would have to do in a language with explicitly accepted assistants.

A dependency table is a reflexive, transitive relation on concern domain names and variables. It has the type $DT : (\mathscr{G} \cup \mathscr{G}_{var}) \to (\mathscr{G} \cup \mathscr{G}_{var})$. Intuitively, for any pair of concern domain names $(g, g') \in DT$, code that has permission to mutate g may also trigger mutation of g'. The "varies with" dependency declarations convey these permissions.

The definition of the evaluation dependency table below includes an auxiliary function, *depTable*. The function creates a reflexive, transitive relation such that for any pair (γ, γ') in the relation, code which has permission to mutate concern domain γ may also mutate γ'. The elements in the pair are in "reverse order": a dependency declaration γ' varies with γ induces a pair (γ, γ') in the relation. I believe that the dependency declaration ordering is more human-readable, while the ordering in the relation is more natural for the formalism.

The dependency table used in a program evaluation is constructed as follows:

Definition 4.1 (Evaluation Dependency Table). Let P be a well-typed program with public concern domains $\hat{g}$ and aspect instantiation instructions $asp_1, \ldots, asp_n$.

For each $i \in \{1..n\}$, let $asp_i = \text{use } a\langle g_1, \ldots, g_q \rangle$, with

$$CT(a) = \text{aspect } a\langle G_1, \ldots, G_q \rangle \ \{\ dep_1;\ \ldots;\ dep_p;\ field^*\ meth^*\ \}\ ,$$

and construct a set representing all the public concern domain dependencies for the aspect instance:

$$\widehat{dep}_i = \left\{ dep_1, \ldots, dep_p \right\} \{\!| g_1 / G_1, \ldots, g_q / G_q |\!\}.$$

With $\widehat{dep}_1, \ldots, \widehat{dep}_n$ constructed in this manner, the *evaluation dependency table* for P is:

$$depTable\left(\hat{g}, \left(\widehat{dep}_1 \cup \ldots \cup \widehat{dep}_n\right)\right),$$

where

$depTable(\hat{\gamma}, \widehat{dep})$ is the reflexive, transitive closure of:

$$\left(\bigcup_{\gamma \in \hat{\gamma}} \{(\gamma, \gamma)\}\right) \cup \left\{(\gamma, \gamma') \cdot \left(\exists dep \in \widehat{dep} \cdot dep = \gamma' \text{ varies with } \gamma\right)\right\}.$$

The evaluation dependency table defined above is a whole-program property (though the search depth is quite shallow, just aspect instantiation instructions and dependency declarations in aspects). In the static semantics the whole-program evaluation dependency table is not required. Instead, static typechecking uses a different dependency table for each top-level declaration. This smaller dependency table is constructed from the concern domain variables and dependency declarations in the class or aspect. The T-PROG, T-MET, and T-ASP rules, described below, each construct a small dependency table for static typechecking. The use of these smaller dependency tables corresponds to separate typechecking of classes and aspects, without the global configuration information provided by the aspect instantiation instructions.

EXPRESSION TYPING JUDGMENT The typing judgment for an expression in MiniMAO$_2$ is:

$$\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : u,$$

where DT is a dependency table, $\hat{\gamma}$ is the set of writable concern domains for e, and the type of the type environment is

$$\Gamma : (\mathcal{V} \cup \mathcal{L} \cup \mathcal{G} \cup \mathcal{G}_{var} \cup \{\text{this}, \text{proceed}\}) \rightarrow (\mathcal{T} \cup (\mathcal{T}^* \rightarrow \mathcal{T}) \cup \{\text{domain}\}).$$

Two invariants on any type environment, Γ, that are not shown in this "simple" typing are that (1) only unsubscripted locations appear in its domain and (2) for any location, loc, in the domain of Γ, $\Gamma(loc) = T\langle \ldots \rangle$. That is, the type to which Γ maps loc does not bear a read-only annotation. These invariants reflect the fact that locations in the type environment are just used to model the store in the subject reduction proofs, and locations and objects in the store do not bear read-only annotations. On the other hand, a type environment may map variable names to read-only types, modeling formal parameter declarations.

The type environment, Γ, may include elements from $\mathcal{G}$ and $\mathcal{G}_{var}$ in its domain, and the special type domain in its range. A type environment element g : domain indicates that the concern domain name g is in scope. A type environment element G : domain indicates that concern domain variable G is in scope.

The dependency table, DT, in the expression typing judgment might more conventionally be written on the left-hand side of the turnstile. I choose to part with convention and use a subscripted turnstile to more easily omit the dependency table from the notation when it is clear from context. Furthermore, unlike the other terms in the typing judgment, the dependency table is a constant throughout any given expression type derivation.

4.3.2.2 Declaration Typing

The declaration typing rules appear in Figure 4.14 on the following page. The following describes the changes to each rule versus MiniMAO$_1$.

T-PROG

$$\frac{\begin{array}{c}\forall i \in \{1..n\} \cdot \vdash decl_i \text{ OK} \\ \forall i \in \{1..r\} \cdot \exists j \in \{1..n\} \cdot decl_j = \text{aspect } a_i\langle G_1,\ldots,G_{q_i}\rangle \{ \ldots \} \qquad \forall i \in \{1..r\} \cdot \left(\forall j \in \{1..q_i\} \cdot g_{i,j} \in \{g_1,\ldots,g_p\}\right) \\ g_1 : \text{domain},\ldots,g_p : \text{domain} \,.\, \{g_1,\ldots,g_p\} \vdash_{\overline{DT}} e : t \qquad DT = depTable(\{g_1,\ldots,g_p\},\emptyset)\end{array}}{\vdash decl_1 \ldots decl_n \{ \text{ domain } g_1;\ \ldots;\ \text{domain } g_p;\ \text{use } a_1\langle g_{1,1},\ldots,g_{1,q_1}\rangle;\ \ldots;\ \text{use } a_r\langle g_{r,1},\ldots,g_{1,q_r}\rangle;\ e \} \text{ OK}}$$

T-CLASS

$$\frac{\begin{array}{c}\forall i \in \{1..n\} \cdot f_i \notin dom\left(fieldsOf\left(d\langle G_1,\ldots,G_r\rangle\right)\right) \qquad \forall i \in \{1..n\} \cdot \{G_1\} \vdash t_i \text{ OK in } c\langle G_1,\ldots,G_q\rangle \\ isClass(d\langle G_1,\ldots,G_r\rangle) \qquad \forall j \in \{1..p\} \cdot \vdash meth_j \text{ OK in } c\langle G_1,\ldots,G_q\rangle \qquad q \geq r \geq 1\end{array}}{\vdash \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle \{ \ t_1\ f_1;\ \ldots;\ t_n\ f_n;\ meth_1 \ldots meth_p \ \} \text{ OK}}$$

T-MET

$$\frac{\begin{array}{c}var_1 : t_1,\ldots,var_n : t_n, \text{this} : \delta\ c\langle G_1,\ldots,G_q\rangle, G_1 : \text{domain},\ldots,G_q : \text{domain} \,.\, \{\gamma_1,\ldots,\gamma_p\} \vdash_{\overline{DT}} e : u \\ \left(G_1 \notin \{\gamma_1,\ldots,\gamma_p\}\right) \Longrightarrow \left(\delta = \text{readonly}\right) \qquad \left(G_1 \in \{\gamma_1,\ldots,\gamma_p\}\right) \Longrightarrow (\delta = \varepsilon) \\ DT = depTable(\{G_1,\ldots,G_q\},\emptyset) \qquad u \preccurlyeq t \qquad CT(c) = \text{class } c\langle G'_1,\ldots,G'_q\rangle \text{ extends } d\langle G'_1,\ldots,G'_r\rangle \{ \ldots \} \\ override(m, d\langle G_1,\ldots,G_r\rangle, t_1 \times \ldots \times t_n \to t, \{\gamma_1,\ldots,\gamma_p\}) \qquad \{G_1,\ldots,G_q\} \vdash t \text{ OK in } c\langle G_1,\ldots,G_q\rangle \\ \forall i \in \{1..p\} \cdot \gamma_i \in \{G_1,\ldots,G_q\} \qquad \forall i \in \{1..n\} \cdot \{\gamma_1,\ldots,\gamma_p\} \vdash t_i \text{ OK in } c\langle G_1,\ldots,G_q\rangle\end{array}}{\vdash t\ m(\ t_1\ var_1,\ldots,\ t_n\ var_n\ ) \text{ writes } \langle\gamma_1,\ldots,\gamma_p\rangle \{ \ e \ \} \text{ OK in } c\langle G_1,\ldots,G_q\rangle}$$

T-ASP

$$\frac{\begin{array}{c}\forall i \in \{1..r\} \cdot \vdash dep_i \text{ OK in } a\langle G_1,\ldots,G_q\rangle \qquad DT = depTable\left(\{G_1,\ldots,G_q\},\{dep_1,\ldots,dep_r\}\right) \\ \forall i \in \{1..p\} \cdot DT \vdash adv_i \text{ OK in } a\langle G_1,\ldots,G_q\rangle \qquad q \geq 1 \qquad \forall i \in \{1..n\} \cdot \{G_1\} \vdash t_i \text{ OK in } a\langle G_1,\ldots,G_q\rangle\end{array}}{\vdash \text{aspect } a\langle G_1,\ldots,G_q\rangle \{ \ dep_1,\ldots,dep_r\ t_1\ f_1;\ \ldots;\ t_n\ f_n;\ adv_1 \ldots adv_p \ \} \text{ OK}}$$

T-DEP

$$\frac{\gamma_1 \in \{G_1,\ldots,G_q\} \qquad \gamma_2 \in \{G_1,\ldots,G_q\}}{\vdash \gamma_1 \text{ varies with } \gamma_2 \text{ OK in } a\langle G_1,\ldots,G_q\rangle}$$

T-ADV

$$\frac{\begin{array}{c}\Gamma \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1,\ldots,u_p\rangle \,.\, u \,.\, \hat{\gamma} \,.\, V \,.\, V \qquad V = \{var_1,\ldots,var_n\} \qquad \hat{\gamma} \subseteq \{\gamma_1,\ldots,\gamma_r\} \\ \{\gamma_1,\ldots,\gamma_r\} \subseteq depClose_{DT}(\hat{\gamma}) \qquad \Gamma, \text{this} : a\langle G_1,\ldots,G_q\rangle, \text{proceed} : (u_0 \times \ldots \times u_p \to u) \,.\, \{\gamma_1,\ldots,\gamma_r\} \vdash_{\overline{DT}} e : s \\ s \preccurlyeq t \preccurlyeq u \qquad \{G_1,\ldots,G_q\} \vdash t \text{ OK in } a\langle G_1,\ldots,G_q\rangle \qquad \forall i \in \{1..r\} \cdot \gamma_i \in \{G_1,\ldots,G_q\} \\ \forall i \in \{1..n\} \cdot \{\gamma_1,\ldots,\gamma_r\} \vdash t_i \text{ OK in } a\langle G_1,\ldots,G_q\rangle \qquad \Gamma = var_1 : t_1,\ldots,var_n : t_n, G_1 : \text{domain},\ldots,G_q : \text{domain}\end{array}}{DT \vdash t\ \text{around}(\ t_1\ var_1,\ldots,t_n\ var_n\ ) \text{ writes } \langle\gamma_1,\ldots,\gamma_r\rangle : pcd \{ \ e \ \} \text{ OK in } a\langle G_1,\ldots,G_q\rangle}$$

Figure 4.14 Static Semantics of Declarations in MiniMAO$_2$

PROGRAM TYPING The T-PROG rule contains new hypotheses to check the aspect instantiation instructions. In particular, the concern domain names used in the instructions must be declared in the program, and the number of names given for each instruction must match the number of concern domain variables for the corresponding aspect declaration (which must also exist).

In MiniMAO$_1$, the main expression of a program is typechecked in the empty environment. In MiniMAO$_2$, the environment records that all declared public concern domain names are in scope. T-PROG allows all public concern domains to be mutated by the main expression; that is, the set of writable domains is all public concern domains. For typechecking the main expression, T-PROG uses a dependency table that is just the reflexive relation on the set of all public concern domains. Because all public domains are writable, no other pairs are needed in the dependency table.

CLASS TYPING The T-CLASS rule threads the concern domain variables of the class declaration through the field-lookup (*fieldsOf*) and class-table-checking (*isClass*) auxiliary functions (see Figure 4.9 on page 136 and Figure 4.15 on the following page, respectively). The concern domain variables are also used in typechecking each method (see the description of T-MET below) and each field type.

The checks on field types are new in MiniMAO$_2$. Figure 4.16 on the next page gives the rules for checking the validity of a type given the concern domain variables in scope, $G_1,\ldots,G_q$, and the set of writable concern domains for the context in which the type appears, $\gamma'_1,\ldots,\gamma'_r$.

The T-TYPE rule (in Figure 4.16) says that a write-enabled type is only valid if the home domain of the type, γ_1, is in the set of writable domains. Furthermore, all of the concern domain variables used in the type must be in scope. The T-TYPERO rule removes the restriction on the type's home domain when the type is read-only.

T-CLASS checks type validity for the fields of a class, considering only the home domain of the class G_1 to be writable. Through this mechanism, the T-CLASS rule prevents the capture of write-enabled, interdomain pointers by requiring that fields containing interdomain pointers be read-only.

Finally, T-CLASS requires that the declared class have at least as many concern domain variables as the class that it extends.

METHOD TYPING Of the declaration typing rules, T-MET and T-ADV (described below) change the most from MiniMAO$_1$.

The body expression, e, of a method is checked in an environment where all the concern domain variables, $G_1,\ldots,G_q$, of the surrounding class are in scope. The environment also sets the read-only status, δ, of the special this variable based on whether or not the method's effects clause allows the home domain of the host class to be mutated (see the second and third hypotheses of T-MET). The last hypothesis of T-MET verifies that any formal parameters that point to non-writable domains are read-only; analogous to my treatment of the this variable.

The set of writable concern domains used for typechecking the method body comes directly from the effects clause of the method. Another hypothesis ensures that every concern domain variable in the effects clause is in scope. The *override* auxiliary function (see Figure 4.15 on the following page) is extended to ensure that, should the method override another method, then their effects clauses match.[2] (The T-MET hypothesis

[2]For behavioral subtyping it would be sufficient to require that the set of writable domains contains no additional elements; that is, overriding methods could safely write to a subset of the writable domains specified by the overridden method. Similarly, a formal parameter of an overriding method could be made read-only even if the corresponding parameter of the overridden method was write-enabled. I have not investigated the implications for advice matching of either relaxation.

Valid method overriding:

$$\frac{\mathit{methodType}(c\langle G_1,\ldots,G_q\rangle, m) = \tau \qquad \mathit{writable}(c\langle G_1,\ldots,G_q\rangle, m) = \hat{\gamma}}{\mathit{override}(m, c\langle G_1,\ldots,G_q\rangle, \tau, \hat{\gamma})}$$

$$\frac{\begin{array}{c}CT(c) = \text{class } c\langle G'_1,\ldots,G'_q\rangle \text{ extends } d\langle G'_1,\ldots,G'_r\rangle \ \{\ \mathit{field}^*\ \mathit{meth}_1 \ldots \mathit{meth}_p\ \} \\ \nexists i \in \{1..p\} \cdot \mathit{meth}_i = t\ m(\ \ldots\ )\ \mathit{eff}\ \{\ e\ \} \qquad \mathit{override}(m, d\langle G_1,\ldots,G_r\rangle, \tau, \hat{\gamma})\end{array}}{\mathit{override}(m, c\langle G_1,\ldots,G_q\rangle, \tau, \hat{\gamma})} \qquad \frac{}{\mathit{override}(m, \text{Object}\langle G\rangle, \tau, \hat{\gamma})}$$

Writable domains dependency closure:

$$\mathit{depClose}_{DT}(\hat{\gamma}) = \{\gamma' \cdot \exists \gamma \in \hat{\gamma} \cdot (\gamma, \gamma') \in DT\},$$
$$\text{where } DT : (\mathcal{G} \cup \mathcal{G}_{var}) \to (\mathcal{G} \cup \mathcal{G}_{var}) \text{ is reflexive and transitive, and } \forall \gamma \in \hat{\gamma} \cdot (\gamma, \gamma) \in DT$$

Valid class:

$$\frac{CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \text{ extends } d\langle G_1,\ldots,G_r\rangle\ \{\ \ldots\ \}}{\mathit{isClass}(\delta\ c\langle G'_1,\ldots,G'_q\rangle)} \qquad \frac{}{\mathit{isClass}(\delta\ \text{Object}\langle G\rangle)}$$

Binding typing:

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

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

Domain variables lookup:

$$\frac{CT(c) = \text{class } c\langle G_1,\ldots,G_q\rangle \ldots}{\mathit{domains}(c) = \langle G_1,\ldots,G_q\rangle} \qquad \frac{CT(a) = \text{aspect } a\langle G_1,\ldots,G_q\rangle \ldots}{\mathit{domains}(a) = \langle G_1,\ldots,G_q\rangle}$$

Figure 4.15 Auxiliary Functions for Static Semantics of MiniMAO$_2$

$$\text{T-TYPE}\quad \frac{\gamma_1 \in \{\gamma'_1,\ldots,\gamma'_r\} \qquad \forall i \in \{1..n\} \cdot \gamma_i \in \{G_1,\ldots,G_q\}}{\{\gamma'_1,\ldots,\gamma'_r\} \vdash T\langle \gamma_1,\ldots,\gamma_n\rangle \text{ OK in } S\langle G_1,\ldots,G_q\rangle}$$

$$\text{T-TYPERO}\quad \frac{\forall i \in \{1..n\} \cdot \gamma_i \in \{G_1,\ldots,G_q\}}{\{\gamma'_1,\ldots,\gamma'_r\} \vdash \text{readonly } T\langle \gamma_1,\ldots,\gamma_n\rangle \text{ OK in } S\langle G_1,\ldots,G_q\rangle}$$

Figure 4.16 Auxiliary Typing Judgments for Declarations in MiniMAO$_2$

that appeals to the class table, *CT*, is used to determine the number, q, of concern domain variables to include in the *override* judgment.)

The dependency table used for typechecking the method body is just the reflexive relation on the concern domain variables of the class; no dependency declarations are in scope with which to extend the relation.

Finally, T-MET checks the validity of the method's return type given the in-scope concern domain variables and without restriction on the read-only status of the return type. Returning a pointer into a concern domain does not mutate that domain; thus, the read-only status of the return type is not related to the effects clause of the method.

ASPECT TYPING The first hypothesis of the T-ASPECT rule checks that all concern domain variables appearing in the aspect's dependency declarations are in scope (see also T-DEP).

The hypothesis of T-ASPECT that checks advice declarations uses a dependency table formed from the dependency declarations of the aspect. The concern domain variables of the aspect declaration are also used.

Like T-CLASS, T-ASPECT requires that all fields pointing to other domains be read-only. T-ASPECT also checks that the aspect has at least one concern domain variable so that it may be instantiated.

ADVICE TYPING T-ADV checks the validity of the advice's formal parameter and return types as in T-MET, discussed above. Also like in T-MET, T-ADV includes the in-scope concern domain variables in the type environment and uses the effects clause to determine the set of writable concern domains when checking the advice body. T-ADV ensures that all concern domains listed in the effects clause of the advice specify concern domain variables that are in scope, again like T-MET.

T-ADV uses the dependency table supplied by T-ASP to check the advice body.

MiniMAO$_2$ extends the pointcut typing judgment to track the set of concern domains, $\hat{\gamma}$, named by writes pointcut descriptors within the pointcut. Section 4.3.2.4 describes these changes in more detail. For the present discussion it is useful to consider how the writable domains, $\hat{\gamma}$, matched by the pointcut should be related to the effects clause of the advice and to the host aspect's dependency declarations. Two hypotheses of T-ADV mediate these relationships.

The hypothesis $\hat{\gamma} \subseteq \{\gamma_1, \ldots, \gamma_r\}$ says that the writable domains of code matched by the advice must be a subset of those that the advice is declared to (possibly) mutate. This relationship ensures that the effects clause of the advice accounts for any mutation that might occur should the advice proceed to the advised code.

The hypothesis $\{\gamma_1, \ldots, \gamma_r\} \subseteq \mathit{depClose}_{DT}(\hat{\gamma})$ says that the effects clause of the advice may only list

— concern domains in the effects clause of code matched by the advice and

— concern domains for which a dependency declaration gives the advice permission.

How is this? Figure 4.15 on the preceding page gives the definition of $\mathit{depClose}_{DT}(\hat{\gamma})$, the *dependency closure* of $\hat{\gamma}$ in *DT*. The dependency closure here includes every element of $\hat{\gamma}$. For any concern domain, γ', named in the effects clause of the advice, but not in the effects clause of matched code, there must be some sequence of dependency declarations such that γ' "varies with" a writable concern domain, γ, of the matched code. But why is this the right notion? Intuitively, the dependency declarations tell a programmer that when this aspect is present in a program, calling a method that modifies γ may trigger advice that modifies γ'. The hypothesis of T-ADV prevents other concern domains from being modified.

An interesting consequence of these two hypotheses restricting the effects clause of advice is related to "pure" methods, methods whose effects clauses are empty [93]. The dependency closure of the empty set,

depClose($\emptyset$), is empty for any dependency table. Thus, the effects clause of advice on a pure method must be empty; only pure advice may bind to pure methods.

4.3.2.3 Expression Typing

The expression typing rules for MiniMAO$_2$ appear in Figure 4.17 on the facing page. Because of the large number of expression typing rules, and the fact that many of them change in similar ways, I structure this discussion based on the sorts of changes made to the rules versus MiniMAO$_1$. The T-TAG rule is new with MiniMAO$_2$; I discuss it in terms of the role it plays in the changes. Figure 4.18 on page 148 shows a class with several ill-typed expressions that serves as an example throughout this discussion.

PROPAGATING INFORMATION In most rules, the set of writable concern domains and the dependency table in the judgment must be passed along to subderivations. For several rules, this is the only necessary change. These mostly unchanged rules are T-VAR, T-GET, T-SEQ, T-PROC, and T-UNDER. Other rules that include this and other changes are T-CALL, T-EXEC, T-SET, T-CAST, T-CHAIN, and T-JOIN. The T-TAG rule uses the dependency table of its judgment in the hypothesis for typing the contained expression.

CHECKING CONCERN DOMAINS IN TYPES The static semantics must check the concern domains named in the expressions that explicitly give types—object instantiation and casts. Thus, MiniMAO$_2$ adds checks to T-NEW, T-OBJ, and T-CAST that any concern domains named in their expressions are in scope. T-NEW, and implicitly T-OBJ, also check that the number of concern domains named in the object instantiation expression match the number from the class declaration.

ENFORCING WRITABLE CONCERN DOMAINS WITH RESPECT TO STORE MODIFICATIONS The type rules for expression that may explicitly modify the store—object instantiation and field set—must enforce the writable concern domains permissions. In T-NEW, the hypothesis $\gamma_1 \in \hat{\gamma}$ says that the home domain of a new object must be writable. That is, instantiating an object within a concern domain requires write access to the domain. A similar hypothesis appears in the T-OBJ rule. Line 8 of the code in Figure 4.18, demonstrates code that is disallowed by T-NEW. The effects clause of the method abusingEffectsClause is empty. Since $(\text{what} \notin \emptyset)$, the expression cannot be typed.

This restriction may not be strictly necessary: in order for the instantiation to affect other code, the new object would have to be passed as a result or parameter, or else stored in a field and then subsequently dereferenced. All these uses could be checked. This would be equivalent to the handling of newly allocated objects in the pointer analysis of Rinard et al. [146]. For conceptual consistency, I am not allowing the allocation of new objects in non-writable domains. This also simplifies the statement of the meta-theory of effects in Section 4.4.3, because I do not have to account for allocated but unreferenced objects—no "garbage collection" on the store.

For the T-SET rule, I add an hypothesis, $\gamma_1 \in \hat{\gamma}$, that says that the home domain of the target object for the set must be writable. Line 6 in the sample code runs afoul of this hypothesis. The target object for the set expression, this.weBook, has its home domain represented by the concern domain variable what. Again $(\text{what} \notin \emptyset)$, so the expression cannot be typed.

ENFORCING WRITABLE CONCERN DOMAINS FOR METHOD CALLS MiniMAO$_2$ also includes a number of changes to enforce writable concern domains for method calls.

T-NEW
$$\frac{\mathit{domains}(T) = \langle G_1,\ldots,G_q\rangle \qquad \forall i \in \{1..q\} \cdot \Gamma(\gamma_i) = \text{domain} \qquad \gamma_1 \in \hat{\gamma}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{new } T\langle\gamma_1,\ldots,\gamma_q\rangle() : T\langle\gamma_1,\ldots,\gamma_q\rangle}$$

T-OBJ
$$\frac{\Gamma(\gamma) = \text{domain} \qquad \gamma \in \hat{\gamma}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{new Object}\langle\gamma\rangle() : \text{Object}\langle\gamma\rangle}$$

T-VAR
$$\frac{\Gamma(\mathit{var}) = t}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \mathit{var} : t}$$

T-LOC
$$\frac{\Gamma(\mathit{loc}) = t}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \mathit{loc}_\delta : \delta\ t}$$

T-NULL
$$\frac{t \in \mathscr{T}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{null}_\delta : \delta\ t}$$

T-CALL
$$\frac{\begin{array}{c}\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_0 : \delta\ T_0\langle\gamma_1,\ldots,\gamma_p\rangle \qquad \forall i \in \{1..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_i : u_i \\ \mathit{methodType}(\delta\ T_0\langle\gamma_1,\ldots,\gamma_p\rangle, m) = t_1 \times \ldots \times t_n \to t \qquad \mathit{writable}(\delta\ T_0\langle\gamma_1,\ldots,\gamma_p\rangle, m) = \hat{\gamma}' \\ \mathit{depClose}_{DT}(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (\delta = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \forall i \in \{1..n\} \cdot u_i \preccurlyeq t_i\end{array}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_0.m(\,e_1,\ldots,e_n\,) : t}$$

T-EXEC
$$\frac{\begin{array}{c}\Gamma, \mathit{var}_0 : t_0,\ldots,\mathit{var}_n : t_n \,.\, \mathit{depClose}_{DT}(\hat{\gamma}') \vdash_{DT} e : s \qquad s \preccurlyeq t \qquad \forall i \in \{0..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_i : u_i \\ \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i \qquad \mathit{depClose}_{DT}(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (\mathit{readonly}(u_0) = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \tau = t_0 \times \ldots \times t_n \to t\end{array}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} (\text{fun } m\langle \mathit{var}_0,\ldots,\mathit{var}_n\rangle.e : \tau \,.\, \hat{\gamma}'\ (\,e_0,\ldots,e_n\,)) : t}$$

T-GET
$$\frac{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e : s \qquad \mathit{fieldsOf}(s)(f) = t}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e.f : t}$$

T-SET
$$\frac{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_1 : T\langle\gamma_1,\ldots,\gamma_n\rangle \qquad \gamma_1 \in \hat{\gamma} \qquad \mathit{fieldsOf}(T\langle\gamma_1,\ldots,\gamma_n\rangle)(f) = t \qquad \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_2 : s \qquad s \preccurlyeq t}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_1.f = e_2 : s}$$

T-CAST
$$\frac{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e : s \qquad \mathit{readonly}(s) = \delta' \qquad \forall i \in \{1..q\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{cast } \delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle\ e : \delta\ \delta'\ T\langle\gamma_1,\ldots,\gamma_q\rangle}$$

T-SEQ
$$\frac{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_1 : s \qquad \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_2 : t}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_1;\ e_2 : t}$$

T-PROC
$$\frac{\forall i \in \{0..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_i : u_i \qquad \Gamma(\text{proceed}) = t_0 \times \ldots \times t_n \to t \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e_0.\text{proceed}(\,e_1,\ldots,e_n\,) : t}$$

T-UNDER
$$\frac{\Gamma \,.\, \hat{\gamma} \vdash_{DT} e : t}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{under } e : t}$$

T-CHAIN
$$\frac{\begin{array}{c}\forall i \in \{0..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{DT} e'_i : u'_i \qquad \forall i \in \{0..n\} \cdot u'_i \preccurlyeq t_i \\ \mathit{depClose}_{DT}(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (\mathit{readonly}(u'_0) = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \forall i \in \{1..p\} \cdot \Gamma \vdash b_i \text{ OK} \\ \forall i \in \{1..p\} \cdot \Gamma, \text{this} : \Gamma(\mathit{loc}_i), \text{proceed} : \tau, \mathit{typeBind}(\Gamma, b_i, \langle t_0,\ldots,t_n\rangle) \,.\, \mathit{depClose}_{DT}(\hat{\gamma}'_i) \vdash_{DT} e_i : s'_i \\ \forall i \in \{1..p\} \cdot s'_i \preccurlyeq t \qquad \forall i \in \{1..p\} \cdot \mathit{depClose}_{DT}(\hat{\gamma}'_i) \subseteq \mathit{depClose}_{DT}(\hat{\gamma}') \qquad \tau = t_0 \times \ldots \times t_n \to t\end{array}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{chain } [\![b_i, \mathit{loc}_i, e_i, \hat{\gamma}'_i, \tau', \tau]\!]_{i \in \{1..p\}}, (\!|\_,\_,\_,\_,\tau,\hat{\gamma}'|\!)(\,e'_0,\ldots,e'_n\,) : t}$$

T-JOIN
$$\frac{\begin{array}{c}\forall i \in \{0..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_i : u_i \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i \qquad \mathit{depClose}_{DT}(\hat{\gamma}') \subseteq \hat{\gamma} \\ (\mathit{readonly}(u_0) = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \\ (v_{opt} = \mathit{loc}_\delta) \implies (\mathit{loc} \in \mathit{dom}(\Gamma))\end{array}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \text{joinpt } (\!|\_, v_{opt}, \_, \_, (t_0 \times \ldots \times t_n \to t), \hat{\gamma}'|\!)(\,e_0,\ldots,e_n\,) : t}$$

T-TAG
$$\frac{\Gamma \,.\, \mathit{depClose}_{DT}(\hat{\gamma}') \vdash_{DT} e : t \qquad \mathit{depClose}_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}}{\Gamma \,.\, \hat{\gamma} \vdash_{DT} \langle e\rangle_{\delta,\hat{\gamma}'} : \delta\ t}$$

Figure 4.17 Static Semantics of Expressions in MiniMAO$_2$

```
1  class Rogue⟨what, who⟩ extends Object⟨what⟩ {
2    readonly Book⟨what, who⟩ roBook;
3    Book⟨what, who⟩ weBook; // write-enabled
4
5    Book⟨what, who⟩ abusingEffectsClause() writes ⟨⟩ { // pure method
6      this.weBook.title = null; // error!
7      this.weBook.setTitle(null); // error!
8      new Book⟨what, who⟩() // error!
9    }
10
11   void⟨what⟩ abusingReadonly() writes ⟨what, who⟩ {
12     this.roBook.title = null; // error!
13     this.roBook.setTitle(null); // error!
14     ...
15   }
16
17   Object⟨what⟩ castAway(readonly Object⟨what⟩ cantTouchThis) writes ⟨⟩ {
18     cast Object⟨what⟩ cantTouchThis // error!
19   }
20 }
```

Figure 4.18 Sample Expression Type Errors in MiniMAO$_2$

The T-CALL rule has two hypotheses for this purpose. One uses the *writable* auxiliary function (see Figure 4.9 on page 136) to look up the writable domains, $\hat{\gamma}'$, of the called method, m. Another new hypothesis, $depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}$, ensures that any concern domains that might be mutated by the call are in the set of writable concern domains. In Figure 4.18, line 7 violates this rule, assuming the declaration of Book from Figure 4.3 on 126. The setTitle method can write the what concern domain. The expression cannot be typed, because $\{\text{what}\} \not\subseteq \emptyset$.

Why use the dependency closure of the method's writable concern domains? This ensures that the rule considers effects on public concern domains of any advice that might bind to the method's call or execution. For static typechecking, the dependency table for checking methods is just reflexive, so the dependency closure does not matter. But it becomes necessary in the subject reduction proof.

I also add the hypothesis $depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}$ to each of T-JOIN, T-CHAIN, T-EXEC, and T-TAG to propagate writable concern domain information through the stages of advice binding and execution. The method body and advice bodies in the T-EXEC, T-CHAIN, and T-TAG rules are checked using a set of writable concern domains that is the dependency closure of the declared set. This is a convenience for the proofs of the meta-theory—fun application, chain, and tagged expressions do not appear in the user syntax—but is intuitively correct. To wit, the set of domains that might be modified by executing a method or advice body is, by definition, the dependency closure of the method's or advice's effects clause. The T-CHAIN rule also has an hypothesis that relates the effects clause of matching advice to that of the matched method, taking the dependency closure of both sets of concern domains. Again, this is a convenience for the proofs of the meta-theory.

ENFORCING READ-ONLY ANNOTATIONS The handling of readonly in the expression typing rules is motivated, in part, by the Universes type system [52, 117].

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

Figure 4.19 Binding for Type Environments

As discussed in Section 4.3.1.2 on page 134, in the operational semantics the read-only status of a pointer is combined with the type of the object to which it points to determine the type of the pointer. The T-LOC and T-NULL rules in MiniMAO$_2$ use the same technique for typing locations and null. MiniMAO$_2$ also updates the *typeBind* auxiliary function to use this technique for typing binding terms (see the first rule in Figure 4.19). The T-TAG rule uses a similar trick, but here the enclosed expression is not yet reduced to a value.

The T-SET rule, besides restricting the home domain of the target object, also checks that the target object is not read-only. This is through the hypothesis that gives the type of e_1 as $T\langle\gamma_1, \ldots, \gamma_n\rangle$; notice the lack of a read-only annotation on the type. So line 12 in Figure 4.18 on the facing page cannot be typed. The target of the set has a read-only type and so is not matched by this hypothesis.

The static semantics must also prevent mutation through method calls on read-only pointers. With the hypothesis $(\delta = \text{readonly}) \implies (\hat{\gamma}' = \emptyset)$, the T-CALL rule only allows calls to pure methods when using a read-only pointer to the target object. (Here δ is the read-only annotation on the type of e_0.) Line 13 is in conflict with this "purity hypothesis". It may be possible to relax the purity hypothesis; technically we would only need to ensure that the target object's representation was not mutated. But because an object's representation may extend into other concern domains (as in the Book example from Figure 4.1 on page 123), it is not immediately obvious how to relax this requirement. I leave the study of this to future work.

As with writable concern domains described above, I add hypotheses to T-JOIN, T-CHAIN, and T-EXEC to push the purity hypothesis of T-CALL through the proofs of the meta-theory.

Finally, the T-CAST rule concatenates any read-only annotation from the type of its expression and the type to which that expression is being cast. This statically ensures that the read-only status of an expression cannot be "cast away". For example, in line 18 of Figure 4.18, the variable reference cantTouchThis has type readonly Object⟨what⟩. Thus the type of the cast is also readonly Object⟨what⟩. So, the expression cannot be used as the result type of the method.

4.3.2.4 Pointcut Typing

Figure 4.20 on the following page gives the typing rules for pointcuts in MiniMAO$_2$. A pointcut typing judgment is of the form:

$$\Gamma \vdash \mathit{pcd} : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, \hat{u}'' \,.\, \hat{\gamma}_\perp \,.\, V_1 \,.\, V_2$$

where $\hat{u}$, $\hat{u}'$, U, $\hat{u}''$, V_1, and V_2 are unchanged from MiniMAO$_1$ (see Section 3.2.3.2). Only $\hat{\gamma}_\perp$ is new in MiniMAO$_2$. It gives the writable concern domains for any code under a join point matched by *pcd*, or is $\perp$ if that information cannot be determined from *pcd*.

The basic pointcut typing rules from MiniMAO$_1$—T-CALLPCD, T-EXECPCD, T-THISPCD, T-TARGPCD, and T-ARGSPCD—are changed as might be expected: they place a $\perp$ symbol in the writable concern domains slot of the pointcut type. Additionally, T-CALLPCD and T-EXECPCD include hypotheses that verify that any concern domains named in the pointcut descriptor are in scope.

$$U ::= \langle t^* \rangle \mid \bot \qquad \hat{u} ::= t \mid \bot \qquad \hat{\gamma}_\bot ::= \hat{\gamma} \mid \bot \qquad V \in \mathscr{P}(\mathcal{V})$$

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

T-CallPCD
$$\frac{\forall i \in \{1..q\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \vdash \text{call}(\ \delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle\ idPat(..)\ ) : \bot.\bot.\bot.\bot.\delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle.\bot.\emptyset.\emptyset}$$

T-ExecPCD
$$\frac{\forall i \in \{1..q\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \vdash \text{execution}(\ \delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle\ idPat(..)\ ) : \bot.\bot.\bot.\bot.\delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle.\bot.\emptyset.\emptyset}$$

T-WrtPCD
$$\frac{\forall i \in \{1..n\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \vdash \text{writes}(\ \gamma_1,\ldots,\gamma_n\ ) : \bot.\bot.\bot.\bot.\bot.\{\gamma_1,\ldots,\gamma_n\}.\emptyset.\emptyset}$$

T-ThisPCD
$$\frac{\Gamma(var) = t}{\Gamma \vdash \text{this}(\ t\ var\ ) : t.\bot.\bot.\bot.\bot.\bot.\{var\}.\{var\}}$$

T-TargPCD
$$\frac{\Gamma(var) = t}{\Gamma \vdash \text{target}(\ t\ var\ ) : \bot.t.\bot.\bot.\bot.\bot.\{var\}.\{var\}}$$

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

T-UnionPCD
$$\frac{\Gamma \vdash pcd_1 : \hat{u}.\hat{u}'.U.\hat{u}''.\hat{\gamma}_\bot.V_1.V_1' \qquad \Gamma \vdash pcd_2 : \hat{u}.\hat{u}'.U.\hat{u}''.\hat{\gamma}_\bot.V_2.V_2' \qquad V = V_1 \cap V_2 \qquad V' = V_1' \cup V_2'}{\Gamma \vdash pcd_1 \mid\mid pcd_2 : \hat{u}.\hat{u}'.U.\hat{u}''.\hat{\gamma}_\bot.V.V'}$$

T-NegPCD
$$\frac{\Gamma \vdash pcd : \hat{u}.\hat{u}'.U.\hat{u}''.\hat{\gamma}.V.V'}{\Gamma \vdash !\ pcd : \bot.\bot.\bot.\bot.\bot.\emptyset.\emptyset}$$

T-IntPCD
$$\frac{\begin{array}{c}\Gamma \vdash pcd_1 : \hat{u}_1.\hat{u}_1'.U_1.\hat{u}_1''.\hat{\gamma}_{\bot_1}.V_1.V_1' \qquad \Gamma \vdash pcd_2 : \hat{u}_2.\hat{u}_2'.U_2.\hat{u}_2''.\hat{\gamma}_{\bot_2}.V_2.V_2' \\ \hat{u} = \hat{u}_1 \sqcup \hat{u}_2 \qquad \hat{u}' = \hat{u}_1' \sqcup \hat{u}_2' \qquad U = U_1 \sqcup U_2 \qquad \hat{u}'' = \hat{u}_1'' \sqcup \hat{u}_2'' \\ \hat{\gamma}_\bot = \hat{\gamma}_{\bot_1} \sqcup \hat{\gamma}_{\bot_2} \qquad V_1' \cap V_2' = \emptyset \qquad V = V_1 \cup V_2 \qquad V' = V_1' \cup V_2'\end{array}}{\Gamma \vdash pcd_1\ \&\&\ pcd_2 : \hat{u}.\hat{u}'.U.\hat{u}''.\hat{\gamma}_\bot.V.V'}$$

Figure 4.20 Static Semantics of Pointcuts in MiniMAO$_2$

The new T-WRTPCD rule types writes pointcut descriptors. Like the preceding rules, T-WRTPCD verifies that any concern domains named in the pointcut descriptor are in scope. T-WRTPCD also records the writable concern domains in the pointcut type.

MiniMAO$_2$ extends the typing rules for the recursive pointcut descriptors in a straightforward way. The T-UNIONPCD rule requires that both combined pointcuts have the same set of writable concern domains. The T-INTPCD rule requires that at most one of the combined pointcuts is not $\bot$. As in MiniMAO$_1$, this level of exactness for the union and intersection rules is needed for type safety. Finally, T-NEGPCD is updated in the expected way.

4.4 Meta-theory

All the meta-theory from MiniMAO$_1$ must be updated to deal with concern domains, effects clauses, and read-only pointers. Besides this, I also introduce the meta-theoretic properties that derive from the new language features. These new properties include:

— *Effects clauses are effective.* Effects clauses, plus the configuration information given by aspect instantiation instructions and dependency declarations, are sufficient for determining the concern domains that may be modified by a method call or advice execution, even in the presence of other aspects.

— *Code cannot mutate an object's representation by dereferencing a* readonly *pointer to the object.* This is slightly different than the first property. This property says that a read-only pointer to a writable domain may not be used for mutation.

The conditions required to prove the second claim point out that reasoning challenges still exist for aspects as powerful as those in MiniMAO$_2$. But the results point the way to a solution—spectators. I discuss this more in Section 4.4.3.2.

I begin the exposition of the meta-theory by stating the auxiliary definitions and lemmas used in the proofs of the more interesting theorems. A subsequent section updates the type safety results for MiniMAO$_2$. A final section then gives the meta-theory for concern domains, effects clauses, and read-only pointers.

4.4.1 Auxiliary Definitions and Lemmas

This section presents the auxiliary definitions and lemmas of the meta-theory. I preface each updated definition or lemma with a few words describing how it has changed. For lemmas, I also highlight any interesting bits from the proof.

DEFINITIONS MiniMAO$_2$ has one new definition and several updated definitions that are used in the type safety proof. (A few other new definitions appear in Section 4.4.3 describing the meta-theory for concern domains, effects clauses, and read-only pointers.)

The one new definition here says that a type environment is "concern complete" if all concern domains in the program are in scope according to the environment. Static typechecking does not use concern-complete type environments, but they are used in the proofs which deal with program evaluation.

Definition 4.2 (Concern-Complete Environments). Given a well-typed program P with concern domains $\hat{g}$, we say that a type environment Γ is *concern complete* if

$$\forall g \in \hat{g} \cdot \Gamma(g) = \text{domain}.$$

For MiniMAO$_2$, I update the definition of environment-store consistency to account for read-only annotations on field types. This change only affects part 1(d) of the definition.

Definition 4.3 (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:

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

The join point abstraction in the definition of stack-store consistency gets an additional slot corresponding to the set of writable concern domains at the join point. However, the value in that slot does not matter for this definition. The *loc* slot in the join point abstraction also gets a read-only annotation, δ, but that is also ignored.

Definition 4.4 (Stack-Store Consistency). A stack J and a store S are *consistent*, and we write $J \approx S$, if

$$\forall \langle\!\langle \_, loc_\delta, \_, \_, \_, \_ \rangle\!\rangle \in J \cdot loc \in dom(S).$$

In MiniMAO$_1$ a valid store contains a single instance of every aspect declared in the program. Rather than having one instance of each aspect, MiniMAO$_2$ includes aspect instantiation instructions. I update the definition of a valid store accordingly.

Definition 4.5 (Store Validity). Given a well-typed program P with aspect instantiation instructions

$$\text{use } a_1\langle g_{1,1},\ldots,g_{1,p_1}\rangle;\ \ldots;\ \text{use } a_n\langle g_{n,1},\ldots,g_{n,p_n}\rangle,$$

we say that a store S is *valid* if both of the following hold:

1. $\forall i \in \{1..n\} \cdot \big(\exists loc \in \mathscr{L} \cdot S(loc) = \big[a_i\langle g_{i,1},\ldots,g_{i,p_i}\rangle \,.\, F\big]\big)$
2. $\exists \Gamma \cdot \Gamma \approx S$

LEMMAS The type safety proof for MiniMAO$_2$ requires two new lemmas. All lemmas from MiniMAO$_1$ also receive some tweaks.

The first new lemma says that if one set of concern domain names and variables is a subset of the dependency closure of another, then the dependency closure of the first is also a subset of the dependency closure of the second. The essential reason for this is that the same dependency table is used throughout. Figure 4.21 on the next page gives a Venn diagram that may be helpful when reading the proof of the lemma.

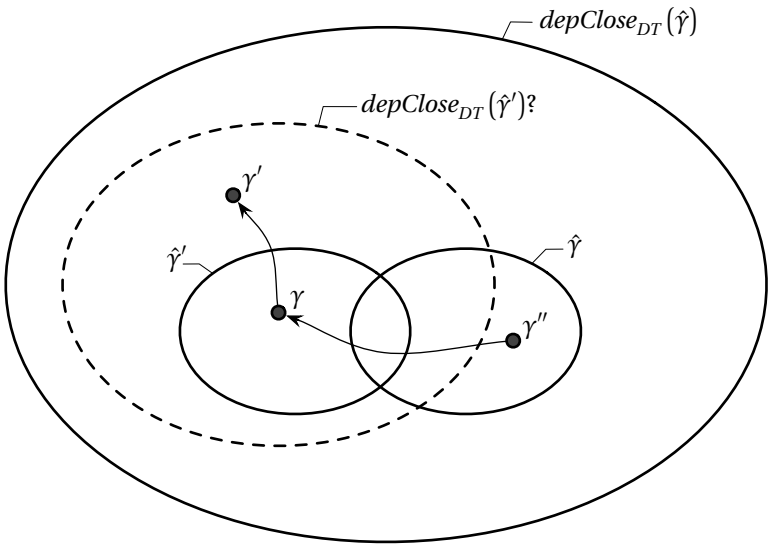

Figure 4.21 Venn Diagram Illustrating Lemma 4.6

Lemma 4.6 (Dependency Closure Inclusion). *Let P be a program with concern domains $\hat{g}$ and evaluation dependency table DT. If $\hat{\gamma} \subseteq \hat{g}$, $\hat{\gamma}' \subseteq \hat{g}$, and $\hat{\gamma}' \subseteq depClose_{DT}(\hat{\gamma})$, then $depClose_{DT}(\hat{\gamma}') \subseteq depClose_{DT}(\hat{\gamma})$.*

Proof. Because DT is constant throughout the proof, I elide it where practical. Let γ' be an arbitrary element of $depClose(\hat{\gamma}')$. By definition of *depClose*, there exists $\gamma \in \hat{\gamma}'$ such that $(\gamma, \gamma') \in DT$. But $\gamma \in \hat{\gamma}'$ implies $\gamma \in depClose(\hat{\gamma})$ by the assumption of the lemma. So again by the definition of *depClose*, there exists $\gamma'' \in \hat{\gamma}$ such that $(\gamma'', \gamma) \in DT$. Now DT is reflexive and transitive, so $(\gamma'', \gamma') \in DT$. By the definition of *depClose*, $\gamma'' \in \hat{\gamma} \implies \gamma' \in depClose(\hat{\gamma})$.

So every element of $depClose(\hat{\gamma}')$ is also an element of $depClose(\hat{\gamma})$. □

The second new lemma says that a typing judgment that holds with a given set of writable concern domains and a given dependency table, also holds using a new dependency table that is a superset of the first. However, the set of writable concern domains in the new typing judgment must be the dependency closure (over the new dependency table) of the original set. I use this lemma in the subject reduction proof to lift the separately typechecked derivations for method and advice bodies into the evaluation, where the whole-program evaluation dependency table must be used.

Lemma 4.7 (Dependency Table Extension). *If e includes only user syntax, $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$, $DT \subseteq DT_2$, and $\forall \gamma \in \hat{\gamma} \cdot (\gamma, \gamma) \in DT_2$, then*

$$\Gamma \centerdot depClose_{DT_2}(\hat{\gamma}) \vdash_{DT_2} e : t.$$

Proof. The proof is by structural induction on the derivation of $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$. The base cases are T-NEW, T-OBJ, T-VAR, and T-NULL. (We do not need to consider T-LOC, because locations are not part of the user syntax.) For all of these, the judgment does not depend on DT, so the claim holds.

The remaining expression typing rules constitute the induction steps. The induction hypothesis is that the claim of the lemma holds for all derivations smaller than the one under consideration. For T-GET, T-SET, T-CAST, T-SEQ, T-PROC, and T-UNDER, the claim is immediate from the induction hypothesis.

All but one of the hypotheses of T-CALL hold immediately by the induction hypothesis. The one hypothesis from the derivation of $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$ that we must consider is $depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}$, where $\hat{\gamma}'$ is the set of writable domains from the effects clause of the called method. The corresponding hypothesis from the derivation of $\Gamma \centerdot depClose_{DT_2}(\hat{\gamma}) \vdash_{DT_2} e : t$ is $depClose_{DT_2}(\hat{\gamma}') \subseteq depClose_{DT_2}(\hat{\gamma})$.

First, note that $depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}$ implies $\hat{\gamma}' \subseteq \hat{\gamma}$. To see this, take $\gamma' \in \hat{\gamma}'$. By definition,

$$\gamma' \in depClose_{DT}(\hat{\gamma}')$$

and thus $\gamma' \in \hat{\gamma}$.

Next, note that $\hat{\gamma}' \subseteq \hat{\gamma}$ implies $depClose_{DT_2}(\hat{\gamma}') \subseteq depClose_{DT_2}(\hat{\gamma})$. To see this, take $\gamma' \in depClose_{DT_2}(\hat{\gamma}')$. Then there exists $\gamma \in \hat{\gamma}'$ such that $(\gamma, \gamma') \in DT_2$. But $\hat{\gamma}' \subseteq \hat{\gamma}$ then implies that there exists $\gamma \in \hat{\gamma}$ such that $(\gamma, \gamma') \in DT_2$. So $\gamma' \in depClose_{DT_2}(\hat{\gamma})$.

Thus, by T-CALL $\Gamma \centerdot depClose_{DT_2}(\hat{\gamma}) \vdash_{DT_2} e : t$, and the claim holds for this case.

The remaining expression typing rules—T-EXEC, T-CHAIN, T-JOIN, and T-TAG—do not apply to user syntax. Thus, the claim holds. □

The Substitution lemma uses the new expression typing judgments, with their sets of writable domains and dependency tables, but these are consistent throughout the statement of the lemma. Otherwise the statement of the Substitution lemma is unchanged from MiniMAO$_1$.

Lemma 4.8 (Substitution). *If* $\Gamma, var_1 : t_1, \ldots, var_n : t_n \centerdot \hat{\gamma} \vdash_{DT} e : t$ *and* $\forall i \in \{1..n\} \cdot \Gamma \centerdot \hat{\gamma} \vdash_{DT} e_i : s_i$ *where* $s_i \preccurlyeq t_i$ *then* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e\{e_1/\, var_1, \ldots, e_n/\, var_n\} : s$ *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 $\{e_1/\, var_1, \ldots, e_n/\, var_n\}$. I omit the *DT* subscript for the remainder of the proof, with the understanding that the same dependency table is used throughout.

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. Here $e = e'_0.m(\, e'_1, \ldots, e'_p\,)$. The last type derivation step has the following form:

$$\frac{\begin{array}{c}\Gamma' \centerdot \hat{\gamma} \vdash e'_0 : \delta\ T_0\langle \gamma_1, \ldots, \gamma_q \rangle \qquad \forall i \in \{1..p\} \cdot \Gamma' \centerdot \hat{\gamma} \vdash e'_i : u'_i \\ methodType(\delta\ T_0\langle \gamma_1, \ldots, \gamma_q \rangle, m) = u_1 \times \ldots \times u_p \to t \qquad writable(\delta\ T_0\langle \gamma_1, \ldots, \gamma_q \rangle, m) = \hat{\gamma}' \\ depClose(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (\delta = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \forall i \in \{1..p\} \cdot u'_i \preccurlyeq u_i\end{array}}{\Gamma' \centerdot \hat{\gamma} \vdash e : t}$$

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 next that T-CALL also gives $\Gamma \vdash e \{\!| \bar{e} / \overline{var} |\!\} : s$ for some $s \preccurlyeq t$. By the induction hypothesis, $\Gamma \vdash e_0'' : u_0''$, where $u_0'' \preccurlyeq \delta\ T_0\langle \gamma_1, \ldots, \gamma_q \rangle$. By the definition of subtyping, $u_0'' = \delta'\ T_0'\langle \gamma_1, \ldots, \gamma_r \rangle$ with $(\delta' = \text{readonly}) \implies (\delta = \text{readonly})$, $T_0'\langle \gamma_1, \ldots, \gamma_r \rangle \preccurlyeq T_0\langle \gamma_1, \ldots, \gamma_q \rangle$, and $r \geq q$ by T-CLASS.

Now suppose $\delta' = \text{readonly}$. Then $\delta = \text{readonly}$ and

$$methodType(u_0'', m) = methodType(\delta\ T_0\langle \gamma_1, \ldots, \gamma_q \rangle, m)$$

by the definitions of *methodType* and *override*.

Otherwise $\delta' = \varepsilon$ and $methodType(u_0'', m) = u_1 \times \ldots \times u_p \to s$, where readonly $s = t$ (i.e., s is t without a readonly annotation) if $\delta = \text{readonly}$ and $s = t$ otherwise. In either case $s \preccurlyeq t$.

To discharge the remaining hypotheses, we note that $writable(u_0'', m) = \hat{\gamma}'$, again by the definition of *override*. Furthermore $(\delta' = \text{readonly}) \implies (\delta = \text{readonly}) \implies (\hat{\gamma}' = \emptyset)$. 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 $e = (\,\text{fun}\ m\langle var_0', \ldots, var_p' \rangle.e' : \tau\,.\,\hat{\gamma}'\ (\, e_0', \ldots, e_p' \,))$, where $\tau = u_0' \times \ldots \times u_p' \to t$. The last derivation step is:

$$\frac{\begin{array}{c}\Gamma, var_0' : u_0', \ldots, var_p' : u_p'\,.\,depClose(\hat{\gamma}') \vdash e' : s' \qquad s' \preccurlyeq t \qquad \forall i \in \{0..p\} \cdot \Gamma\,.\,\hat{\gamma} \vdash e_i' : u_i \qquad depClose(\hat{\gamma}') \subseteq \hat{\gamma} \\ (readonly(u_0) = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \forall i \in \{0..p\} \cdot u_i \preccurlyeq u_i' \qquad \tau = u_0' \times \ldots \times u_p' \to t\end{array}}{\Gamma'\,.\,\hat{\gamma} \vdash e : t}$$

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

$$e \{\!| \bar{e} / \overline{var} |\!\} = (\,\text{fun}\ m\langle var_0', \ldots, var_p' \rangle.e'' : \tau\,.\,\hat{\gamma}'\ (\, e_0'', \ldots, e_p'' \,)).$$

By the induction hypothesis, for $i \in \{1..p\}$, $\Gamma\,.\,\hat{\gamma} \vdash e_i'' : u_i''$ where $u_i'' \preccurlyeq u_i$. Also, if $readonly(u_0'') = \text{readonly}$ then $readonly(u_0) = \text{readonly}$ (by the definition of subtyping) and $\hat{\gamma}' = \emptyset$ (by hypothesis of T-EXEC above). Finally, by T-EXEC and transitivity of subtyping, $\Gamma\,.\,\hat{\gamma} \vdash e \{\!| \bar{e} / \overline{var} |\!\} : t$.

Case 3—T-GET. In this case $e = e'.f$. The last step in the type derivation for e is

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

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 type.

If $isClass(u')$, then by the definition of *fieldsOf* and by the first hypothesis of T-CLASS, $fieldsOf(u')(f) = s \preccurlyeq t = fieldsOf(u)(f)$, where $\delta\ s = t$ for some δ. In this case, $\Gamma \vdash e \{\!| \bar{e} / \overline{var} |\!\} : s$ and the claim holds.

On the other hand, if u' is an aspect, then $u' = u$ (since an aspect is only a subtype of itself and Object, and $u \neq \text{Object}$ because $fieldsOf(u) \neq \emptyset$). So $fieldsOf(u')(f) = fieldsOf(u)(f) = t$, $\Gamma \vdash e \{\!| \bar{e} / \overline{var} |\!\} : t$, and again the claim holds.

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

$$\frac{\Gamma' \bullet \hat{\gamma} \vdash e'_1 : T_1\langle\gamma_1,\ldots,\gamma_p\rangle \qquad \gamma_1 \in \hat{\gamma} \qquad \mathit{fieldsOf}\left(T_1\langle\gamma_1,\ldots,\gamma_p\rangle\right)(f) = u \qquad \Gamma' \bullet \hat{\gamma} \vdash e'_2 : t \qquad t \preccurlyeq u}{\Gamma' \bullet \hat{\gamma} \vdash e'_1.f = e'_2 : t}$$

Now $e\{\!|\bar{e}/\overline{var}|\!\} = (e'_1\{\!|\bar{e}/\overline{var}|\!\}.f = e'_2\{\!|\bar{e}/\overline{var}|\!\})$. By the induction hypothesis $\Gamma \bullet \hat{\gamma} \vdash e'_1\{\!|\bar{e}/\overline{var}|\!\} : T'_1\langle\gamma_1,\ldots,\gamma_q\rangle$, $T'_1\langle\gamma_1,\ldots,\gamma_q\rangle \preccurlyeq T_1\langle\gamma_1,\ldots,\gamma_p\rangle$ and $\Gamma \bullet \hat{\gamma} \vdash e'_2\{\!|\bar{e}/\overline{var}|\!\} : t'$, $t' \preccurlyeq t$.

If T'_1 is a class, then by the definition of *fieldsOf* and the first hypothesis of T-CLASS, we have

$$\mathit{fieldsOf}\left(T'_1\langle\gamma_1,\ldots,\gamma_q\rangle\right)(f) = \mathit{fieldsOf}\left(T_1\langle\gamma_1,\ldots,\gamma_p\rangle\right)(f) = u.$$

On the other hand, if T'_1 is an aspect then, by the same argument as in the T-GET case, $T'_1 = T_1$, $p = q$, and again $\mathit{fieldsOf}\left(T'_1\langle\gamma_1,\ldots,\gamma_q\rangle\right)(f) = \mathit{fieldsOf}\left(T_1\langle\gamma_1,\ldots,\gamma_p\rangle\right)(f) = u$.

In either case, by transitivity $t' \preccurlyeq u$. Therefore, $\Gamma \bullet \hat{\gamma} \vdash e\{\!|\bar{e}/\overline{var}|\!\} : t'$, where $t' \preccurlyeq t$ and the claim holds.

Case 5—T-CAST. In this case, $e = \mathsf{cast}\ t\ e'$, where $t = \delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle$. Here the last derivation step is:

$$\frac{\Gamma' \bullet \hat{\gamma} \vdash e : u \qquad \mathit{readonly}(u) = \delta' \qquad \forall i \in \{1..q\} \cdot \Gamma\left(\gamma_i\right) = \mathsf{domain}}{\Gamma' \bullet \hat{\gamma} \vdash \mathsf{cast}\ t\ e' : \delta'\ t}$$

By the induction hypothesis, $\Gamma \bullet \hat{\gamma} \vdash e'\{\!|\bar{e}/\overline{var}|\!\} : u'$ where $u' \preccurlyeq u$. Let $\mathit{readonly}\left(u'\right) = \delta''$. We need to show that $\delta\ \delta''\ T\langle\gamma_1,\ldots,\gamma_q\rangle \preccurlyeq \delta\ \delta'\ T\langle\gamma_1,\ldots,\gamma_q\rangle$. If $\delta = \mathsf{readonly}$ this holds by idempotency of read-only annotations. If $\delta = \varepsilon$, then we must show

$$\delta''\ T\langle\gamma_1,\ldots,\gamma_q\rangle \preccurlyeq \delta'\ T\langle\gamma_1,\ldots,\gamma_q\rangle. \tag{4.1}$$

If $\delta'' = \varepsilon$ this holds. If $\delta'' = \mathsf{readonly}$, then by the definition of subtyping $\delta' = \mathsf{readonly}$. So (4.1) holds.

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

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

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

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

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

Let $e''_i = e'_i\{\!|\bar{e}/\overline{var}|\!\}$ for all $i \in \{0..p\}$. Then $e\{\!|\bar{e}/\overline{var}|\!\} = e''_0.\mathsf{proceed}(\ e''_1,\ldots,e''_p\ )$. Now $\Gamma(\mathsf{proceed}) = \Gamma'(\mathsf{proceed}) = u_0 \times \ldots \times u_p \to t$ and by the induction hypothesis

$$\forall i \in \{0..p\} \cdot \left(\Gamma \bullet \hat{\gamma} \vdash e''_i : u''_i, \text{where } u''_i \preccurlyeq u'_i \preccurlyeq u_i\right).$$

Thus, by T-PROC, $\Gamma \bullet \hat{\gamma} \vdash e\{\!|\bar{e}/\overline{var}|\!\} : t$ and the claim holds.

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

$$\frac{\Gamma' \centerdot \hat{\gamma} \vdash e' : t}{\Gamma' \centerdot \hat{\gamma} \vdash \mathsf{under}\ e' : t}$$

The claim is immediate by the induction hypothesis.

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

$$\forall i \in \{0..p\} \cdot \Gamma' \centerdot \hat{\gamma} \vdash e'_i : u'_i \qquad \forall i \in \{0..p\} \cdot u'_i \preccurlyeq u_i \qquad \big(\mathit{readonly}(u'_0) = \mathsf{readonly}\big) \implies \big(\hat{\gamma}' = \emptyset\big)$$

Let $e''_i = e'_i\{\!|\bar{e}/\overline{var}|\!\}$ for all $i \in \{0..p\}$. Then

$$e\{\!|\bar{e}/\overline{var}|\!\} = \mathsf{chain}\ \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (u_0 \times \ldots \times u_p \to t), \hat{\gamma}'|\!)(\ e''_0, \ldots, e''_p\ ).$$

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 $\forall i \in \{0..p\} \cdot \big(\Gamma \centerdot \hat{\gamma} \vdash e''_i : u''_i$, where $u''_i \preccurlyeq u'_i \preccurlyeq u_i\big)$. Also $\big(\mathit{readonly}(u''_0) = \mathsf{readonly}\big) \implies \big(\mathit{readonly}(u'_0) = \mathsf{readonly}\big) \implies \big(\hat{\gamma}' = \emptyset\big)$. Because substitution does not replace variables within $\bar{B}$ or within the join point abstraction, 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, since none of the variables in the statement of the lemma are free in the tuples of $\bar{B}$. Thus, $\Gamma \centerdot \hat{\gamma} \vdash e\{\!|\bar{e}/\overline{var}|\!\} : t$.

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

Case 11—T-TAG. Here $e = \langle e' \rangle_{\delta, \hat{\gamma}'}$ and the last step in the type derivation is

$$\frac{\Gamma' \centerdot \mathit{depClose}(\hat{\gamma}') \vdash e' : t' \qquad \mathit{depClose}(\hat{\gamma}') \subseteq \hat{\gamma}}{\Gamma' \centerdot \hat{\gamma} \vdash \langle e' \rangle_{\delta, \hat{\gamma}'} : t}$$

where $t = \delta\ t'$. Now $e\{\!|\bar{e}/\overline{var}|\!\} = \langle e'\{\!|\bar{e}/\overline{var}|\!\} \rangle_{\delta, \hat{\gamma}'}$. By the induction hypothesis $\Gamma \centerdot \mathit{depClose}(\hat{\gamma}') \vdash e'\{\!|\bar{e}/\overline{var}|\!\} : s'$ where $s' \preccurlyeq t'$. Consider two cases.

If $\delta = \varepsilon$, then $t = t'$ and, by T-TAG, $\Gamma \centerdot \hat{\gamma} \vdash e\{\!|\bar{e}/\overline{var}|\!\} : s'$ and the claim holds.

On the other hand, if $\delta = \mathsf{readonly}$, then by T-TAG $\Gamma \centerdot \hat{\gamma} \vdash e\{\!|\bar{e}/\overline{var}|\!\} : \mathsf{readonly}\ s'$. By the definition of subtyping $\mathsf{readonly}\ s' \preccurlyeq \mathsf{readonly}\ t' = t$, thus the claim holds. □

The Environment Extension lemma holds the set of writable concern domains and the dependency table constant.

Lemma 4.9 (Environment Extension). *If* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$ *and* $a \notin \mathit{dom}(\Gamma)$, *then* $\Gamma, a : t' \centerdot \hat{\gamma} \vdash_{DT} e : t$.

Proof. The proof is by a straightforward structural induction on the derivation of $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$.

For the base case, the last step in the derivation is T-NULL, T-NEW, T-OBJ, T-VAR, or T-LOC. In the first case, the type environment does not appear in the hypothesis of the judgment, so the claim holds. For

T-NEW and T-OBJ, $a \notin dom(\Gamma)$ implies that no hypotheses change, so the claim holds. For the T-VAR case, $e = var$ and $\Gamma(var) = t$. But $a \notin dom(\Gamma)$, so $var \neq a$. Therefore $(\Gamma, a : t')(var) = t$ and the claim holds for this case. The T-LOC case is similar.

The remaining typing rules cover the induction step. By the induction hypothesis, changing the type environment to $\Gamma, a : t'$ does not change the types assigned by any hypotheses. Furthermore, because $a \notin dom(\Gamma)$, we have $\forall \gamma \cdot \Gamma(\gamma) = \text{domain} \implies (\Gamma, a : t')(\gamma) = \text{domain}$. Therefore, the types assigned by each rule are also unchanged and the claim holds. □

Like Lemma 4.9 (Environment Extension) on the preceding page, the Environment Contraction lemma holds the set of writable concern domains and the dependency table constant. The lemma states that unused type mappings may be dropped from the type environment in a typing judgment without changing the judgment. The lemma does not allow domain mappings to be dropped from the environment, though in principle this could be done if the dropped domain variable did not appear in the writable domains of the environment.

Lemma 4.10 (Environment Contraction). *If* $\Gamma, a : t' \centerdot \hat{\gamma} \vdash_{DT} e : t$, *a is not free in e, and* $t' \neq domain$, *then* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$.

Proof. The proof is by a straightforward structural induction on the derivation of $\Gamma, a : t' \centerdot \hat{\gamma} \vdash_{DT} e : t$.

For the base case, the last step in the derivation is T-NULL, T-NEW, T-OBJ, T-VAR, or T-LOC. In the first case, the type environment does not appear in the hypothesis of the judgment, so the claim holds. For T-NEW and T-OBJ, $t' \neq \text{domain}$ implies that no hypotheses change, so the claim holds. For the T-VAR case, $e = var$ and $(\Gamma, a : t')(var) = t$. But a is not free in e, so $var \neq a$. Therefore $\Gamma(var) = t$ and the claim holds for this case. The T-LOC case is similar.

The remaining typing rules cover the induction step. By the induction hypothesis, changing the type environment to Γ does not change the types assigned by any hypotheses. Furthermore, because $t' \neq \text{domain}$, $\forall \gamma \cdot (\Gamma, a : t')(\gamma) = \text{domain} \implies \Gamma(\gamma) = \text{domain}$. Therefore, the types assigned by each rule are also unchanged and the claim holds. □

In MiniMAO$_2$, the Replacement and Replacement with Subtyping lemmas allow subexpressions to be typed using a subset of the writable concern domains from the outer typing judgments. This is necessary to allow substitution within tagged expressions, for example.

Lemma 4.11 (Replacement). *If* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} \mathbb{E}[e] : t$, $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e : t'$, *and* $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e' : t'$ *for some* $\hat{\gamma}' \subseteq \hat{\gamma}$, *then* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} \mathbb{E}[e'] : t$.

Proof. By examining the evaluation context rules and corresponding typing rules, we see that $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e : t'$ must be a sub-derivation of $\Gamma \centerdot \hat{\gamma} \vdash_{DT} \mathbb{E}[e] : t$. Now the typing derivation for $\Gamma \centerdot \hat{\gamma} \vdash_{DT} \mathbb{E}[e'] : t''$ must have the same shape as that for $\mathbb{E}[e] : t$, except for the sub-derivation for $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e' : t'$. However, because this sub-derivation yields the same type and uses the same environment as the sub-derivation it replaces, it must be the case that $t'' = t$. □

Lemma 4.12 (Replacement with Subtyping). *If* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} \mathbb{E}[e] : t$, $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e : u$, *and* $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e' : u'$ *where* $u' \preccurlyeq u$ *and* $\hat{\gamma}' \subseteq \hat{\gamma}$, *then* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} \mathbb{E}[e'] : t'$ *where* $t' \preccurlyeq t$.

Proof. The proof is by induction on the size of the evaluation context $\mathbb{E}$, where the size is the number of recursive applications of the syntactic rules necessary to build $\mathbb{E}$. In the base case, $\mathbb{E}$ has size zero, $\mathbb{E} = -$, $\hat{\gamma}' = \hat{\gamma}$, and $t' = u' \preccurlyeq u = t$.

For the induction step we divide the evaluation context into two parts so that $\mathbb{E}[-] = \mathbb{E}_1[\mathbb{E}_2[-]]$, where $\mathbb{E}_2$ has size one. The induction hypothesis is that the claim of the lemma holds for all evaluation contexts smaller than the one considered in the induction step, and therefore holds for $\mathbb{E}_1$. We use a case analysis on the rule used to generate $\mathbb{E}_2$. In each case we show that if $\Gamma \centerdot \hat{\gamma}'' \vdash_{DT} \mathbb{E}_2[e] : s$ then $\Gamma \centerdot \hat{\gamma}'' \vdash_{DT} \mathbb{E}_2[e'] : s'$ where $s' \preccurlyeq s$ and $\hat{\gamma}' \subseteq \hat{\gamma}'' \subseteq \hat{\gamma}$, and therefore the claim holds by the induction hypothesis. I omit the *DT* subscript for the remainder of the proof, with the understanding that the same dependency table is used throughout.

Case 1—$\mathbb{E}_2 = -.m(\, e_1, \ldots, e_n\,)$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-CALL:

$$\frac{\begin{array}{c}\Gamma \centerdot \hat{\gamma}'' \vdash e : \delta\ T\langle\gamma_1, \ldots, \gamma_p\rangle \qquad \forall i \in \{1..n\} \cdot \Gamma \centerdot \hat{\gamma}'' \vdash e_i : u_i \\ \mathit{methodType}(\delta\ T\langle\gamma_1, \ldots, \gamma_p\rangle, m) = s_1 \times \ldots \times s_n \to s \qquad \mathit{writable}(\delta\ T\langle\gamma_1, \ldots, \gamma_p\rangle, m) = \hat{\gamma}_\mathsf{m} \\ \mathit{depClose}(\hat{\gamma}_\mathsf{m}) \subseteq \hat{\gamma}'' \qquad (\delta = \text{readonly}) \implies (\hat{\gamma}_\mathsf{m} = \emptyset) \qquad \forall i \in \{1..n\} \cdot u_i \preccurlyeq s_i\end{array}}{\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

where $u = \delta\ T\langle\gamma_1, \ldots, \gamma_p\rangle$ and $\hat{\gamma}' = \hat{\gamma}''$. By the definitions of *override* and *writable*,

$$\mathit{writable}(u, m) = \mathit{writable}(u', m).$$

By the definition of subtyping, $u' = \delta'\ S\langle\gamma_1, \ldots, \gamma_q\rangle$, where $S\langle\gamma_1, \ldots, \gamma_q\rangle \preccurlyeq T\langle\gamma_1, \ldots, \gamma_p\rangle$.

There are two possibilities depending on the value of δ'. If $\delta' =$ readonly, then by the definition of subtyping $\delta =$ readonly too. So by the definitions of *override* and *methodType*, $\mathit{methodType}(u', m) = \mathit{methodType}(u, m)$. The remaining hypotheses are unchanged, so T-CALL gives $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

On the other hand, if $\delta' = \varepsilon$, then $\mathit{methodType}(u', m) = s_1 \times \ldots \times s_n \to s'$, where $\delta\ s' = s$. The remaining hypotheses all hold, so T-CALL gives $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s'$. Regardless of the value of δ, $s' \preccurlyeq s$, so the claim holds.

Case 2—$\mathbb{E}_2 = v_0.m(\, v_1, \ldots, v_{p-1}, -, e_{p+1}, e_n\,)$ *where* $p \in \{1..n\}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-CALL, with $\hat{\gamma}' = \hat{\gamma}''$:

$$\frac{\begin{array}{c}\Gamma \centerdot \hat{\gamma}'' \vdash v_0 : \delta\ T_0\langle\gamma_1, \ldots, \gamma_q\rangle \\ \forall i \in \{1..(p-1)\} \cdot \Gamma \centerdot \hat{\gamma}'' \vdash v_i : u_i \qquad \Gamma \centerdot \hat{\gamma}'' \vdash e : u \qquad \forall i \in \{(p+1)..n\} \cdot \Gamma \centerdot \hat{\gamma}'' \vdash e_i : u_i \\ \mathit{methodType}(\delta\ T_0\langle\gamma_1, \ldots, \gamma_q\rangle, m) = s_1 \times \ldots \times s_n \to s \qquad \mathit{writable}(\delta\ T_0\langle\gamma_1, \ldots, \gamma_q\rangle, m) = \hat{\gamma}_\mathsf{m} \\ \mathit{depClose}(\hat{\gamma}_\mathsf{m}) \subseteq \hat{\gamma}'' \qquad (\delta = \text{readonly}) \implies (\hat{\gamma}_\mathsf{m} = \emptyset) \qquad \forall i \in \{1..n\} \setminus \{p\} \cdot u_i \preccurlyeq s_i \qquad u \preccurlyeq s_p\end{array}}{\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

Now $u' \preccurlyeq u \preccurlyeq s_p$, so by T-CALL $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

Case 3—$\mathbb{E}_2 = (\ l\ (\ v_0, \ldots, v_{p-1}, -, e_{p+1}, e_n\ )\ )$ *where* $p \in \{0..n\}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-EXEC (where $u = u_p$ and $\hat{\gamma}' = \hat{\gamma}''$):

$$\frac{\begin{array}{c}\Gamma, var_0 : s_0, \ldots, var_n : s_n \,.\, depClose(\hat{\gamma}_\mathsf{m}) \vdash e'' : u'' \qquad u'' \preccurlyeq s \qquad \forall i \in \{0..(p-1)\} \cdot \Gamma \,.\, \hat{\gamma}'' \vdash v_i : u_i \\ \Gamma \,.\, \hat{\gamma}'' \vdash e : u_p \qquad \forall i \in \{(p+1)..n\} \cdot \Gamma \,.\, \hat{\gamma}'' \vdash e_i : u_i \qquad \forall i \in \{0..n\} \setminus \{p\} \cdot u_i \preccurlyeq s_i \\ u_p \preccurlyeq s_p \qquad depClose(\hat{\gamma}_\mathsf{m}) \subseteq \hat{\gamma}'' \qquad (readonly(u_0) = \mathsf{readonly}) \implies (\hat{\gamma}_\mathsf{m} = \emptyset)\end{array}}{\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

where $l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e'' : (s_0 \times \ldots \times s_n \to s) \,.\, \hat{\gamma}_\mathsf{m}$. Now $u' \preccurlyeq u = u_p \preccurlyeq s_p$. If $p = 0$, then

$$(readonly(u') = \mathsf{readonly}) \implies (readonly(u_0) = \mathsf{readonly}) \implies (\hat{\gamma}_\mathsf{m} = \emptyset).$$

So by T-EXEC, $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

Case 4—$\mathbb{E}_2 = -.f$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-GET (with $\hat{\gamma}' = \hat{\gamma}''$):

$$\frac{\Gamma \,.\, \hat{\gamma}'' \vdash e : u \qquad fieldsOf(u)(f) = s}{\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

If u' is read-only, then because $u' \preccurlyeq u$, the definition of subtyping says that u is also read-only. By the first hypothesis of T-CLASS and the definition of field lookup, $fieldsOf(u')(f) = fieldsOf(u)(f)$. Thus, by T-GET, $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

On the other hand, if u' is not read-only, then $fieldsOf(u')(f) = s'$, where $\delta\ s' = s$ for some δ, and by T-GET, $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s'$. Now $s' \preccurlyeq s$, so the claim holds.

Case 5—$\mathbb{E}_2 = cast\ \delta\ S\langle\gamma_1, \ldots, \gamma_n\rangle\ -$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-CAST (with $\hat{\gamma}' = \hat{\gamma}''$):

$$\frac{\Gamma \,.\, \hat{\gamma}'' \vdash e : u \qquad readonly(u) = \delta' \qquad \forall i \in \{1..n\} \cdot \Gamma(\gamma_i) = \mathsf{domain}}{\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

where $s = \delta\ \delta'\ S\langle\gamma_1, \ldots, \gamma_n\rangle$. By assumption $\Gamma \,.\, \hat{\gamma}'' \vdash e' : u'$, $u' \preccurlyeq u$. Let $readonly(u') = \delta''$. By T-CAST $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : \delta\ \delta''\ S\langle\gamma_1, \ldots, \gamma_n\rangle$. We need to show $\delta\ \delta''\ S\langle\gamma_1, \ldots, \gamma_n\rangle \preccurlyeq \delta\ \delta'\ S\langle\gamma_1, \ldots, \gamma_n\rangle$. This follows from $u' \preccurlyeq u$, as argued in Case 5 of the proof of Lemma 4.8 (Substitution) (see page 156).

Because $\Gamma \,.\, \hat{\gamma}'' \vdash e' : u'$, $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$ by T-CAST.

Case 6—$\mathbb{E}_2 = -;\ e''$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-SEQ (with $\hat{\gamma}' = \hat{\gamma}''$):

$$\frac{\Gamma \,.\, \hat{\gamma}'' \vdash e : u \qquad \Gamma \,.\, \hat{\gamma}'' \vdash e'' : s}{\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

Thus, also by T-SEQ, $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

Case 7—$\mathbb{E}_2 = (-.f = e'')$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-SET (with $u = T\langle\gamma_1, \ldots, \gamma_n\rangle$

and $\hat{\gamma}' = \hat{\gamma}''$):

$$\frac{\Gamma \centerdot \hat{\gamma}'' \vdash e : T\langle\gamma_1,\ldots,\gamma_n\rangle \qquad \gamma_1 \in \hat{\gamma}'' \qquad \mathit{fieldsOf}\big(T\langle\gamma_1,\ldots,\gamma_n\rangle\big)(f) = u'' \qquad \Gamma \centerdot \hat{\gamma}'' \vdash e'' : s \qquad s \preccurlyeq u''}{\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

Because $u' \preccurlyeq u$, the definition of subtyping says that u' is read-only and $u' = S\langle\gamma_1,\ldots,\gamma_p\rangle$ where $p \geq n$. By the first hypothesis of T-Class and the definition of field lookup, $\mathit{fieldsOf}(u')(f) = \mathit{fieldsOf}(u)(f)$. Thus, by T-Set, $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

Case 8—$\mathbb{E}_2 = (v_0.f = -)$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-Set, letting $s = u$ and $\hat{\gamma}' = \hat{\gamma}''$:

$$\frac{\Gamma \centerdot \hat{\gamma}'' \vdash v_0 : T\langle\gamma_1,\ldots,\gamma_n\rangle \qquad \gamma_1 \in \hat{\gamma}'' \qquad \mathit{fieldsOf}\big(T\langle\gamma_1,\ldots,\gamma_n\rangle\big)(f) = u'' \qquad \Gamma \centerdot \hat{\gamma}'' \vdash e : u \qquad u \preccurlyeq u''}{\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

Now $u' \preccurlyeq u \preccurlyeq u''$, so let $s' = u'$ and $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s'$.

Case 9—$\mathbb{E}_2 = \langle - \rangle_{\delta,\hat{\gamma}_t}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-Tag:

$$\frac{\Gamma \centerdot \mathit{depClose}(\hat{\gamma}_\mathrm{t}) \vdash e : u \qquad \mathit{depClose}(\hat{\gamma}_\mathrm{t}) \subseteq \hat{\gamma}''}{\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

where, by the assumptions of the lemma, $\hat{\gamma}' = \mathit{depClose}(\hat{\gamma}_\mathrm{t})$ and $s = \delta\ u$. Let $s' = \delta\ u'$, then T-Tag says $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s'$. Because $u' \preccurlyeq u$, $s' \preccurlyeq s$, and the claim holds.

Case 10—$\mathbb{E}_2 = \mathit{joinpt}\ (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}_m|\!)(\ v_0,\ldots,v_{p-1},-,e_{p+1},e_n\ )$ *where* $p \in \{0..n\}$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-Join (where $u_p = u$ and $\hat{\gamma}' = \hat{\gamma}''$):

$$\frac{\begin{array}{c}\forall i \in \{0..(p-1)\} \cdot \Gamma \centerdot \hat{\gamma}'' \vdash v_i : u_i \qquad \Gamma \centerdot \hat{\gamma}'' \vdash e : u \qquad \forall i \in \{(p+1)..n\} \cdot \Gamma \centerdot \hat{\gamma}'' \vdash e_i : u_i \\ \forall i \in \{0..n\} \setminus \{p\} \cdot u_i \preccurlyeq t_i \qquad u_p \preccurlyeq s_p \qquad \mathit{depClose}(\hat{\gamma}_\mathrm{m}) \subseteq \hat{\gamma}'' \\ \big(\mathit{readonly}(u_0) = \mathrm{readonly}\big) \implies (\hat{\gamma}_\mathrm{m} = \emptyset) \qquad \big(v_{opt} = \mathit{loc}_\delta\big) \implies (\mathit{loc} \in \mathit{dom}(\Gamma))\end{array}}{\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

Now $u' \preccurlyeq u = u_p \preccurlyeq s_p$. If $p = 0$, then $\big(\mathit{readonly}(u') = \mathrm{readonly}\big) \implies \big(\mathit{readonly}(u_0) = \mathrm{readonly}\big) \implies (\hat{\gamma}_\mathrm{m} = \emptyset)$. So, also by T-Join, $\Gamma \centerdot \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : s$.

Case 11—$\mathbb{E}_2 = \mathit{under}\ -$. The proof for this case is immediate from T-Under with $s = u$ and $s' = u'$.

Case 12—$\mathbb{E}_2 = \mathit{chain}\ \bar{B}, j(\ v_0,\ldots,v_{p-1},-,e_{p+1},e_n\ )$ *where* $p \in \{0..n\}$. The proof is like that for Case 10, but using T-Chain instead of T-Join. The additional hypotheses of T-Chain, beyond those of T-Join, are unchanged in the type derivations for $\mathbb{E}_2[e]$ and $\mathbb{E}_2[e']$. □

The Environment Subtyping lemma also holds the set of writable concern domains and the dependency table constant.

Lemma 4.13 (Environment Subtyping). *Let* $\Gamma, var : t \,.\, \hat{\gamma} \vdash_{DT} e : s$. *Then for all* $t' \preccurlyeq t$, *there exists some* $s' \preccurlyeq s$ *such that,* $\Gamma, var : t' \,.\, \hat{\gamma} \vdash_{DT} e : s'$.

Proof. Let var' be a variable reference such that $var' \notin dom(\Gamma)$, $var' \neq var$, and var' is not free in e. Then by the assumption of the lemma and Lemma 4.9 (Environment Extension) on page 157, $\Gamma, var' : t', var : t \,.\, \hat{\gamma} \vdash_{DT} e : s$. By Lemma 4.8 (Substitution) on page 154, $\Gamma, var' : t' \,.\, \hat{\gamma} \vdash_{DT} e\{\!|var' / var|\!\} : s'$ for some $s' \preccurlyeq s$. Finally, by α-converting var' to var (relying on the correspondence of α-conversion with capture avoiding substitution of one variable reference for another), we have $\Gamma, var : t' \,.\, \hat{\gamma} \vdash_{DT} e : s'$ for some $s' \preccurlyeq s$. □

The Binding Soundness lemma now handles sets of writable concern domains in join point abstractions and advice body tuples. A new consequent asserts that the dependency closure of the writable domains of matching advice is a subset of the dependency closure of the writable domains of the matched join point abstraction. I suspect that the two sets could be proven equal, but the given claim is strong enough for the use of the lemma in the subject reduction proof. The last consequent of the lemma is also updated to include a set of writable concern domains: the dependency closure of the set from the advice declaration.

Lemma 4.14 (Binding Soundness). *Let* P *be a well-typed program with evaluation dependency table* DT. *Let* S *be a valid store for* P *and* $J = (\!|\ldots, (t_0 \times \ldots \times t_n \to t), \hat{\gamma}|\!) + J'$ *be a stack consistent with* S. *If* $\bar{B} = adviceBind(J, S)$, *then* $\forall [\![b, loc, e, \hat{\gamma}', \tau, \tau']\!] \in \bar{B}$ *the following conditions hold:*

Consequent 1. $depClose_{DT}(\hat{\gamma}') \subseteq depClose_{DT}(\hat{\gamma})$

Consequent 2. $\tau' = t_0 \times \ldots \times t_n \to t$

Consequent 3. $\emptyset \vdash b\ OK$

Consequent 4. For concern-complete $\Gamma \approx S$, *the judgment*

$$\Gamma, this : \Gamma(loc), proceed : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \,.\, depClose_{DT}(\hat{\gamma}') \vdash_{DT} e : t'$$

holds for some $t' \preccurlyeq t$.

Proof. I will use a common setup and some common meta-variables throughout the proof.

Pick an arbitrary element of $\bar{B}$, $[\![b, loc, e, \hat{\gamma}', \tau, \tau']\!]$. Let the advice corresponding to $[\![b, loc, e, \hat{\gamma}', \tau, \tau']\!]$ be

$$s''\ \text{around}(\ s''_1\ var_1, \ldots, s''_p\ var_p\ )\ \text{writes}\ \langle \gamma''_1, \ldots, \gamma''_r \rangle\ :\ pcd''\ \{\ e''\ \}$$

with advice table entry $\langle loc, pcd, e, \hat{\gamma}', \tau, \tau' \rangle$. Let this advice be declared in an aspect a with concern domain variables $G_1, \ldots, G_{q'}$ and dependency declarations $dep'_1, \ldots, dep'_x$. Let $S(loc) = \left[a\langle g_1, \ldots, g_{q'} \rangle \,.\, F \right]$. We will consider the typing derivation for this advice, which must exist because the program is well typed. However, we will α-convert the entire derivation, replacing G_i with g_i for all $i \in \{1..q'\}$.[3]

To simplify the notation, I will write $\{\!|\bar{g} / \bar{G}|\!\}$ for $\{\!|g_1 / G_1, \ldots, g_{q'} / G_{q'}|\!\}$. Let $s = s''\{\!|\bar{g} / \bar{G}|\!\}$, $\forall i \in \{1..p\} \cdot s_i = s''_i\{\!|\bar{g} / \bar{G}|\!\}$, $\forall i \in \{1..r\} \cdot \gamma'_i = \gamma''_i\{\!|\bar{g} / \bar{G}|\!\}$, $\forall i \in \{1..x\} \cdot dep_i = dep'_i\{\!|\bar{g} / \bar{G}|\!\}$, and $\Gamma' = var_1 : s_1, \ldots, var_p : s_p, g_1 :$

[3] This is an α-conversion at the meta-level. One might also think of it as sort of a β-conversion, replacing concern domain variables with concern domain names, if one were so inclined.

Meta-variable Bindings:

$$
\begin{gathered}
[\![b, loc, e, \hat{\gamma}', \tau, \tau']\!] \in \bar{B} \\
S(loc) = \left[a\langle g_1, \ldots, g_{q'} \rangle \centerdot F \right] \\
\tau = s_1 \times \ldots \times s_p \to s \\
\tau' = u_0 \times \ldots \times u_q \to u \\
\Gamma' = var_1 : s_1, \ldots, var_p : s_p, g_1 : \text{domain}, \ldots, g_{q'} : \text{domain}
\end{gathered}
$$

Advice Type Derivation (with domains reified):

$$
\frac{\begin{array}{c}
\Gamma' \vdash pcd : \_ \centerdot u_0 \centerdot \langle u_1, \ldots, u_q \rangle \centerdot u \centerdot \hat{\gamma}_{\text{pcd}} \centerdot V \centerdot V \qquad V = \{var_1, \ldots, var_p\} \\
\hat{\gamma}_{\text{pcd}} \subseteq \hat{\gamma}' \qquad \hat{\gamma}' \subseteq depClose_{DT_a}\left(\hat{\gamma}_{\text{pcd}}\right) \qquad \Gamma', \text{this} : a\langle g_1, \ldots, g_{q'} \rangle, \text{proceed} : (u_0 \times \ldots \times u_q \to u) \centerdot \hat{\gamma}' \vdash_{DT_a} e : s' \\
s' \preccurlyeq s \preccurlyeq u \qquad g_1, \ldots, g_{q'} \vdash s \text{ OK in } a\langle g_1, \ldots, g_{q'} \rangle \\
\forall i \in \{1..r\} \cdot \gamma'_i \in \left\{ g_1, \ldots, g_{q'} \right\} \qquad \forall i \in \{1..p\} \cdot \gamma'_1, \ldots, \gamma'_r \vdash s_i \text{ OK in } a\langle g_1, \ldots, g_{q'} \rangle
\end{array}}{DT_a \vdash s \text{ around}(\, s_1\ var_1, \ldots, s_p\ var_p \,) \text{ writes } \langle \gamma'_1, \ldots, \gamma'_r \rangle : pcd\ \{\ e\ \}\ \text{ OK in } a\langle g_1, \ldots, g_{q'} \rangle}
$$

Figure 4.22 Setup and Common Meta-variable Bindings Used in the Proof of Lemma 4.14

domain,…,$g_{q'}$: domain. By the construction of AT, $\tau = s_1 \times \ldots \times s_p \to s$, $e = e''\{\!|\bar{g}/\bar{G}|\!\}$, $pcd = pcd''\{\!|\bar{g}/\bar{G}|\!\}$, and $\hat{\gamma}' = \{\gamma'_1, \ldots, \gamma'_r\}$. Let the dependency table of the advice typing be

$$DT_a = depTable\left(\left\{ g_1, \ldots, g_{q'} \right\}, \{dep_1, \ldots, dep_x\}\right).$$

This comes from T-ASP, with concern domain variables replaced by concern domain names.

Plugging this notation into the α-converted derivation from T-ADV gives:

$$
\frac{\begin{array}{c}
\Gamma' \vdash pcd : \_ \centerdot u_0 \centerdot \langle u_1, \ldots, u_q \rangle \centerdot u \centerdot \hat{\gamma}_{\text{pcd}} \centerdot V \centerdot V \qquad V = \{var_1, \ldots, var_p\} \\
\hat{\gamma}_{\text{pcd}} \subseteq \hat{\gamma}' \qquad \hat{\gamma}' \subseteq depClose_{DT_a}\left(\hat{\gamma}_{\text{pcd}}\right) \qquad \Gamma', \text{this} : a\langle g_1, \ldots, g_{q'} \rangle, \text{proceed} : (u_0 \times \ldots \times u_q \to u) \centerdot \hat{\gamma}' \vdash_{DT_a} e : s' \\
s' \preccurlyeq s \preccurlyeq u \qquad g_1, \ldots, g_{q'} \vdash s \text{ OK in } a\langle g_1, \ldots, g_{q'} \rangle \\
\forall i \in \{1..r\} \cdot \gamma'_i \in \left\{ g_1, \ldots, g_{q'} \right\} \qquad \forall i \in \{1..p\} \cdot \gamma'_1, \ldots, \gamma'_r \vdash s_i \text{ OK in } a\langle g_1, \ldots, g_{q'} \rangle
\end{array}}{DT_a \vdash s \text{ around}(\, s_1\ var_1, \ldots, s_p\ var_p \,) \text{ writes } \langle \gamma'_1, \ldots, \gamma'_r \rangle : pcd\ \{\ e\ \}\ \text{ OK in } a\langle g_1, \ldots, g_{q'} \rangle} \tag{4.2}
$$

By the construction of AT, $\tau' = u_0 \times \ldots \times u_q \to u$.

For convenience, Figure 4.22 summarizes the setup of the proof and the use of these meta-variables.

Because a well-typed pointcut descriptor in MiniMAO$_2$ must consist of multiple primitive pointcut descriptors, it is difficult to prove the consequents of the lemma using a single inductive argument. Instead, I 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 facing page relates the writable domains recorded in the join point abstraction to

those recorded in the advice body tuple. We know that $\hat{\gamma}_{\text{pcd}} \subseteq \hat{\gamma}'$ by an hypothesis of T-ADV in (4.2). By the definition of *adviceBind*, $[\![b, loc, e, \hat{\gamma}', \tau, \tau']\!] \in \bar{B}$ implies $matchPCD(J, pcd, S) \neq \bot$. By T-ADV, the writable domains slot of the pointcut type for *pcd* is not $\bot$. The following subclaim says that in this situation $\hat{\gamma}' \subseteq depClose(\hat{\gamma})$. Thus by Lemma 4.6 (Dependency Closure Inclusion) on page 153, $depClose(\hat{\gamma}') \subseteq depClose(\hat{\gamma})$ and consequent 1 holds.

Subclaim 1. Assume $\Gamma' \vdash pcd : \hat{u} \,.\, \hat{u}_0 \,.\, U \,.\, \hat{u}' \,.\, \hat{\gamma}_{\text{pcd}} \,.\, V' \,.\, V''$ (i.e., $\hat{\gamma}_{\text{pcd}} \neq \bot$). Then

$$matchPCD(J, pcd, S) \neq \bot \implies \hat{\gamma}' \subseteq depClose(\hat{\gamma})$$

Proof of subclaim.

— $pcd = \text{call}(\ldots)$. Subclaim assumption cannot hold (because $\hat{\gamma}_{\text{pcd}} = \bot$).

— $pcd = \text{execution}(\ldots)$. Subclaim assumption cannot hold.

— $pcd = \text{writes}(\gamma_1, \ldots, \gamma_w)$. By T-WRTPCD, $\hat{\gamma}_{\text{pcd}} = \{\gamma_1, \ldots, \gamma_w\}$. By the definition of *matchPCD*,

$$matchPCD(J, pcd, S) \neq \bot \implies \hat{\gamma} = \hat{\gamma}_{\text{pcd}}$$

To see that the subclaim holds, let γ' be an arbitrary element of $\hat{\gamma}'$.

If $\gamma' \in \hat{\gamma}_{\text{pcd}}$, then $\gamma' \in \hat{\gamma}$ and, by definition of *depClose* and the fact that *DT* is reflexive, $\gamma' \in depClose(\hat{\gamma})$.

On the other hand, if $\gamma' \notin \hat{\gamma}_{\text{pcd}}$ then, by the *depClose* hypothesis of T-ADV in (4.2) on the preceding page, there exists $\gamma \in \hat{\gamma}_{\text{pcd}}$ such that $(\gamma, \gamma') \in DT_a$. But $\gamma \in \hat{\gamma}_{\text{pcd}}$ implies that $\gamma \in \hat{\gamma}$. By the construction of DT_a (in T-ASP) and Definition 4.1 (Evaluation Dependency Table) on page 140, $(\gamma, \gamma') \in DT$. So by the definition of *depClose*, $\gamma' \in depClose(\hat{\gamma})$.

Thus $\hat{\gamma}' \subseteq depClose(\hat{\gamma})$.

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

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

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

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

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ or } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies \hat{\gamma}' \subseteq depClose(\hat{\gamma}) \end{aligned}$$

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

- $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}_1' \,.\, U_1 \,.\, \hat{u}_1'' \,.\, \hat{\gamma}_{\text{pcd}} \,.\, V_1 \,.\, V_1'$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}_2' \,.\, U_2 \,.\, \hat{u}_2'' \,.\, \bot \,.\, V_2 \,.\, V_2'$
- $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}_1' \,.\, U_1 \,.\, \hat{u}_1'' \,.\, \bot \,.\, V_1 \,.\, V_1'$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}_2' \,.\, U_2 \,.\, \hat{u}_2'' \,.\, \hat{\gamma}_{\text{pcd}} \,.\, V_2 \,.\, V_2'$

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

$$\begin{aligned} matchPCD(J, pcd, S) \neq \bot &\implies matchPCD(J, pcd_1, S) \neq \bot \text{ and } matchPCD(J, pcd_2, S) \neq \bot \\ &\implies \hat{\gamma}' \subseteq depClose(\hat{\gamma}) \end{aligned}$$

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

Subclaim-□

Consequent 2 on page 162 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, $pcd : \_ \,.\, u_0 \,.\, \langle u_1, \ldots, u_q \rangle \,.\, \hat{\gamma}_{\text{pcd}} \,.\, V \,.\, V$. 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 2. Assume $\Gamma' \vdash pcd : \hat{u} \,.\, u_0 \,.\, U \,.\, \hat{u}' \,.\, \hat{\gamma}_\perp \,.\, V' \,.\, V''$ (i.e., the "target type" is not $\perp$). Then

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

Proof of subclaim.

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

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

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

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

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

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

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

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UnionPCD, $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, u_0 \,.\, U_1 \,.\, \hat{u}'_1 \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, u_0 \,.\, U_2 \,.\, \hat{u}'_2 \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2$. By the induction hypothesis, $matchPCD(J, pcd_1, S) \neq \perp \implies u_0 = t_0$ and $matchPCD(J, pcd_2, S) \neq \perp \implies u_0 = t_0$. By the definition of *matchPCD*,

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

— $pcd = pcd_1 \ \&\&\ pcd_2$. By T-IntPCD and the definition of $\sqcup$ (in Figure 4.20 on page 150), one of the following hold:

  – $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, u_0 \,.\, U_1 \,.\, \hat{u}'_1 \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \perp \,.\, U_2 \,.\, \hat{u}'_2 \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2$
  – $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \perp \,.\, U_1 \,.\, \hat{u}'_1 \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, u_0 \,.\, U_2 \,.\, \hat{u}'_2 \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2$

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

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

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

Subclaim-□

Subclaim 3. Assume $\Gamma' \vdash pcd : \hat{u} \centerdot \hat{u}' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}'' \centerdot \hat{\gamma}_\bot \centerdot V' \centerdot V''$ (i.e., the argument type sequence is not $\bot$). Then

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

Proof of subclaim.

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

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

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

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

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

— $pcd = \mathsf{args}(\ t_1'' \ var_1'', \ldots, t_w'' \ var_w'' \ )$. 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, $\Gamma' \vdash pcd_1 : \hat{u}_1 \centerdot \hat{u}_1' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}_1'' \centerdot \hat{\gamma}_{\bot_1} \centerdot V_1 \centerdot V_1'$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \centerdot \hat{u}_2' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}_2'' \centerdot \hat{\gamma}_{\bot_2} \centerdot V_2 \centerdot V_2'$. 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:

- $\Gamma' \vdash pcd_1 : \hat{u}_1 \centerdot \hat{u}_1' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}_1'' \centerdot \hat{\gamma}_{\bot_1} \centerdot V_1 \centerdot V_1'$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \centerdot \hat{u}_2' \centerdot \bot \centerdot \hat{u}_2'' \centerdot \hat{\gamma}_{\bot_2} \centerdot V_2 \centerdot V_2'$
- $\Gamma' \vdash pcd_1 : \hat{u}_1 \centerdot \hat{u}_1' \centerdot \bot \centerdot \hat{u}_1'' \centerdot \hat{\gamma}_{\bot_1} \centerdot V_1 \centerdot V_1'$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \centerdot \hat{u}_2' \centerdot \langle u_1, \ldots, u_q \rangle \centerdot \hat{u}_2'' \centerdot \hat{\gamma}_{\bot_2} \centerdot V_2 \centerdot V_2'$

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 4. Assume $\Gamma' \vdash pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, u \,.\, \hat{\gamma}_\perp \,.\, V' \,.\, V''$ (i.e., the "return type" is not $\perp$). Then

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

Proof of subclaim.

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

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

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

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

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

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

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

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UNIONPCD, $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, u \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, u \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2$. By the induction hypothesis, $matchPCD(J, pcd_1, S) \neq \perp \implies u = t$ and $matchPCD(J, pcd_2, S) \neq \perp \implies u = t$. By the definition of *matchPCD*,

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

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

- $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, u \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, \perp \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2$
- $\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, \perp \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1$ and $\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, u \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2$

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

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

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

Subclaim-□

With these three subclaims we can now prove consequent 2 on page 162. The first hypothesis of T-ADV (see (4.2) on page 163) is:

$$\Gamma' \vdash pcd : \_ \,.\, u_0 \,.\, \langle u_1, \dots, u_q \rangle \,.\, u \,.\, \hat{\gamma}_{\mathrm{pcd}} \,.\, V \,.\, V$$

Thus, the target type is not $\perp$, 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*, $[\![b, loc, e, \hat{\gamma}', \tau, \tau']\!] \in \bar{B}$

implies $matchPCD(J, pcd, S) \neq \bot$. Thus:

$$\begin{aligned}
\tau' &= u_0 \times \ldots \times u_q \to u && \text{by construction of } AT \\
&= t_0 \times u_1 \times \ldots \times u_q \to u && \text{by Subclaim 2} \\
&= t_0 \times t_1 \times \ldots \times t_n \to u && \text{by Subclaim 3} \\
&= t_0 \times \ldots \times t_n \to u \\
&= t_0 \times \ldots \times t_n \to t && \text{by Subclaim 4}
\end{aligned}$$

We next turn to consequent 3 on page 162. 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 4. In the subclaim, $var(b)$ means all variables appearing in b (as defined in Figure 4.15 on page 144).

Subclaim 5. Assume $\Gamma' \vdash pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, \hat{u}'' \,.\, \hat{\gamma}_\bot \,.\, V' \,.\, V''$. Then $matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle$ implies all of the following:

$$\emptyset \vdash b \text{ OK} \tag{4.3a}$$

$$V' \subseteq var(b) \subseteq V'' \tag{4.3b}$$

$$\hat{u} = \bot \iff \alpha = - \tag{4.3c}$$

$$\hat{u}' = \bot \iff \beta_0 = - \tag{4.3d}$$

$$U = \bot \implies x = 0 \tag{4.3e}$$

$$U \neq \bot \implies x = n \tag{4.3f}$$

$$U = \bot \iff \forall i \in \{1..x\} \cdot \beta_i = - \tag{4.3g}$$

Proof of subclaim.

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

$$\begin{aligned}
matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, - \rangle \\
&\implies \emptyset \vdash b \text{ OK} \\
&\quad V' = \emptyset \subseteq var(b) \subseteq \emptyset = V'' \\
&\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (4.3c) holds} \\
&\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{so (4.3d) holds} \\
&\quad U = \bot \text{ and } x = 0 \text{ so (4.3e) holds} \\
&\quad U = \bot \text{ so (4.3f) holds} \\
&\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (4.3g) holds}
\end{aligned}$$

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

— $pcd = \mathsf{writes}(\ \ldots\ )$. Similar to the first case, but by T-WRTPCD.

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

matchPCD,

$$\begin{aligned}
\mathit{matchPCD}(J, \mathit{pcd}, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle \mathit{var}'' \mapsto v, - \rangle \text{ for some } v \in \mathcal{V} \\
&\implies \emptyset \vdash b \text{ OK} \\
&\quad V' = \{\mathit{var}''\} \subseteq \mathit{var}(b) \subseteq \{\mathit{var}''\} = V'' \\
&\quad \hat{u} \neq \bot \text{ and } \alpha \neq - \text{ so (4.3c) holds} \\
&\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{ so (4.3d) holds} \\
&\quad U = \bot \text{ and } x = 0 \text{ so (4.3e) holds} \\
&\quad U = \bot \text{ so (4.3f) holds} \\
&\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (4.3g) holds}
\end{aligned}$$

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

$$\begin{aligned}
\mathit{matchPCD}(J, \mathit{pcd}, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, \mathit{var}'' \rangle \\
&\implies \emptyset \vdash b \text{ OK} \\
&\quad V' = \{\mathit{var}''\} \subseteq \mathit{var}(b) \subseteq \{\mathit{var}''\} = V'' \\
&\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (4.3c) holds} \\
&\quad \hat{u}' \neq \bot \text{ and } \beta_0 \neq - \text{ so (4.3d) holds} \\
&\quad U = \bot \text{ and } x = 0 \text{ so (4.3e) holds} \\
&\quad U = \bot \text{ so (4.3f) holds} \\
&\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (4.3g) holds}
\end{aligned}$$

— $\mathit{pcd} = \text{args}(\ t_1''\ \mathit{var}_1'', \ldots, t_w''\ \mathit{var}_w''\ )$. By T-ArgsPCD, $\Gamma' \vdash \mathit{pcd} : \bot \,.\, \bot \,.\, \langle t_1'', \ldots, t_w'' \rangle \,.\, \bot \,.\, \bot \,.\, V' \,.\, V''$ where $V' = V'' = \{\mathit{var}_1'', \ldots, \mathit{var}_w''\}$, and all var_i'' are unique. By the definition of *matchPCD*,

$$\begin{aligned}
\mathit{matchPCD}(J, \mathit{pcd}, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, -, \mathit{var}_1'', \ldots, \mathit{var}_w'' \rangle \\
&\implies \emptyset \vdash b \text{ OK} \\
&\quad V' \subseteq \mathit{var}(b) \subseteq V'' \\
&\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (4.3c) holds} \\
&\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{ so (4.3d) holds} \\
&\quad U \neq \bot \text{ so (4.3e) holds} \\
&\quad U \neq \bot \text{ and } x = w = n \text{ by Subclaim 3, so (4.3f) holds} \\
&\quad U \neq \bot \text{ and } \exists i \in \{1..0\} \cdot \beta_i \neq - \text{ so (4.3g) holds}
\end{aligned}$$

— $\mathit{pcd} = \mathit{pcd}_1 \ ||\ \mathit{pcd}_2$. By T-UnionPCD, let

$$\begin{aligned}
\Gamma' &\vdash \mathit{pcd}_1 : \hat{u}_1 \,.\, \hat{u}_1' \,.\, U_1 \,.\, \hat{u}_1'' \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V_1' \\
\Gamma' &\vdash \mathit{pcd}_2 : \hat{u}_2 \,.\, \hat{u}_2' \,.\, U_2 \,.\, \hat{u}_2'' \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V_2'
\end{aligned}$$

Also let $\mathit{matchPCD}(J, \mathit{pcd}_1, S) = r_1$ and $\mathit{matchPCD}(J, \mathit{pcd}_2, S) = r_2$.

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:

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

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

$$\begin{aligned} \Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}_1' \,.\, U_1 \,.\, \hat{u}_1'' \,.\, \hat{\gamma}_{\bot_1} \,.\, V_1 \,.\, V_1' \\ \Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}_2' \,.\, U_2 \,.\, \hat{u}_2'' \,.\, \hat{\gamma}_{\bot_2} \,.\, V_2 \,.\, V_2' \end{aligned}$$

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

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

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 (4.3c) holds.

– Similarly, $\hat{u}' = - \iff \beta_0 = -$, and (4.3d) 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 $\big(U = - \iff \forall i \in \{1..x\} \cdot \beta_i = -\big)$. Thus, (4.3e), (4.3f), and (4.3g) 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, (4.3a) 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 (4.3b) holds.

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

$$\begin{aligned} matchPCD(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle &\implies b = \langle -, - \rangle \\ &\implies \emptyset \vdash b \text{ OK} \\ &\quad V' = \emptyset \subseteq var(b) \subseteq \emptyset = V'' \\ &\quad \hat{u} = \bot \text{ and } \alpha = - \text{ so (4.3c) holds} \\ &\quad \hat{u}' = \bot \text{ and } \beta_0 = - \text{so (4.3d) holds} \\ &\quad U = \bot \text{ and } x = 0 \text{ so (4.3e) holds} \\ &\quad U = \bot \text{ so (4.3f) holds} \\ &\quad U = \bot \text{ and } \forall i \in \{1..0\} \cdot \beta_i = - \text{ vacuously, so (4.3g) holds} \end{aligned}$$

Subclaim-□

By T-ADV, the assumption of the subclaim holds. Therefore, consequent 3 on page 162 holds by (4.3a).

Consequent 4 is more complex. To prove this consequent, it will suffice to show that

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s'_1, \ldots, var_p : s'_p \text{ where } \forall i \in \{1..p\} \cdot s'_i \preccurlyeq s_i \tag{4.4}$$

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 to this end.

Subclaim 6. Assume $\Gamma' \vdash pcd : \hat{u}.\hat{u}'.U.\hat{u}''.\hat{\gamma}_\bot.V'.V''$, where $V'' \subseteq \{var_1, \ldots, var_p\}$. Then

$$\begin{aligned} &matchPCD(J, pcd, S) = b \neq \bot \\ &\implies \forall var \in var(b) \cdot \left(\exists i \in \{1..p\}, s'_i \in \mathcal{T} \cdot \left(var = var_i,\ typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_i) = s'_i, \text{ and } s'_i \preccurlyeq s_i\right)\right) \end{aligned}$$

Proof of subclaim. The assumption of this subclaim implies the assumption for Subclaim 5 on page 168; we will make free use of the earlier result.

— $pcd = \mathsf{call}(\ \ldots\ )$. By T-CALLPCD, $V' = V'' = \emptyset$. By (4.3b) on page 168, $matchPCD(J, pcd, S) = b \neq \bot$ implies $var(b) = \emptyset$, satisfying the subclaim.

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

— $pcd = \mathsf{writes}(\ \ldots\ )$. Similar to the first case, but by T-WRTPCD.

— $pcd = \mathsf{this}(\ t''\ var''\ )$. 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', $t'' = s_1$.

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = \langle var_1 \mapsto loc'_\delta, - \rangle$$

for some loc'_δ in J, where

$$\begin{aligned} &loc' \in dom(S) \text{ by } J \approx S, \\ &S\left(loc'\right) = [s'_1 \,.\, F], \delta\ s'_1 \preccurlyeq s_1, \text{ by definition of } matchPCD, \text{ and} \\ &\Gamma\left(loc'\right) = s'_1 \text{ by } \Gamma \approx S. \end{aligned}$$

Thus,

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : \delta\ s'_1 \text{ where } \delta\ s'_1 \preccurlyeq s_1.$$

— 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 $var''_1, \ldots, t''_w$ 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,

$$\forall\, i \in \{1..w\} \cdot \left(\exists! j \in \{1..p\} \cdot \left(t''_i = s_j \text{ and } var''_i = var_j\right)\right) \tag{4.5}$$

The definition of *matchPCD* gives

$$matchPCD(J, pcd, S) = b \neq \bot \implies b = \left\langle -, -, var''_1, \ldots, var''_w \right\rangle$$

where $n = w$ and $\forall\, i \in \{1..w\} \cdot \left(t''_i = t_i\right)$. 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

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)\left(var''_1\right) = t''_1.$$

By (4.5), there exists j such that $var''_1 = var_j$ and $t''_1 = s_j$, thus the subclaim holds.

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

$$\begin{aligned} &\Gamma' \vdash pcd_1 : \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, \hat{u}''_1 \,.\, \hat{\gamma}_{\perp_1} \,.\, V_1 \,.\, V'_1 && matchPCD(J, pcd_1, S) = r_1 \\ &\Gamma' \vdash pcd_2 : \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, \hat{u}''_2 \,.\, \hat{\gamma}_{\perp_2} \,.\, V_2 \,.\, V'_2 && matchPCD(J, pcd_2, S) = r_2 \end{aligned}$$

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 5, $V_1' \subseteq V''$ 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

$$\Gamma' \vdash pcd_1 : \hat{u}_1 \centerdot \hat{u}_1' \centerdot U_1 \centerdot \hat{u}_1'' \centerdot \hat{\gamma}_{\perp_1} \centerdot V_1 \centerdot V_1' \qquad matchPCD(J, pcd_1, S) = r_1$$
$$\Gamma' \vdash pcd_2 : \hat{u}_2 \centerdot \hat{u}_2' \centerdot U_2 \centerdot \hat{u}_2'' \centerdot \hat{\gamma}_{\perp_2} \centerdot V_2 \centerdot V_2' \qquad matchPCD(J, pcd_2, S) = r_2$$

By the definition of *matchPCD*,

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

As argued in the corresponding case of Subclaim 5, $var(r_1)$ 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, $V' = V'' = \emptyset$.

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

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 (4.2) on page 163) are:

$$\Gamma' \vdash pcd : \_ \centerdot u_0 \centerdot \langle u_1, \ldots, u_q \rangle \centerdot u \centerdot \hat{\gamma}_{\text{pcd}} \centerdot V \centerdot V$$
$$V = \{var_1, \ldots, var_p\}$$

By definition of *adviceBind*, $[\![b, loc, e, \hat{\gamma}', \tau, \tau']\!] \in \bar{B}$ implies $matchPCD(J, pcd, S) \neq \perp$. We first use Subclaim 5 and Subclaim 6 to prove equation (4.4) from page 172.

$$\begin{aligned} V = \{var_1, \ldots, var_p\} & && \text{by T-Adv} \\ \implies var(b) = \{var_1, \ldots, var_p\} & && \text{by (4.3b)} \\ \implies \forall i \in \{1..p\} \cdot \exists s_i' \in \mathscr{T} \cdot & \\ \quad \left(typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_i) = s_i',\ s_i' \preccurlyeq s_i \right) & && \text{by Subclaim 6} \end{aligned}$$

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

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

Thus, no additional variables are bound and (4.4) on page 172 holds:

$$\mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = \mathit{var}_1 : s'_1, \ldots, \mathit{var}_p : s'_p \text{ where } \forall\, i \in \{1..p\} \cdot s'_i \preccurlyeq s_i$$

T-ADV gives:

$$\begin{array}{l}
\mathit{var}_1 : s_1, \ldots, \mathit{var}_p : s_p, \mathsf{this} : a\langle g_1, \ldots, g_{q'} \rangle, \mathsf{proceed} : \tau', g_1 : \mathsf{domain}, \ldots, g_{q'} : \mathsf{domain} \,.\, \hat{\gamma}' \vdash_{DT_a} e : s' \\
\quad \Longrightarrow \qquad \text{by Lemma 4.13} \\
\quad \mathit{var}_1 : s'_1, \ldots, \mathit{var}_p : s'_p, \mathsf{this} : a\langle g_1, \ldots, g_{q'} \rangle, \mathsf{proceed} : \tau', g_1 : \mathsf{domain}, \ldots, g_{q'} : \mathsf{domain} \,.\, \hat{\gamma}' \vdash_{DT_a} e : s'' \\
\qquad \text{where } s'' \preccurlyeq s' \text{ and } \forall\, i \in \{1..p\} \cdot s'_i \preccurlyeq s_i \\
\quad \Longrightarrow \qquad \text{by (4.4)} \\
\quad \mathsf{this} : a\langle g_1, \ldots, g_{q'} \rangle, \mathsf{proceed} : \tau', \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle), g_1 : \mathsf{domain}, \ldots, g_{q'} : \mathsf{domain} \,.\, \hat{\gamma}' \vdash_{DT_a} e : s'' \\
\quad \Longrightarrow \qquad \text{by Lemma 4.9, with appropriate } \alpha\text{-conversion of } b \text{ and } e \\
\quad \Gamma, \mathsf{this} : a\langle g_1, \ldots, g_{q'} \rangle, \mathsf{proceed} : \tau', \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle), g_1 : \mathsf{domain}, \ldots, g_{q'} : \mathsf{domain} \,.\, \hat{\gamma}' \vdash_{DT_a} e : s'' \\
\quad \Longrightarrow \qquad \text{by concern-completeness of } \Gamma,\ \forall\, i \in \{1..q'\} \cdot \Gamma(g_i) = \mathsf{domain} \\
\quad \Gamma, \mathsf{this} : a\langle g_1, \ldots, g_{q'} \rangle, \mathsf{proceed} : \tau', \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \,.\, \hat{\gamma}' \vdash_{DT_a} e : s''
\end{array}$$

By the definition of evaluation dependency table (see Definition 4.1 (Evaluation Dependency Table) on page 140), $DT_a \subseteq DT$ and $\forall \gamma \in \hat{\gamma}' \cdot (\gamma, \gamma) \in DT$. The advice body expression e contains only user syntax, by the construction of AT. Thus Lemma 4.7 (Dependency Table Extension) on page 153 gives:

$$\Gamma, \mathsf{this} : a\langle g_1, \ldots, g_{q'} \rangle, \mathsf{proceed} : \tau', \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \,.\, \mathit{depClose}_{DT}(\hat{\gamma}') \vdash_{DT} e : s''$$

Another hypothesis of T-ADV gives $s' \preccurlyeq s \preccurlyeq u$. By transitivity of subtyping $s'' \preccurlyeq u = t$. Thus the final consequent holds. □

The Advice Chaining lemma handles sets of writable concern domains in join point abstractions and advice body tuples. The basic claim—replacing all proceed subexpressions in a well-typed expression with appropriate chain expressions does not change the expression's type—remains unchanged.

Lemma 4.15 (Advice Chaining). *Let* $\Gamma, \mathit{proceed} : \tau \,.\, \hat{\gamma} \vdash_{DT} e : t$, $j = (\!|\_, \_, \_, \_, \tau, \hat{\gamma}''|\!)$, $\tau = t_0 \times \ldots \times t_n \to t$, $\big(\mathit{readonly}(t_0) = \mathit{readonly}\big) \implies \big(\hat{\gamma}'' = \emptyset\big)$, *and for all* $[\![b, \mathit{loc}, e', \hat{\gamma}', \tau', \tau]\!] \in \bar{B}$ *let*

— $\Gamma, \mathit{this} : \Gamma(\mathit{loc}), \mathit{proceed} : \tau, \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \,.\, \mathit{depClose}(\hat{\gamma}') \vdash_{DT} e' : s'$,

— $\Gamma \vdash b$ *OK*,

— $\mathit{depClose}(\hat{\gamma}') \subseteq \mathit{depClose}(\hat{\gamma}'') \subseteq \hat{\gamma}$, *and*

— $s' \preccurlyeq t$.

Then $\Gamma \,.\, \hat{\gamma} \vdash_{DT} \langle\!\langle e \rangle\!\rangle_{\bar{B}, j} : t$.

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 4.10 (Environment Contraction) on page 158.

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

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

Assuming that $\Gamma, \text{proceed}:\tau \centerdot \hat{\gamma} \vdash_{\overline{DT}} e : t$, we need to show that $\Gamma \centerdot \hat{\gamma} \vdash_{\overline{DT}} \langle\!\langle e \rangle\!\rangle_{\bar{B},j} : t$. The latter 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:

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

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

$$\forall i \in \{0..n\} \cdot \Gamma \centerdot \hat{\gamma} \vdash_{\overline{DT}} \langle\!\langle e_i \rangle\!\rangle_{\bar{B},j} : u_i \text{ where } u_i \preccurlyeq t_i$$

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

Finally, the Join Point Abstraction lemma also handles sets of writable concern domains.

Lemma 4.16 (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 $j = (\!|\text{exec}, v, m, l, \tau, \hat{\gamma}|\!)$ and $l = \text{fun } m\langle var_0, \ldots, var_n \rangle . e : \tau \centerdot \hat{\gamma}$, or else*
2. *$j = (\!|\text{call}, -, m, -, (t_0 \times \ldots \times t_n \to t), \hat{\gamma}|\!)$, $\mathit{methodType}(t_0, m) = t_1 \times \ldots \times t_n \to t$, and $\mathit{writable}(t_0, m) = \hat{\gamma}$.*

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. □

4.4.2 Type Safety

As for MiniMAO$_1$, I prove the soundness of MiniMAO$_2$'s static type system using the standard subject reduction and progress technique.

I update the statement of the Subject Reduction theorem to consider the public concern domains declared in the program, a set of writable concern domains, and the evaluation dependency table. The proof differs substantially from that for MiniMAO$_1$, as one would expect given the magnitude of the changes to the static semantics. The greatest difficulty is handling the evaluation steps that introduce tagged expressions, EXEC$_\text{B}$ and ADVISE. The typing rule for tagged expressions, T-TAG, uses a different set of writable domains for typing the contained expression. I leverage the various subset relationships on sets of writable concern domains (and their dependency closures) to handle these cases.

Theorem 4.17 (Subject Reduction). *Given a well-typed program P with public concern domains $\hat{g} \subset \mathscr{G}$, for an expression e, a valid store S, a stack J consistent with S, a concern-complete type environment Γ consistent with S, a set of public concern domains $\hat{\gamma} \subseteq \hat{g}$, and the evaluation dependency table, DT, of P, if $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$ and $\langle e, J, S\rangle \hookrightarrow \langle e', J', S'\rangle$, then $J' \approx S'$, S' is valid, and there exist concern-complete $\Gamma' \approx S'$ and $t' \preccurlyeq t$, such that $\Gamma' \centerdot \hat{\gamma} \vdash_{DT} e' : t'$.*

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. By the monotonicity property, S valid implies that part 1 of Definition 4.5 (Store Validity) on page 152 holds for S' in each case of the proof. Based on the reduction step, in each case we demonstrate how to construct a Γ' consistent with S' that witnesses to the validity of S' and satisfies the claim. I omit the dependency table subscript on type judgments and uses of the *depClose* auxiliary function, unless the dependency table used differs from DT.

Case 1—NEW. In this case

$$\begin{aligned}
e &= \mathbb{E}[\mathsf{new}\ c\langle g_1,\ldots,g_n\rangle()] \\
e' &= \mathbb{E}[loc] \\
loc &\notin dom(S) \\
J' &= J \\
S' &= S \oplus \left(loc \mapsto \left[c\langle g_1,\ldots,g_n\rangle \centerdot F\right]\right) \\
F &= \left\{f \mapsto \mathsf{null} \cdot f \in dom\left(fieldsOf\left(c\langle g_1,\ldots,g_n\rangle\right)\right)\right\}.
\end{aligned}$$

Let $\Gamma' = \Gamma, loc : c\langle g_1,\ldots,g_n\rangle$.

By the monotonicity property of the store, $J' = J \implies J' \approx S'$.

We now show that $\Gamma' \approx S'$. Because $loc \notin dom(S)$, $(\Gamma \approx S) \implies loc \notin dom(\Gamma)$ by part 2 of Definition 4.3 (Environment-Store Consistency) on page 152. Thus part 1 of the definition for $\Gamma' \approx S'$ holds for all $loc' \in \mathscr{L}$, $loc' \neq loc$. Now $S'(loc) = \left[c\langle g_1,\ldots,g_n\rangle \centerdot F\right]$, $\Gamma'(loc) = c\langle g_1,\ldots,g_n\rangle$, $dom(F) = dom\left(fieldsOf\left(c\langle g_1,\ldots,g_n\rangle\right)\right)$, $rng(F) = \{\mathsf{null}\} \subseteq dom(S) \cup \{\mathsf{null}\}$, and 1(d) holds vacuously. So part 1 of $\Gamma' \approx S'$ holds. Parts 2 and 3 hold because $\Gamma \approx S$, $loc \in dom\left(\Gamma'\right)$, and $loc \in dom\left(S'\right)$.

We now show that $\Gamma' \centerdot \hat{\gamma} \vdash \mathbb{E}[loc] : t$. By Lemma 4.9 (Environment Extension) on page 157 and $loc \notin dom(\Gamma)$, we have $\Gamma' \centerdot \hat{\gamma} \vdash \mathbb{E}[\mathsf{new}\ c\langle g_1,\ldots,g_n\rangle()] : t$. Now $\Gamma' \centerdot \hat{\gamma} \vdash \mathsf{new}\ c\langle g_1,\ldots,g_n\rangle() : c\langle g_1,\ldots,g_n\rangle$ (by P well-typed and Γ' concern complete) and $\Gamma' \centerdot \hat{\gamma} \vdash loc : c\langle g_1,\ldots,g_n\rangle$, so by Lemma 4.11 (Replacement) on page 158, $\Gamma' \centerdot \hat{\gamma} \vdash \mathbb{E}[loc] : t$.

Case 2—CALL$_A$. Here

$$\begin{aligned}
e &= \mathbb{E}[loc_\delta.m(\ v_1,\ldots,v_n\ )]\\
e' &= \mathbb{E}[\text{joinpt }\langle\!|\text{call},-,m,-,(s_0\times\ldots\times s_n\to s),\hat\gamma'|\!\rangle(\ loc_\delta,v_1,\ldots,v_n\ )]\\
S(loc) &= [u\,.\,F]\\
methodType(s_0,m) &= s_1\times\ldots\times s_n\to s\\
writable(s_0,m) &= \hat\gamma'\\
origType(\delta\ u,m) &= s_0\\
J' &= J\\
S' &= S
\end{aligned}$$

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

We now show that $\Gamma.\hat\gamma \vdash e' : t$. The judgment $\Gamma.\hat\gamma \vdash e : t$ implies that $loc_\delta.m(\ v_1,\ldots,v_n\ )$ and all its subterms are well typed in Γ. Let $\Gamma.\hat\gamma \vdash v_i : t_i$ for all $i \in \{1..n\}$. By part 1(a) of $\Gamma \approx S$ and T-LOC, $\Gamma.\hat\gamma \vdash loc_\delta : \delta\ u$. The type judgment for $loc_\delta.m(\ v_1,\ldots,v_n\ )$ must be by T-CALL with $\forall i \in \{1..n\}\cdot t_i \preccurlyeq s_i$, $\Gamma.\hat\gamma \vdash loc_\delta.m(\ v_1,\ldots,v_n\ ) : s$, $depClose(\hat\gamma') \subseteq \hat\gamma$, and $(\delta = \text{readonly}) \implies (\hat\gamma' = \emptyset)$. By the definition of *origType*, $\delta\ u \preccurlyeq s_0$. T-JOIN gives:[4]

$$\frac{\begin{array}{ll}\Gamma.\hat\gamma \vdash loc_\delta : \delta\ u & \forall i \in \{1..n\}\cdot \Gamma.\hat\gamma \vdash v_i : t_i \qquad depClose(\hat\gamma') \subseteq \hat\gamma\\ \delta\ u \preccurlyeq s_0 & \forall i \in \{1..n\}\cdot t_i \preccurlyeq s_i \qquad (\delta = \text{readonly}) \implies (\hat\gamma' = \emptyset)\end{array}}{\Gamma.\hat\gamma \vdash \text{joinpt }\langle\!|\text{call},-,m,-,(s_0\times\ldots\times s_n\to s),\hat\gamma'|\!\rangle(\ loc_\delta,v_1,\ldots,v_n\ ) : s}$$

Therefore, Lemma 4.11 (Replacement) on page 158 gives $\Gamma.\hat\gamma \vdash e' : t$.

Case 3—CALL$_B$. Here $e = \mathbb{E}[\text{chain }\bullet,\langle\!|\text{call},-,m,-,\tau,\hat\gamma'|\!\rangle(\ loc_\delta,v_1,\ldots,v_n\ )]$, $e' = \mathbb{E}[(\ l\ (\ loc_\delta,v_1,\ldots,v_n\ )\ )]$ (where $S(loc) = [t_0\,.\,F]$ and $methodBody(\delta\ t_0,m) = l$), $J' = J$, and $S' = S$.

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

We now show that $\Gamma.\hat\gamma \vdash e' : t$. Let $e_{\text{left}} = \text{chain }\bullet,\langle\!|\text{call},-,m,-,\tau,\hat\gamma'|\!\rangle(\ loc_\delta,v_1,\ldots,v_n\ )$. The judgment $\Gamma.\hat\gamma \vdash e : t$ implies that e_{left} and all its subterms are well typed. Let $\Gamma.\hat\gamma \vdash v_i : t_i$ for all $i \in \{1..n\}$ and let $\Gamma.\hat\gamma \vdash e_{\text{left}} : s$. By part 1(a) of $\Gamma \approx S$ and T-LOC, $\Gamma.\hat\gamma \vdash loc_\delta : \delta\ t_0$. 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 $\delta\ t_0 \preccurlyeq s_0$, $t_i \preccurlyeq s_i$ for all $i \in \{1..n\}$, $depClose(\hat\gamma') \subseteq \hat\gamma$, and $(\delta = \text{readonly}) \implies (\hat\gamma' = \emptyset)$.

By Lemma 4.16 (Join Point Abstractions) on page 176, 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*, it must be the case that $l = methodBody(\delta\ t_0,m) = \text{fun }m\langle\text{this},var_1,\ldots,var_n\rangle.e'' : (u\times s_1\times\ldots\times s_n\to s)\,.\,\hat\gamma'$, for some $u, \delta\ t_0 \preccurlyeq u$. Now $\delta = \text{readonly}$ implies $\hat\gamma' = \emptyset$ (from T-CHAIN) and $readonly(u) = \text{readonly}$ (from definition of subtyping). By T-CLASS, T-MET, and *override*, we have $\Gamma, \text{this} : u, var_1 : s_1,\ldots,var_n : s_n\,.\,\hat\gamma' \vdash_{DT_m} e'' : s'$ for some $s' \preccurlyeq s$ and $\forall(\gamma,\gamma') \in DT_m\cdot\gamma = \gamma'$ (where we are relying on the hypotheses of T-MET that relate $readonly(u)$ in this judgment to whether or not $\hat\gamma'$ includes the home domain of the self object, and hence whether or not $\hat\gamma'$ is empty). By the definition of the evaluation dependency table, $\forall g \in \hat g\cdot(g,g) \in DT$. So

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

$DT_m \subseteq DT$. Because e'' is a method body, it only contains user syntax. Thus, by Lemma 4.7 (Dependency Table Extension) on page 153, $\Gamma, \mathsf{this} : u, var_1 : s_1, \ldots, var_n : s_n \bullet depClose_{DT}(\hat{\gamma}') \vdash_{DT} e'' : s'$

Thus, T-EXEC gives

$$\frac{\begin{array}{c}\Gamma, \mathsf{this} : u, var_1 : s_1, \ldots, var_n : s_n \bullet depClose(\hat{\gamma}') \vdash e'' : s' \\ s' \preccurlyeq s \qquad \Gamma \bullet \hat{\gamma} \vdash loc_\delta : \delta\ t_0 \qquad \forall i \in \{1..n\} \cdot \Gamma \bullet \hat{\gamma} \vdash v_i : t_i \\ \delta\ t_0 \preccurlyeq u \qquad \forall i \in \{1..n\} \cdot t_i \preccurlyeq s_i \qquad depClose(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (\delta = \mathsf{readonly}) \implies (\hat{\gamma}' = \emptyset)\end{array}}{\Gamma \bullet \hat{\gamma} \vdash (\ \mathsf{fun}\ m\langle \mathsf{this}, var_1, \ldots, var_n\rangle . e'' : (u \times s_1 \times \ldots \times s_n \to s) \bullet \hat{\gamma}'\ (\ loc_\delta, v_1, \ldots, v_n\ )) : s}$$

and Lemma 4.11 (Replacement) on page 158 gives $\Gamma \bullet \hat{\gamma} \vdash e' : t$.

Case 4—EXEC$_\mathrm{A}$. Here $e = \mathbb{E}[(\ l\ (\ v_0, \ldots, v_n\ )\ )]$ (where $l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle . e'' : (s_0 \times \ldots \times s_n \to s) \bullet \hat{\gamma}'$), $e' = \mathbb{E}[\mathsf{joinpt}\ (\!|\mathsf{exec}, v_0, m, l, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\ v_0, \ldots, v_n\ )]$, $J' = J$, and $S' = S$.

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

We now show that $\Gamma \bullet \hat{\gamma} \vdash e' : t$. The judgment $\Gamma \bullet \hat{\gamma} \vdash e : t$ implies that $(\ l\ (\ v_0, \ldots, v_n\ )\ )$ and all its subterms are well typed. Let $\Gamma \bullet \hat{\gamma} \vdash v_i : t_i$ for all $i \in \{0..n\}$. The type derivation of $(\ l\ (\ v_0, \ldots, v_n\ )\ )$ must be by T-EXEC with $\Gamma \bullet \hat{\gamma} \vdash (\ l\ (\ v_0, \ldots, v_n\ )\ ) : s$, $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$, $depClose(\hat{\gamma}') \subseteq \hat{\gamma}$, and $(readonly(t_0) = \mathsf{readonly}) \implies (\hat{\gamma}' = \emptyset)$. If v_0 is a location, then $\Gamma \bullet \hat{\gamma} \vdash v_0 : t_0$ must be by T-LOC, so the location of v_0 is in $dom(\Gamma)$. Thus, $\Gamma \bullet \hat{\gamma} \vdash \mathsf{joinpt}\ (\!|\mathsf{exec}, v_0, m, l, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\ v_0, \ldots, v_n\ ) : s$ by T-JOIN. Lemma 4.11 (Replacement) on page 158 gives $\Gamma \bullet \hat{\gamma} \vdash e' : t$.

Case 5—EXEC$_\mathrm{B}$. Here

$$\begin{array}{rl} & e = \mathbb{E}[\mathsf{chain}\ \bullet, (\!|\mathsf{exec}, v, m, l, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\ v_0, \ldots, v_n\ )] \\ & l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle . e'' : (s_0 \times \ldots \times s_n \to s) \bullet \hat{\gamma}' \\ & e' = \mathbb{E}[\mathsf{under}\ \langle e''\{\!| v_0 / var_0, \ldots, v_n / var_n |\!\}\rangle_{\delta', \hat{\gamma}'}] \\ readonly(s) = \delta' & \\ & J' = (\!|\mathsf{this}, v_0, -, -, -, -|\!) + J \\ & S' = S \end{array}$$

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

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

To complete the case, we will next see that $\Gamma \bullet \hat{\gamma} \vdash e' : t'$ for some $t' \preccurlyeq t$. Rule T-CHAIN must be the last step in the type derivation for e_{left} with $\Gamma \bullet \hat{\gamma} \vdash e_{\mathsf{left}} : s$. The second hypothesis of T-CHAIN says that $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$.

Let $e_{\mathsf{right}} = \mathsf{under}\ \langle e''\{\!| v_0 / var_0, \ldots, v_n / var_n |\!\}\rangle_{\delta', \hat{\gamma}'}$. We now show that $\Gamma \bullet \hat{\gamma} \vdash e_{\mathsf{right}} : s'$ for some $s' \preccurlyeq s$.

That is, we want to show:

$$\dfrac{\dfrac{\Gamma \centerdot depClose(\hat{\gamma}') \vdash e''\{\!|v_0/\ var_0,\ldots,v_n/\ var_n|\!\} : u \qquad depClose(\hat{\gamma}') \subseteq \hat{\gamma}}{\Gamma \centerdot \hat{\gamma} \vdash \langle e''\{\!|v_0/\ var_0,\ldots,v_n/\ var_n|\!\}\rangle_{\delta',\hat{\gamma}'} : \delta'\ u}\ \text{T-TAG}}{\Gamma \centerdot \hat{\gamma} \vdash e_{\mathsf{right}} : s'}\ \text{T-UNDER}$$

where $s' = \delta'\ u$. The hypothesis $depClose(\hat{\gamma}') \subseteq \hat{\gamma}$ in this derivation is an hypothesis of T-CHAIN. We will appeal to the Substitution Lemma to show the other hypothesis. To do this we must first show that $\Gamma, var_0 : s_0, \ldots, var_n : s_n \centerdot depClose(\hat{\gamma}') \vdash e'' : u$ for some u such that $\delta'\ u = s' \preccurlyeq s$. 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$_\text{B}$. 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 $var_0 : s_0, \ldots, var_n : s_n \centerdot \hat{\gamma}' \vdash_{DT_m} e'' : u$, where $u \preccurlyeq s$ and $\forall\,(\gamma,\gamma') \in DT_m \cdot \gamma = \gamma'$ (i.e., DT_m is just reflexive). As in Case 3, Lemma 4.7 (Dependency Table Extension) gives $var_0 : s_0, \ldots, var_n : s_n \centerdot depClose(\hat{\gamma}') \vdash_{DT} e'' : u$. We need to show $\delta'\ u \preccurlyeq s$. But $readonly(s) = \delta'$, so by the idempotency of read-only annotations $s = \delta'\ s$. Thus $(u \preccurlyeq s) \implies (\delta'\ u \preccurlyeq \delta'\ s = s)$.

By α-conversion and Lemma 4.9 (Environment Extension) on 157 we have $\Gamma, var_0 : s_0, \ldots, var_n : s_n \centerdot depClose(\hat{\gamma}') \vdash e'' : u$. Thus, by Lemma 4.8 (Substitution) on page 154,

$$\Gamma \centerdot depClose(\hat{\gamma}') \vdash e''\{\!|v_0/\ var_0,\ldots,v_n/\ var_n|\!\} : u'$$

where $\delta'\ u' \preccurlyeq \delta'\ u \preccurlyeq s$. So Lemma 4.12 (Replacement with Subtyping) on page 159 gives $\Gamma \centerdot \hat{\gamma} \vdash e' : t'$ for some $t' \preccurlyeq t$.

Case 6—GET. In this case $e = \mathbb{E}[loc_\delta.f]$, $e' = \mathbb{E}[v_{\delta'}]$ (where $S(loc) = [T\langle\gamma_1,\ldots,\gamma_n\rangle \centerdot F]$, $F(f) = v$, and $readonly(fieldsOf(\delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle)(f)) = \delta'$), $J' = J$, and $S' = S$.

Let $\Gamma' = \Gamma$.

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

We now show that $\Gamma \centerdot \hat{\gamma} \vdash \mathbb{E}[v_{\delta'}] : t'$ for some $t' \preccurlyeq t$. If $v = \mathsf{null}$, then this is immediate by T-NULL and Lemma 4.11 (Replacement) on page 158. So assume $v = loc'$. Let $\Gamma \centerdot \hat{\gamma} \vdash loc_\delta.f : s$. The last step in this derivation must be T-GET. By $\Gamma \approx S$, we have $\Gamma(loc) = T\langle\gamma_1,\ldots,\gamma_n\rangle$. By the second hypothesis of T-GET, $fieldsOf(\delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle)(f) = s = \delta'\ S\langle\gamma'_1,\ldots,\gamma'_p\rangle$ for some S and $\gamma'_1,\ldots,\gamma'_p$. Also by $\Gamma \approx S$, $S(loc') = [u' \centerdot F']$ and $\Gamma(loc') = u'$. Thus, $\Gamma \centerdot \hat{\gamma} \vdash loc'_{\delta'} : \delta'\ u'$. It remains to be seen that $\delta'\ u' \preccurlyeq s$.

Now there are two subcases depending on the values of δ and $readonly(fieldsOf(T\langle\gamma_1,\ldots,\gamma_n\rangle)(f))$—call this later value δ''. Note that δ'' is the read-only status of the field as declared, ignoring the value of δ.

Subcase 1. If $\delta = \varepsilon$ and $\delta'' = \varepsilon$, then part 1(d) of $\Gamma \approx S$ gives $u' \preccurlyeq s$. Furthermore, by the definition of *fieldsOf*, $\delta' = \varepsilon$. Thus $\delta'\ u' \preccurlyeq s$.

Subcase 2. If $\delta = \mathsf{readonly}$ or $\delta'' = \mathsf{readonly}$, then by the definition of *fieldsOf*, $\delta' = \mathsf{readonly}$. By part 1(d) of $\Gamma \approx S$, $\delta''\ u' \preccurlyeq s$. Finally, by idempotency of readonly and the definition of subtyping, $\delta'\ u' = \mathsf{readonly}\ u' = \mathsf{readonly}\ \delta''\ u' \preccurlyeq s$.

So in either case, Lemma 4.12 (Replacement with Subtyping) on page 159 gives $\Gamma \centerdot \hat{\gamma} \vdash \mathbb{E}[loc'_{\delta'}] : t'$ where $t' \preccurlyeq t$.

Case 7—SET. In this case $e = \mathbb{E}[loc_\delta.f = v]$. Because e is well typed, we know from T-SET that $\delta = \varepsilon$, so we can omit it for the remainder of the case. Also $e' = \mathbb{E}[v]$, $J' = J$, and $S' = S \oplus \left(loc \mapsto \left[u\,.\,F \oplus (f \mapsto v')\right]\right)$, where $S(loc) = [u\,.\,F]$ and $v' = \begin{cases} loc' & \text{if } v = loc'_{\delta'} \\ \mathsf{null} & \text{otherwise} \end{cases}$.

Let $\Gamma' = \Gamma$.

By the monotonicity property of the store, $J' = J \implies J' \approx S'$.

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

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

Part 1(d) holds for all $f' \in dom(F)$, $f' \neq f$. Part 1(d) holds vacuously for f if $v = \mathsf{null}_{\delta'}$. Otherwise, by T-SET and T-LOC, $\Gamma\left(v'\right) \preccurlyeq fieldsOf(u)(f)$. Part 1(d) holds because $(F \oplus \left(f \mapsto v'\right))(f) = v'$ and $\delta''\ \Gamma(v) \preccurlyeq fieldsOf(u)(f)$ for all values of δ'' (by the definition of subtyping).

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

To see that $\Gamma\,.\,\hat{\gamma} \vdash \mathbb{E}[v] : t$, let $\Gamma\,.\,\hat{\gamma} \vdash loc.f = v : s$. By T-SET, $\Gamma\,.\,\hat{\gamma} \vdash v : s$ and by Lemma 4.11 (Replacement) on page 158, $\Gamma\,.\,\hat{\gamma} \vdash \mathbb{E}[v] : t$.

Case 8—CAST. Here $e = \mathbb{E}[\mathsf{cast}\ t''\ loc_\delta]$, $e' = \mathbb{E}[loc_\delta]$, $J' = J$, $S' = S$, $S(loc) = [u\,.\,F]$, and $\delta\ u \preccurlyeq t''$.

Let $\Gamma' = \Gamma$.

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

To see that $\Gamma\,.\,\hat{\gamma} \vdash \mathbb{E}[loc_\delta] : t'$ for some $t' \preccurlyeq t$, note that $\Gamma(loc) = u$ by consistency of Γ with S. Thus $\Gamma\,.\,\hat{\gamma} \vdash loc_\delta : \delta\ u$. By T-CAST, $\Gamma\,.\,\hat{\gamma} \vdash \mathsf{cast}\ t''\ loc_\delta : \delta\ t''$.

If $\delta = \mathsf{readonly}$, then $\delta u \preccurlyeq t''$ implies $readonly\left(t''\right) = \mathsf{readonly}$. By idempotency of read-only annotations, $\delta\ t'' = \mathsf{readonly}\ t'' = t''$. If $\delta = \varepsilon$, then $\delta\ t'' = \varepsilon\ t'' = t''$. In either case, $\delta\ u \preccurlyeq t'' = \delta\ t''$, so by Lemma 4.12 (Replacement with Subtyping) on page 159 we have $\Gamma\,.\,\hat{\gamma} \vdash \mathbb{E}[loc_\delta] : t'$ where $t' \preccurlyeq t$.

Case 9—NCAST. Here $e = \mathbb{E}[\mathsf{cast}\ \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle\ \mathsf{null}_{\delta'}]$, $e' = \mathbb{E}[\mathsf{null}_{\delta'}]$, $J' = J$, $S' = S$, and either $\delta = \mathsf{readonly}$ or $\delta' = \varepsilon$.

Let $\Gamma' = \Gamma$.

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

Now $\Gamma\,.\,\hat{\gamma} \vdash \mathsf{cast}\ \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle\ \mathsf{null}_{\delta'} : \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle$. By T-NULL, $\Gamma\,.\,\hat{\gamma} \vdash \mathsf{null}_{\delta'} : \delta'\ t''$ for any $t'' \in \mathscr{T}$. We want $\delta'\ t'' \preccurlyeq \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle$. If $\delta = \mathsf{readonly}$, then let $t'' = \mathsf{readonly}\ T\langle\gamma_1,\ldots,\gamma_n\rangle$; by idempotency of read-only annotations, the value of δ' does not matter. On the other hand, if $\delta' = \varepsilon$, then let $t'' = \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle$. In either case, by Lemma 4.12 (Replacement with Subtyping) on page 159, $\Gamma\,.\,\hat{\gamma} \vdash \mathbb{E}[\mathsf{null}_{\delta'}] : t'$ for some $t' \preccurlyeq t$.

Case 10—SKIP. Here $e = \mathbb{E}[v;\ e'']$, $e' = \mathbb{E}[e'']$, $J' = J$, and $S' = S$.

Let $\Gamma' = \Gamma$.

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

Since $\Gamma \centerdot \hat{\gamma} \vdash \mathbb{E}[v;\ e''] : t$, let $\Gamma \centerdot \hat{\gamma} \vdash v;\ e'' : t''$. This derivation must be by T-SEQ, the second hypothesis of which says $\Gamma \centerdot \hat{\gamma} \vdash e'' : t''$. By Lemma 4.11 (Replacement) on page 158, $\Gamma \centerdot \hat{\gamma} \vdash \mathbb{E}[e''] : t$.

Case 11—BIND. Here:

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

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

We will see that $J' \approx S'$. Let

$$e_{\text{left}} = \mathsf{joinpt}\ (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\ v_0, \ldots, v_n\ ).$$

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 with $\Gamma \centerdot \hat{\gamma} \vdash e_{\text{left}} : s$. Thus, if v_{opt} is a location it must be in $\mathit{dom}(\Gamma)$ and so $J' \approx S'$.

It remains to show that $\Gamma \centerdot \hat{\gamma} \vdash e' : t$. Let

$$e_{\text{right}} = \mathsf{chain}\ \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\ v_0, \ldots, v_n\ ).$$

(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 for e_{right}.

By the well-typedness of e_{left} and its subterms, let $\Gamma \centerdot \hat{\gamma} \vdash v_i : t_i$ for all $i \in \{0..n\}$. By T-JOIN, we have $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$, $\mathit{depClose}(\hat{\gamma}') \subseteq \hat{\gamma}$, and $\big(\mathit{readonly}(u_0) = \mathsf{readonly}\big) \implies \big(\hat{\gamma}' = \emptyset\big)$.

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

$$B = [\![b, loc, e'', \hat{\gamma}'', \tau, \tau']\!]$$

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 advice table entry $\langle loc, pcd, e'', \hat{\gamma}'', \tau, \tau' \rangle$ such that $\mathit{matchPCD}(J', pcd, S) = b \neq \bot$. By Lemma 4.14 (Binding Soundness) on page 162 we have:

$$
\begin{gathered}
\mathit{depClose}(\hat{\gamma}'') \subseteq \mathit{depClose}(\hat{\gamma}') \\
\tau' = s_0 \times \ldots \times s_n \to s \\
\emptyset \vdash b\ \text{OK} \\
\Gamma, \mathsf{this} : \Gamma(loc), \mathsf{proceed} : \tau', \mathit{typeBind}(\Gamma, b, \langle s_0, \ldots, s_n \rangle) \centerdot \mathit{depClose}(\hat{\gamma}') \vdash e'' : s' \text{ for some } s' \preccurlyeq s
\end{gathered}
$$

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, $\Gamma \centerdot \hat{\gamma} \vdash e_{\text{right}} : s$ and by T-UNDER and Lemma 4.11 (Replacement) on page 158, $\Gamma \centerdot \hat{\gamma} \vdash e' : t$.

Case 12—ADVISE. Here

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

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 $\Gamma \centerdot \hat{\gamma} \vdash e' : t'$ for some $t' \preccurlyeq t$. Let

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

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 two elements of j be $t_0 \times \ldots \times t_n \to t_{\text{m}}$ and $\hat{\gamma}_{\text{m}}$. Then by T-CHAIN the proceed type $\tau' = t_0 \times \ldots \times t_n \to t_{\text{m}}$, $\mathit{depClose}(\hat{\gamma}_{\text{m}}) \subseteq \hat{\gamma}$, and $\Gamma \centerdot \hat{\gamma} \vdash e_{\text{left}} : t_{\text{m}}$. We want to show that $\Gamma \centerdot \hat{\gamma} \vdash \text{under } \langle e_{\text{right}} \rangle_{\delta,\hat{\gamma}'} : u$ for some $u \preccurlyeq t_{\text{m}}$. That is, we want to show:

$$\dfrac{\dfrac{\Gamma \centerdot \mathit{depClose}(\hat{\gamma}') \vdash e_{\text{right}} : t'_{\text{m}} \qquad \mathit{depClose}(\hat{\gamma}') \subseteq \hat{\gamma}}{\Gamma \centerdot \hat{\gamma} \vdash \langle e_{\text{right}} \rangle_{\delta,\hat{\gamma}'} : \delta\ t'_{\text{m}}}\ \text{T-TAG}}{\Gamma \centerdot \hat{\gamma} \vdash \text{under } \langle e_{\text{right}} \rangle_{\delta,\hat{\gamma}'} : \delta\ t'_{\text{m}}}\ \text{T-UNDER} \tag{4.6}$$

where $\delta\ t'_{\text{m}} \preccurlyeq t_{\text{m}}$. From the hypotheses of T-CHAIN, $\mathit{depClose}(\hat{\gamma}') \subseteq \mathit{depClose}(\hat{\gamma}_{\text{m}}) \subseteq \hat{\gamma}$. So the second hypothesis of the above derivation holds. The first hypothesis will require a bit more effort.

From the hypotheses of T-CHAIN, we have

$$\Gamma, \text{this} : \Gamma(loc), \text{proceed} : (t_0 \times \ldots \times t_n \to t_{\text{m}}), \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \centerdot \mathit{depClose}(\hat{\gamma}') \vdash e'' : s$$

where $s \preccurlyeq t_{\text{m}}$. The constraints on $\bar{B}$ and j imposed by T-CHAIN satisfy the conditions of Lemma 4.15 (Advice Chaining) on page 175, so we have

$$\Gamma, \text{this} : \Gamma(loc), \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \centerdot \mathit{depClose}(\hat{\gamma}') \vdash \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} : s \tag{4.7}$$

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 $\alpha = var' \mapsto loc'_\delta$ and $\beta_0 = var_0$.[5] Then (4.7) expands to

$$\Gamma, \text{this} : \Gamma(loc), var' : \delta\ \Gamma(loc'), (var_i : t_i)_{i \in \{0..p\} \cdot \beta_i = var_i} \centerdot \mathit{depClose}(\hat{\gamma}') \vdash \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} : s. \tag{4.8}$$

[5]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.

The binding substitution in e_{right} expands to give

$$e_{\text{right}} = \langle\!\langle e'' \rangle\!\rangle_{\bar{B},j} \{\!| loc/\,\text{this}, loc'_\delta/\,var', (v_i/\,var_i)_{i\in\{0..p\}\cdot\beta_i = var_i} |\!\}. \tag{4.9}$$

By the hypotheses of T-CHAIN in the typing of e_{left} we have $\forall i \in \{0..n\} \cdot \left(\Gamma \centerdot \hat{\gamma} \vdash v_i : u'_i \text{ where } u'_i \preccurlyeq t_i\right)$. Each of these typing judgments must be by T-LOC or T-NULL, neither of which uses $\hat{\gamma}$ in its hypotheses. So we have $\forall i \in \{0..n\} \cdot \left(\Gamma \centerdot depClose\left(\hat{\gamma}'\right) \vdash v_i : u'_i \text{ where } u'_i \preccurlyeq t_i\right)$.[6] Using these judgments, along with (4.8) and (4.9), Lemma 4.8 (Substitution) on page 154 gives $\Gamma \centerdot depClose\left(\hat{\gamma}'\right) \vdash e_{\text{right}} : t'_{\text{m}}$ where $t'_{\text{m}} \preccurlyeq s \preccurlyeq t_{\text{m}}$.

Now suppose $\delta = \varepsilon$, then $\delta\ t'_{\text{m}} = t'_{\text{m}} \preccurlyeq t_{\text{m}}$.

On the other hand, suppose $\delta = \text{readonly}$. Recall that $readonly(t_{\text{m}}) = \delta$, thus $\text{readonly}\ t_{\text{m}} = t_{\text{m}}$. So $\delta\ t'_{\text{m}} = \text{readonly}\ t'_{\text{m}} = \text{readonly}\ t_{\text{m}} = t_{\text{m}}$.

So (4.6) on the preceding page holds. By Lemma 4.12 (Replacement with Subtyping) on page 159, $\Gamma \centerdot \hat{\gamma} \vdash e' : t'$ for some $t' \preccurlyeq t$. Whew!

Case 13—UNDER. Here $e = \mathbb{E}[\text{under}\ v]$, $e' = \mathbb{E}[v]$, $J = j + J'$ 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 now show that $\Gamma \centerdot \hat{\gamma} \vdash e' : t$. The judgment $\Gamma \centerdot \hat{\gamma} \vdash e : t$ implies that $\text{under}\ v$ is well typed. Let $\Gamma \centerdot \hat{\gamma} \vdash \text{under}\ v : t'$. This judgment must be by T-UNDER with the hypothesis $\Gamma \centerdot \hat{\gamma} \vdash v : t'$. So by Lemma 4.11 (Replacement), we have $\Gamma \centerdot \hat{\gamma} \vdash e' : t$.

Case 14—TAG. Here $e = \mathbb{E}[\langle v \rangle_{\delta,\hat{\gamma}'}]$, $e' = \mathbb{E}[v_\delta]$, $J' = J$, and $S' = S$.

Let $\Gamma' = \Gamma$.

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

We now show that $\Gamma \centerdot \hat{\gamma} \vdash e' : t$. Because e is well typed, it must be the case that $\langle v \rangle_{\delta,\hat{\gamma}'}$ is also well typed. Let $\Gamma \centerdot \hat{\gamma} \vdash \langle v \rangle_{\delta,\hat{\gamma}'} : \delta\ s$. This must be by T-TAG with $\Gamma \centerdot depClose\left(\hat{\gamma}'\right) \vdash v : s$ and $depClose\left(\hat{\gamma}'\right) \subseteq \hat{\gamma}$.

If $v = \text{null}_{\delta'}$, then T-NULL gives $\Gamma \centerdot \hat{\gamma} \vdash v_\delta : \delta\ s$. On the other hand, if $v = loc_{\delta'}$ then $\Gamma \centerdot depClose\left(\hat{\gamma}'\right) \vdash v : s$ must be by T-LOC with $loc \in dom(S)$. So T-LOC gives $\Gamma \centerdot \hat{\gamma} \vdash v_\delta : \delta\ s$. In either case, Lemma 4.11 (Replacement) gives $\Gamma \centerdot \hat{\gamma} \vdash e' : t$.

The remaining evaluation rules reduce e to some exception, which is not an expression. Thus, those rules are not applicable to the theorem. □

The statement of the Progress theorem includes the public concern domains declared in the program, a set of writable concern domains, and the evaluation dependency table. It also considers that values may now have read-only annotations. These changes do not affect the result, or the proof, in a substantial way.

[6] Intuitively, the values v_i in these judgments were generated by either the code that created the join point j, or by a previous piece of advice in the chain. In either case, they may have been generated using a different set of writable domains. That does not matter for execution of the advice, just as we do not care about the writable domains used to generate the actual arguments to a method call.

Theorem 4.18 (Progress). *Given a well-typed program, P, with public concern domains $\hat{g} \subset \mathcal{G}$, for an expression e, a valid store S, a stack J consistent with S, a concern-complete type environment Γ consistent with S, a set of public concern domains $\hat{\gamma} \subseteq \hat{g}$, and the evaluation dependency table DT, such that the triple $\langle e, J, S\rangle$ is reached in the evaluation of P, if $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$ then either:*

— *$e = loc_\delta$ for some δ and $loc \in dom(S)$,*

— *$e = null_\delta$ for some δ, 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 $e = loc_\delta$, then $\Gamma \centerdot \hat{\gamma} \vdash_{DT} loc_\delta : t$ by T-LOC. This means that $loc \in dom(\Gamma)$ and, since $\Gamma \approx S$ we have $loc \in dom(S)$.

If $e = \mathsf{null}_\delta$, 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, ADVISE, EXEC$_\mathrm{A}$, SKIP, BIND, TAG, NCALL$_\mathrm{A}$, NCALL$_\mathrm{B}$, NGET, and NSET are trivial. For the remaining cases we must show that the side conditions hold for the relevant evaluation rules, and that the join point abstractions are of the correct form. I omit the *DT* subscript for the remainder of the proof, with the understanding that the same dependency table is used throughout.

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

Because $loc_\delta.m(\,v_1, \ldots, v_n\,)$ is well typed, we know by the hypotheses of T-CALL that $methodType(\delta\ s, m)$ yields an n-arity method type and $writable(\delta\ s, m)$ is well defined. By the definition of *origType*, we know that $origType(\delta\ s) = t_0$, where $\delta\ s \preccurlyeq t_0$. By T-CLASS, T-MET, and *override*, we know that $methodType(t_0, m)$ also yields an n-arity method and $writable(t_0, m)$ is well defined. Thus, $\langle e, J, S\rangle$ evolves by CALL$_\mathrm{A}$.

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 4.16 (Join Point Abstractions), there are two cases:

— $j = (\!|\mathsf{exec}, v, m, l, \tau, \hat{\gamma}'|\!)$: By Lemma 4.16, $l = \mathsf{fun}\ m\langle var_0, \ldots, var_n\rangle.e : \tau \centerdot \hat{\gamma}'$. Thus, $\langle e, J, S\rangle$ evolves by EXEC$_\mathrm{B}$.

— $j = (\!|\mathsf{call}, -, m, -, \tau, \hat{\gamma}|\!)$: There are two subcases. If $v_0 = \mathsf{null}_\delta$, then $\langle e, J, S\rangle$ evolves by NCALL$_\mathrm{B}$ to a triple with a NullPointerException. Otherwise, v_0 is a location. Let $v_0 = loc_\delta$. Because e is well typed we have $\Gamma \centerdot \hat{\gamma} \vdash loc_\delta : \delta\ u_0'$ for some u_0'; this is by T-LOC with $loc \in dom(\Gamma)$. By $\Gamma \approx S$, $S(loc) = \left[u_0' \centerdot F\right]$. 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 4.16, $methodType(t_0, m) = t_1 \times \ldots \times t_n \to t$. Also by T-CHAIN, $\delta\ u_0' \preccurlyeq t_0$. 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 $methodBody(\delta\ u_0', m) = l$. Therefore, $\langle e, J, S\rangle$ evolves by CALL$_\mathrm{B}$ in this subcase.

Case 3—$e = \mathbb{E}[loc_\delta.f]$. As in Case 1, e well typed implies $S(loc) = [s\,.\,F]$ where $\Gamma(loc) = s$. Now $loc_\delta.f$ well typed implies $f \in dom\big(fieldsOf(\delta\ s)\big)$ 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 4—$e = \mathbb{E}[loc_\delta.f = v]$. Similar to the preceding case.

Case 5—$e = \mathbb{E}[cast\ t'\ loc_\delta]$. As in Case 1, e well typed implies $S(loc) = [s\,.\,F]$, where $\Gamma(loc) = s$. If $\delta\ s \preccurlyeq t'$, then $\langle e, J, S\rangle \hookrightarrow \big\langle \mathbb{E}[loc_\delta], J, S\big\rangle$ by CAST; otherwise $\langle e, J, S\rangle \hookrightarrow \big\langle \text{ClassCastException}, J, S\big\rangle$ by XCAST.

Case 6—$e = \mathbb{E}[cast\ \delta\ T\langle\gamma_1,\ldots,\gamma_n\rangle\ null_{\delta'}]$. If $\delta = \text{readonly}$ or $\delta' = \varepsilon$, then $\langle e, J, S\rangle \hookrightarrow \big\langle \mathbb{E}[\text{null}_{\delta'}], J, S\big\rangle$ by NCAST. Now

$$\begin{aligned}\neg(\delta = \text{readonly or } \delta' = \varepsilon) &= (\delta \neq \text{readonly and } \delta' \neq \varepsilon)\\ &= (\delta = \varepsilon \text{ and } \delta' = \text{readonly}).\end{aligned}$$

So if $\langle e, J, S\rangle$ does not evolve by NCAST, it must evolve to $\big\langle \text{ClassCastException}, J, S\big\rangle$ by NXCAST.

Case 7—$e = \mathbb{E}[under\ v]$. In this case, we only need to argue that the stack, J, is not empty. Note that under expressions are not part of the user syntax. These expressions are only introduced during the evaluation of a program, by rules 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, $\big\langle \mathbb{E}[\text{under } v], j + J, S\big\rangle \hookrightarrow \langle \mathbb{E}[v], J, S\rangle$ by rule UNDER. □

In MiniMAO_2, the Type Safety theorem considers that values may now have read-only annotations. This change does not affect the result. I update the proof to establish the conditions under which the main expression of the program is well typed. This is slightly more involved than in MiniMAO_1, where the T-PROG rule typechecks the main expression in an empty environment. In MiniMAO_2, T-PROG typechecks the main expression using a judgment that has a non-empty type environment, a set of writable concern domains, and a dependency table that does not match the evaluation dependency table used in the Subject Reduction and Progress theorems. However, after establishing the appropriate initial conditions, the proof follows immediately.

Theorem 4.19 (Type Safety). *Given a program P, with main expression e, concern domains $\hat{g}$, $\vdash P$ OK, and a valid store S_0, then either the evaluation of e diverges or else $\langle e, \bullet, S_0\rangle \overset{*}{\hookrightarrow} \langle x, J, S\rangle$ and one of the following hold for x:*

— *$x = loc_\delta$ for some δ and $loc \in dom(S)$,*

— *$x = null_\delta$ for some δ,*

— *$x = NullPointerException$, or*

— *$x = ClassCastException$*

Proof. If e diverges then the claim holds. If e converges, then note that the empty stack is consistent with any store, the validity of S_0 implies the existence of an initial type environment consistent with S_0, and $\vdash P$ OK implies $\Gamma\,.\,\hat{g} \vdash_{DT_P} e : t$ for some t, where $DT_P = \bigcup_{g\in\hat{g}} (g, g)$. Let DT be the evaluation dependency table

for P. By Definition 4.1 (Evaluation Dependency Table) on page 140, $DT_P \subseteq DT$ and $\forall g \in \hat{g} \cdot (g, g) \in DT$. Because e is the main expression of the program, it only contains user syntax. Also, because $\hat{g}$ includes every concern domain in P, $\hat{g} = depClose_{DT}(\hat{g})$. Thus, by Lemma 4.7 (Dependency Table Extension) on page 153, $\Gamma \centerdot \hat{g} \vdash_{DT} e : t$.

The proof (by induction on the number of evaluation steps) follows from Theorem 4.17 (Subject Reduction) on page 177 and Theorem 4.18 (Progress) on page 185. □

4.4.3 Effects Properties

MiniMAO$_2$ includes features that let programmers control effects in two ways. With read-only annotations, programmers can write code that captures object references, while keeping that code from mutating the referenced objects or their representations. With concern domains and effects clauses, programmers may specify the domains that a piece of code is intended to modify. These specifications allow the type system to help the programmer by pointing out code that differs from its specified intent. These specifications also serve to document the possible side effects of a code block during maintenance. In this section, I show that the features of MiniMAO$_2$ really do allow such control of effects. In particular, I show that a readonly pointer cannot be used to mutate the object to which it points, or any object reached through that pointer. I also show that, given the aspect instantiation instructions and dependency declarations for a program, for any code block (either a method or piece of advice) the only domains that may be mutated by executing the block are those listed in its effects clause, plus those domains that vary with the listed domains.

4.4.3.1 Effects Clauses

I treat effects clauses first. I begin by introducing a notation for describing the portion of a store that appears in a given concern domain.

Definition 4.20 (Concern Domain). Let S be a valid store for a well-typed program P. For any concern domain name g, the *concern domain* g in the store S, written $S|g$, is:

$$S|g = \left\{ \left(loc \mapsto \left[T\langle g_1, \ldots, g_n \rangle \centerdot F \right] \right) \in S \cdot g_1 = g \right\}.$$

The following lemma says that the expression typing rules in MiniMAO$_2$ obey a monotonicity property: any writable domain set used in the environment of an hypothesis is a subset of the writable domain set used in the judgment.

Lemma 4.21 (Expression Typing Monotonicity). *If* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$*, then for any subderivation* $\Gamma' \centerdot \hat{\gamma}' \vdash_{DT} e' : t'$*, it is the case that* $\hat{\gamma}' \subseteq \hat{\gamma}$.

Proof. The proof is immediate by inspection of the typing rules. □

Consider a tagged expression $\langle e \rangle_{\delta, \hat{\gamma}}$. Intuitively, such an expression is introduced to the evaluation triple when a method or piece of advice begins executing. Theorem 4.23 (Tag Frame Soundness) below states that the evaluation of such an expression may only mutate the concern domains in the dependency closure of $\hat{\gamma}$. All other concern domains are immutable until the tagged expression has been reduced to a value. The following lemma is useful in proving the theorem.

Lemma 4.22 (Domain Preservation). *Let P be a well-typed program with concern domains $\hat{g}$ and evaluation dependency table DT. Suppose the evaluation step $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle \hookrightarrow \langle \mathbb{E}[e'], J', S'\rangle$ occurs in an evaluation of P. Then $\forall g \in (\hat{g} \setminus depClose_{DT}(\hat{\gamma})) \cdot S|g = S'|g$.*

Proof. By $\vdash P$ OK, Theorem 4.17 (Subject Reduction), and Theorem 4.18 (Progress) we know that $\mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}]$ is well typed. Therefore $\langle e\rangle_{\delta,\hat{\gamma}}$ is also well typed. This must be by T-TAG. By the hypotheses of that rule, there must be some Γ and t such that $\Gamma \centerdot depClose_{DT}(\hat{\gamma}) \vdash_{DT} e : t$.

Let e'' be the current redex of $\mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}]$. Let $\mathbb{E}'[-]$ be defined such that $\mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}] = \mathbb{E}[\langle \mathbb{E}'[e'']\rangle_{\delta,\hat{\gamma}}]$. By Lemma 4.21 (Expression Typing Monotonicity), $\Gamma \centerdot depClose_{DT}(\hat{\gamma}) \vdash_{DT} e : t$ implies that there exists $\hat{\gamma}' \subseteq depClose_{DT}(\hat{\gamma})$ such that $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e'' : s$ for some type s.

We must consider all of the possible evaluation rules that might generate the evaluation step assumed by the theorem. For all rules except NEW and SET, $S' = S$ and the claim holds.

For NEW, let $e'' = \text{new } c\langle g_1,\ldots,g_n\rangle$. $S' = S \oplus (loc \mapsto [c\langle g_1,\ldots,g_n\rangle \centerdot F])$, where $loc \notin dom(S)$ and $rng(F) = \{\text{null}\}$. For all $g \in \hat{g}$ such that $g \neq g_1$, we see that $S'|g = S|g$. Now $S'|g_1 \neq S|g_1$, but we will see that g_1 must be in $depClose_{DT}(\hat{\gamma})$. The type judgment $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e'' : s$ must be by T-NEW or T-OBJ. In either case $g_1 \in \hat{\gamma}'$, and $\hat{\gamma}' \subseteq depClose_{DT}(\hat{\gamma})$ implies $g_1 \in depClose_{DT}(\hat{\gamma})$. Thus the claim holds.

For SET, let $e'' = (loc_\delta.f{=}v)$, where $S(loc) = [T\langle g_1,\ldots,g_n\rangle \centerdot F]$. Then

$$S' = S \oplus (loc \mapsto [T\langle g_1,\ldots,g_n\rangle \centerdot F \oplus (f \mapsto v')]),$$

where

$$v' = \begin{cases} loc' & \text{if } v = loc'_{\delta'} \\ \text{null} & \text{otherwise} \end{cases}$$

For all $g \in \hat{g}$ such that $g \neq g_1$, we see that $S'|g = S|g$. Now if $F(f) \neq v'$, then $S'|g_1 \neq S|g_1$. But again we will see that g_1 must be in $depClose_{DT}(\hat{\gamma})$. The type judgment $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e'' : s$ must be by T-SET with hypotheses $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} loc : T\langle g_1,\ldots,g_n\rangle$ (by $\Gamma \approx S$) and $g_1 \in \hat{\gamma}'$. As for NEW, $g_1 \in depClose_{DT}(\hat{\gamma})$ and the claim holds. □

Theorem 4.23 (Tag Frame Soundness). *Let P be a well-typed program with concern domains $\hat{g}$ and evaluation dependency table DT. Suppose the evaluation triple $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle$ appears in an evaluation of P. Then either the evaluation diverges or $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle \overset{*}{\hookrightarrow} \langle \mathbb{E}[v], J', S'\rangle$, where $\forall g \in (\hat{g} \setminus depClose_{DT}(\hat{\gamma})) \cdot S|g = S'|g$.*

Proof. By inspection of the semantics, to reach a value with the tagged expression removed the evaluation must be

$$\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle \overset{*}{\hookrightarrow} \langle \mathbb{E}[\langle v\rangle_{\delta,\hat{\gamma}}], J', S'\rangle \hookrightarrow \langle \mathbb{E}[v_\delta], J', S'\rangle.$$

The claim holds for the last step of this evaluation since the store is unchanged by the TAG rule. The claim holds for each of the other steps in this evaluation by Lemma 4.22 (Domain Preservation). □

This theorem is sufficient for reasoning about the concern domains affected by a method. To reason about the execution of a method one must know the method's signature including its effects clause, the concern domains of the target object, and the configuration of aspects in the program, as represented by the aspect instantiation instructions and dependency declarations.

In the operational semantics of MiniMAO$_2$, the execution of a method begins when the method body is inserted in the evaluation context, substituting actual arguments for formal parameters. This only occurs in the EXEC$_B$ rule, where the method lookup occurs in CALL$_B$ and the fun term representing the method is threaded through joinpt and chain expressions unchanged. Execution of the method ends when the method body has been reduced to a value. In EXEC$_B$, method execution begins with a tagged expression where the writable domains set is that of the method's effects clause. So by Theorem 4.23 (Tag Frame Soundness) on the preceding page, only public concern domains in the dependency closure of the effects clause may change.

Similarly, Theorem 4.23 (Tag Frame Soundness) is also sufficient for reasoning about the concern domains affected by a piece of advice, where advice lookup occurs in the BIND rule and advice execution begins in the ADVISE rule.

4.4.3.2 Read-Only Annotations

Next I deal with the properties of read-only annotations in MiniMAO$_2$. Read-only annotations are associated with pointers in the operational semantics. Their meta-theory can be described in terms of the pattern of location references in the store. I begin with some definitions to help formalize this notion. A small, illustrative example follows the definitions.

The first definition gives the reachability relation for a store. Intuitively, one location may reach a second location if a sequence of field accesses beginning with the first location eventually yields the second location.

Definition 4.24 (Reach). Let S be a valid store occurring in the evaluation of some program. The *reach* of S, denoted $reach(S)$, is the reflexive, transitive closure of the set

$$\{(\mathsf{null},\mathsf{null})\} \cup \{(loc, v) \cdot loc \in dom(S),\ S(loc) = [t\,.\,F], \text{ and } \exists f \cdot \big((f \mapsto v) \in F\big)\}.$$

The next definition refines the notion of reach to include locations reachable by accessing a sequence of write-enabled fields.

Definition 4.25 (Writable Reach). Let S be a valid store occurring in the evaluation of some program. The *writable reach* of S, denoted $writeReach(S)$, is the reflexive, transitive closure of the set

$$\{(\mathsf{null},\mathsf{null})\} \cup \{(loc, v) \cdot loc \in dom(S),\ S(loc) = [t\,.\,F], \text{ and } \exists f \cdot \big((f \mapsto v) \in F \text{ and } readonly\big(fieldsOf(t)(f)\big) = \varepsilon\big)\}.$$

A third definition lets me discuss the "value" of an object in terms of the values of its fields.

Definition 4.26 (Object Graph). Let S be a valid store occurring in the evaluation of some program, and let loc be a location, $loc \in dom(S)$. The *object graph* of loc in S, denoted $\mathbb{G}_S(loc) = (L, E)$, is the least fix point of the following pair of mutually recursive functions:

$$E_0 = \emptyset \qquad E_i = \left\{\left(loc' \xrightarrow{f} v\right) \cdot loc' \in L_{i-1},\ S\big(loc'\big) = [t\,.\,F], \text{ and } \big(f \mapsto v\big) \in F\right\}$$

$$L_0 = \{loc\} \qquad L_i = \left\{v \cdot \left(\exists f, v' \cdot \left(v \xmapsto{f} v'\right) \in E_i\right) \text{ or } \left(\exists f, loc' \cdot \left(loc' \xrightarrow{f} v\right) \in E_i\right)\right\}$$

It is sometimes useful to refer directly to the set of locations in the object graph of some $S(loc)$. I define the auxiliary function $rep_S(loc)$, see Figure 4.23 on the following page, for this purpose.

$rep_S(loc) = L$, where $\mathbb{G}_S(loc) = (L, E)$ $\qquad$ $domains_S(loc) = \{g_1, \ldots, g_n\}$, where $S(loc) = [T\langle g_1, \ldots, g_n\rangle \centerdot F]$

Figure 4.23 Auxiliary Functions for the Meta-theory of MiniMAO$_2$

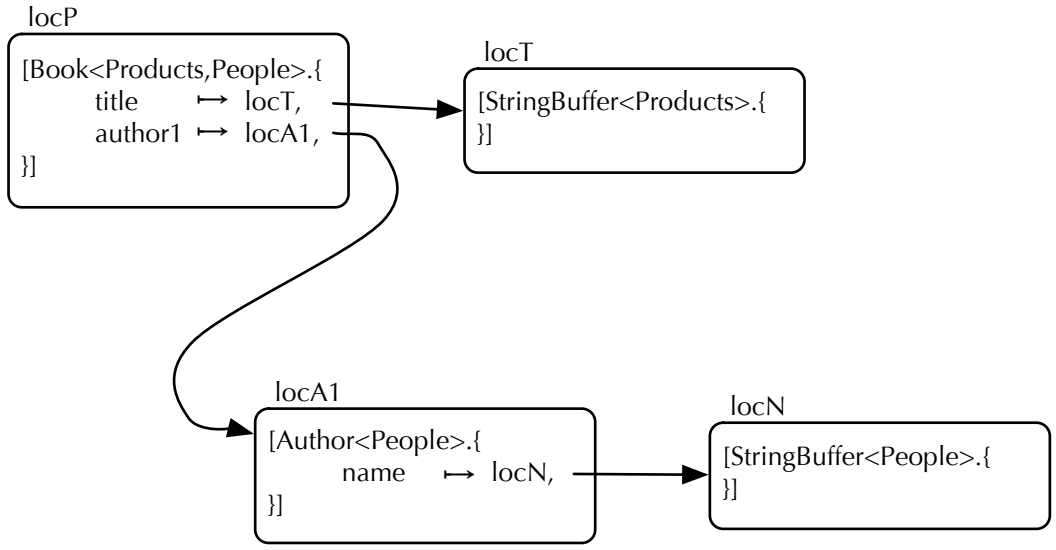

Figure 4.24 Schematic View of a Sample Store, S

Figure 4.24 shows a small MiniMAO$_2$ store. Table 4.1 on the facing page compares *reach*(S) and *writeReach*(S) for the store, S in the figure. Because the author1 field of Book is read-only, the pair $(\text{locP}, \text{locA1})$ that appears in *reach*(S) does not appear in *writeReach*(S), nor does the transitively induced pair $(\text{locP}, \text{locN})$.

Below is an example derivation of the object graph of the Book object in S, $\mathbb{G}_S(\text{locP})$:

i	E_i	L_i
0	$\emptyset$	$\{\text{locP}\}$
1	$\left\{\left(\text{locP} \overset{\text{title}}{\mapsto} \text{locT}\right), \left(\text{locP} \overset{\text{author1}}{\mapsto} \text{locA1}\right)\right\}$	$L_0 \cup \{\text{locT}, \text{locA1}\}$
2	$E_1 \cup \left\{\left(\text{locA1} \overset{\text{title}}{\mapsto} \text{locN}\right)\right\}$	$L_1 \cup \{\text{locN}\}$
3	E_2	L_2

Thus,

$$\mathbb{G}_S(\text{locP}) = (L_2, E_2)$$
$$= \left(\{\text{locP}, \text{locT}, \text{locA1}, \text{locN}\}, \left\{\left(\text{locP} \overset{\text{title}}{\mapsto} \text{locT}\right), \left(\text{locP} \overset{\text{author1}}{\mapsto} \text{locA1}\right), \left(\text{locA1} \overset{\text{title}}{\mapsto} \text{locN}\right)\right\}\right)$$

In addition to these definitions about pointer patterns in the store, I use a notion of the locations "included" in an expression. Intuitively, any *loc* that appears syntactically in e is included in e—whether the *loc* appears

Table 4.1 Reach and Writable Reach for the Store, S, of Figure 4.24

Source	Elements of *reach*(S)	Elements of *writeReach*(S)
From pointers	(null,null), (locP,locT), (locP,locA1), (locA1,locN),	(null,null), (locP,locT), (locA1,locN),
By transitivity	(locP,locN),	
By reflexivity	(locP,locP), (locT,locT), (locA1,locA1), (locN,locN)	(locP,locP), (locT,locT), (locA1,locA1),(locN,locN)

in an expression, in a join point abstraction, or in an advice body tuple. This is formalized by the following definition and associated figure.

Definition 4.27 (Included Locations). Given an expression e, the set of locations *included* in e, denoted *locations*(e), is given by the recursive definition in Figure 4.25 on the next page.

Finally, I formalize the notion of home domain.

Definition 4.28 (Home Domain). Given a store, S, and a location, loc, with $loc \in dom(S)$ such that $S(loc) = \left[T\langle g_1,\ldots,g_n\rangle \,.\, F\right]$, the *home domain* of loc in S, denoted $home_S(loc)$, is g_1.

The next theorem asserts that, for a certain sort of MiniMAO$_2$ program, no mutation is possible by dereferencing a readonly pointer. This property is different than that of Theorem 4.23 (Tag Frame Soundness) in that this property says that a read-only pointer may not be used for mutation even if it points to a writable domain. This theorem ensures that code in the right sort of MiniMAO$_2$ program cannot change the value of an object to which it does not have a write-enabled pointer. I will come back to the issue of the sorts of MiniMAO$_2$ programs considered by the theorem.

As for Theorem 4.23 (Tag Frame Soundness), I first state a lemma for a single evaluation step and then extend it for a sequence of steps. (The restriction on the domains of the object in the statement of the lemma, and subsequent theorem, is trivially satisfied for MiniMAO$_2$, because all concern domains are public. I include the restriction here so that the theory is also applicable to MiniMAO$_3$ in the subsequent chapter.)

Lemma 4.29 (Read-only Preservation). *Suppose the evaluation triple* $\langle \mathbb{E}[e], J, S\rangle$ *appears in the evaluation of a well-typed program P. Let loc be a location in dom(S) such that* $domains_S(loc) \subset \mathcal{G}$*, i.e., $S(loc)$ only names public concern domains. Let* $\mathbb{G}_S(loc) = (L, E)$*, and let the following assumptions hold:*

Assumption 1. $\forall \delta \cdot \left(loc_\delta \in locations(e)\right) \implies \left(\delta = readonly\right)$ *. (Intuitively, no write-enabled pointers to the object of interest appear in the expression.)*

Assumption 2. $\forall loc'_\delta \in locations(e) \cdot (\delta = \varepsilon) \implies \left(\forall loc'' \in rep_S(loc) \cdot \left(loc', loc''\right) \notin writeReach(S)\right)$ *. (Intuitively, the expression does not contain any write-enabled pointers that reach into the graph of the object of interest.)*

Assumption 3. $\forall loc' \in dom(S) \cdot S\left(loc'\right) = [t \,.\, F] \implies isClass(t)$ *. (No aspect instances appear in the store.)*

If $\langle \mathbb{E}[e], J, S\rangle \hookrightarrow \left\langle \mathbb{E}[e'], J', S'\right\rangle$*, then*

Consequent 1. $\forall \delta \cdot \left(loc_\delta \in locations\left(e'\right)\right) \implies \left(\delta = readonly\right)$

$$locations(\text{new } c\langle\ldots\rangle()) = \emptyset \qquad locations(var) = \emptyset \qquad locations\left(\text{null}_\delta\right) = \emptyset$$

$$locations(e_0.m(\,e_1,\ldots,e_n\,)) = \bigcup_{i\in\{0..n\}} locations(e_i) \qquad locations\left(e.f\right) = locations(e)$$

$$locations\left(e_0.f{=}e_1\right) = locations(e_0) \cup locations(e_1) \qquad locations(\text{cast } t\ e) = locations(e)$$

$$locations(e_0;\ e_1) = locations(e_0) \cup locations(e_1) \qquad locations\left(e_0.\text{proceed}(\,e_1,\ldots,e_n\,)\right) = \bigcup_{i\in\{0..n\}} locations(e_i)$$

$$locations\left(loc_\delta\right) = \left\{loc_\delta\right\} \qquad locations((\,l\,(\,e_0,\ldots,e_n\,))) = locations(l) \cup \bigcup_{i\in\{0..n\}} locations(e_i)$$

$$locations\left(\langle e\rangle_{\delta,\hat{\gamma}}\right) = locations(e) \qquad locations\left(\text{jointpt } j\,(\,e_0,\ldots,e_n\,)\right) = locations\left(j\right) \cup \bigcup_{i\in\{0..n\}} locations(e_i)$$

$$locations\left(\text{under } e\right) = locations(e)$$

$$locations\left(\text{chain } \bar{B}, j(\,e_0,\ldots,e_n\,)\right) = locations\left(\bar{B}\right) \cup locations\left(j\right) \cup \bigcup_{i\in\{0..n\}} locations(e_i)$$

$$locations\left(\text{fun } m\langle\ldots\rangle.e:\tau\,.\,\hat{\gamma}\right) = locations(e) \qquad locations(\bullet) = \emptyset$$

$$locations\left([\![b, loc, e, \sqcup, \sqcup, \sqcup]\!] + \bar{B}\right) = locations(b) \cup \{loc\} \cup locations(e) \cup locations\left(\bar{B}\right)$$

$$locations(\langle\alpha,\ldots\rangle) = locations(\alpha) \qquad locations\left(var \mapsto loc_\delta\right) = \left\{loc_\delta\right\} \qquad locations(-) = \emptyset$$

Figure 4.25 Recursive Definition of the Locations Included in an Expression

Consequent 2. $\forall loc'_\delta \in locations\left(e'\right) \cdot (\delta = \varepsilon) \implies \left(\forall loc'' \in rep_S(loc) \cdot \left(loc', loc''\right) \notin writeReach(S')\right)$

Consequent 3. $\forall loc' \in dom\left(S'\right) \cdot S'\left(loc'\right) = [t\,.\,F] \implies isClass(t)$

Consequent 4. $\mathbb{G}_S(loc) = \mathbb{G}_{S'}(loc)$,

Proof. Let e_{left} be the current redex of $\mathbb{E}[e]$. Let $\mathbb{E}'[-]$ be defined such that $\mathbb{E}[e] = \mathbb{E}[\mathbb{E}'[e_{\text{left}}]]$. (That is, $\mathbb{E}'$ is the unique evaluation context matching the expression e down to the current redex.) Let e_{right} be such that $\mathbb{E}[\mathbb{E}'[e_{\text{right}}]] = \mathbb{E}[e']$. By $\vdash P$ OK, Theorem 4.17 (Subject Reduction), and Theorem 4.18 (Progress), we know that $\mathbb{E}[\mathbb{E}'[e_{\text{left}}]]$ is well typed. Therefore e_{left} is also well typed.

Consequent 2 implies consequent 1. To see this note that $loc \in L$ by the definition of $\mathbb{G}_S(loc)$, and $(loc, loc) \in writeReach(S')$ by reflexivity. Thus if $loc_\delta \in locations\left(e'\right)$, then $\delta = \text{readonly}$.[7]

So we must show that consequents 2, 3, and 4 hold. For each of these, we must consider all of the possible evaluation rules that might generate the evaluation step assumed by the theorem. For each rule we must show that these consequents of the lemma hold in the result.

Case 1—NEW. $e_{\text{left}} = \text{new } t()$, $e_{\text{right}} = loc''$, $loc'' \notin dom(S)$, $S' = S \oplus \left(loc'' \mapsto [t\,.\,F]\right)$, and $rng(F) = \{\text{null}\}$. Because loc'' is fresh and S' only differs from S by the addition of a mapping for loc'', $\left(loc, loc''\right) \notin reach(S')$. Therefore loc'' is not a node in $\mathbb{G}_{S'}(loc)$, and consequent 4 holds. Because $rng(F) = \{\text{null}\}$, consequent 2 holds. Syntactically, only classes may be instantiated by expressions, so consequent 3 holds.

[7] Technically, I could omit assumption 1 (and consequent 1). However, it is useful to make the fact of this assumption explicit.

Case 2—CALL$_A$. Here $S' = S$ and $locations\left(e_{left}\right) = locations\left(e_{right}\right)$, so all the consequents hold.

Case 3—CALL$_B$. $e_{left} = \mathsf{chain}\ \bullet, (\!|\mathsf{call}, -, m, -, \tau, \hat{\gamma}|\!)(\ loc''_\delta, v_1, \ldots, v_n\ )$, $e_{right} = (\ l\ (\ loc''_\delta, v_1, \ldots, v_n\ ))$, and $S' = S$. Because $S' = S$, consequents 3 and 4 hold. Now $locations\left(e_{right}\right) = locations(l) \cup locations\left(e_{left}\right)$. But l is retrieved by the *methodBody* auxiliary function and $locations(l)$ is just the set of locations in the body of some method. Method bodies are typed using an environment without any locations. Thus $locations(l) = \emptyset$, $locations\left(e_{right}\right) = locations\left(e_{left}\right)$, and consequent 2 holds.

Case 4—EXEC$_A$. Here $S' = S$ and $locations\left(e_{left}\right) = locations\left(e_{right}\right)$, so all the consequents hold.

Case 5—EXEC$_B$. $e_{left} = \mathsf{chain}\ \bullet, (\!|\mathsf{exec}, v, m, l, \tau, \hat{\gamma}|\!)(\ v_0, \ldots, v_n\ )$, $e_{right} = \mathsf{under}\ \langle e'' \{\!| v_0 / var_0, \ldots, v_n / var_n |\!\}\rangle_{\delta', \hat{\gamma}}$, and $S' = S$. Because $S' = S$, consequents 3 and 4 hold.

The expression e'' in e_{right} is the body of the method from l. Substituting into e'' could cause some of $v_0, \ldots, v_n$ to drop from e_{right}. Also v from the join point abstraction does not appear in e_{right}. No new locations are added. So we have $locations\left(e_{right}\right) \subseteq locations\left(e_{left}\right)$ and $S' = S$. So consequent 2 holds.

Case 6—GET. $e_{left} = loc''_\delta.f$, $e_{right} = v_{\delta'}$, $S\left(loc''\right) = [T\langle\ldots\rangle\,.\,F]$, $readonly\left(fieldsOf\left(\delta\ T\langle\ldots\rangle\right)(f)\right) = \delta'$, $F(f) = v$, and $S' = S$. Because $S' = S$, consequents 3 and 4 hold. For consequent 2, let

$$readonly\left(fieldsOf\left(T\langle\ldots\rangle\right)(f)\right) = \delta''$$

(the declared read-only status of the field). Then by the definitions of *readonly* and *fieldsOf*:

$$\delta' = \begin{cases} \mathsf{readonly} & \text{if } \delta = \mathsf{readonly} \text{ or } \delta'' = \mathsf{readonly} \\ \varepsilon & \text{otherwise} \end{cases}$$

Now if $v = \mathsf{null}$, consequent 2 holds trivially.

If $v = loc$, then $\left(loc'', loc\right) \in reach(S)$. By the assumptions of the lemma, either $\delta = \mathsf{readonly}$ or else $\left(loc'', loc\right) \notin writeReach(S)$, which implies $\delta'' = \mathsf{readonly}$. In either case $\delta' = \mathsf{readonly}$ and consequent 2 holds.

Finally, suppose $v = loc' \neq loc$. If $\delta' = \mathsf{readonly}$, consequent 2 holds trivially. On the other hand, if $\delta' = \varepsilon$ we must show that $\left(loc', loc_L\right) \notin writeReach(S)$ for all $loc_L \in L = rep_S(loc)$. But $\delta' = \varepsilon$ implies $\delta = \delta'' = \varepsilon$. Because $v = loc'$, we know $\left(loc'', loc'\right) \in reach(S)$. Because $\delta'' = \varepsilon$, we know that $\left(loc'', loc'\right) \in writeReach(S)$. For purposes of showing a contradiction, choose an arbitrary $loc_L \in L$ and suppose $\left(loc', loc_L\right) \in writeReach(S)$. Then, because *writeReach* is transitive, $\left(loc'', loc_L\right) \in writeReach(S)$. But this contradicts assumption 2 of the lemma, so it must be that $\left(loc', loc_L\right) \notin writeReach(S)$ and consequent 2 holds.

Case 7—SET. $e_{left} = \left(loc''.f{=}v\right)$, $e_{right} = v$, $S\left(loc''\right) = [t\,.\,F]$, $S' = S \oplus \left(loc'' \mapsto \left[t\,.\,F \oplus (f \mapsto v')\right]\right)$, and

$$v' = \begin{cases} loc' & \text{if } v = loc'_{\delta'} \\ \mathsf{null} & \text{otherwise} \end{cases}$$

(Since e_{left} is well typed, we can omit any δ-subscript on loc''.)

The only object changed in S' versus S is $S\left(loc''\right)$. Its type is not changed, so consequent 3 holds. To prove consequent 4, it suffices to show that $loc'' \notin L = rep_S(loc)$. Suppose not, i.e., suppose $loc'' \in L$. This implies, by assumption 2 of the lemma, that $\left(loc'', loc''\right) \notin writeReach(S)$. But because $S\left(loc''\right)$ has a write-enabled field, f, we know $\exists v'' \cdot \left(loc'', v''\right) \in writeReach(S)$. By reflexivity of *writeReach*, $\left(loc'', loc''\right) \in writeReach(S)$. Thus our supposition leads to a contradiction, $loc'' \notin L$, and consequent 4 holds.

For consequent 2, we have a modified store and $writeReach(S')$ differs from $writeReach(S)$. If $v' = \mathsf{null}$, then $\left(loc'', \mathsf{null}\right)$ is the only element possibly in $writeReach(S') \setminus writeReach(S)$, so consequent 2 holds. If $v' = loc'$ and $\delta' = \mathsf{readonly}$, then by T-SET and the definition of subtyping, $readonly\left(fieldsOf(t)(f)\right) = \mathsf{readonly}$. So $writeReach(S') \setminus writeReach(S) = \emptyset$ and consequent 2 holds. Finally, if $v' = loc'$ and $\delta' = \varepsilon$, then $writeReach(S')$ is the reflexive, transitive closure of $writeReach(S) \cup \left(loc'', loc'\right)$, less whatever pairs were induced by the pair $\left(loc'', F(f)\right)$. Suppose this process results in a pair $(loc_e, loc_L) \in writeReach(S')$ that violates consequent 2. Then it must be the case that this pair is induced by the transitive closure with $\left(loc_e, loc''\right)$ and $\left(loc', loc_L\right)$ both in $writeReach(S)$. This second inclusion violates assumption 2 of the lemma, since $v = loc' \in locations(e)$. By this contradiction, there exists no pair $(loc_e, loc_L) \in writeReach(S')$ that violates consequent 2, and the claim holds.

Case 8—CAST. Here $S' = S$ and $locations\left(e_{\text{left}}\right) = locations\left(e_{\text{right}}\right)$, so all the consequents hold.

Case 9—NCAST. Here $S' = S$ and $locations\left(e_{\text{left}}\right) = locations\left(e_{\text{right}}\right)$, so all the consequents hold.

Case 10—SKIP. $e_{\text{left}} = (v;\ e)$, $e_{\text{right}} = e$, $S' = S$. Because $S' = S$, consequents 3 and 4 hold. Also

$$locations\left(e_{\text{right}}\right) \subseteq locations\left(e_{\text{left}}\right),$$

so consequent 2 holds.

Case 11—BIND. $e_{\text{left}} = \mathsf{joinpt}\ j\ (\ v_0, \ldots, v_n\ )$, $e_{\text{right}} = \mathsf{under\ chain}\ \bar{B}, j\ (\ v_0, \ldots, v_n\ )$, $\bar{B} = adviceBind(j + J)$, and $S' = S$. Because $S' = S$, consequents 3 and 4 hold.

For explanatory purposes, I briefly consider the case as if assumption 3 did not hold. Then

$$locations\left(e_{\text{right}}\right) \supseteq locations\left(e_{\text{left}}\right).$$

By the definitions of *locations* and *adviceBind*, $locations\left(e_{\text{right}}\right)$ includes the locations of the aspects of any matching advice. Advice body expressions in $\bar{B}$ do not contribute any locations, by a similar argument to that for method bodies in Case 3. The other possible source of new locations for $locations\left(e_{\text{right}}\right)$ is the binding terms in $\bar{B}$. In particular, the left-most join point abstraction in $j + J$ of the form $\langle\!\langle \_, v, \_, \_, \_, \_ \rangle\!\rangle$ may contribute v to $locations\left(e_{\text{right}}\right)$ because of a this pointcut descriptor.

But by assumption 3 and the validity of S, there can be no matching advice—there is no advice at all. Thus, $\bar{B} = \bullet$, $locations\left(e_{\text{right}}\right) = locations\left(e_{\text{left}}\right)$, and consequent 2 holds.

Case 12—ADVISE. Here

$$\begin{aligned} e_{\text{left}} &= \text{chain } [\![b, loc_{\text{a}}, e_{\text{a}}, \hat{\gamma}, \tau, \lrcorner]\!] + \bar{B}, j \ (\ v_0, \ldots, v_n\ ) \\ e_{\text{right}} &= \text{under } \langle\!\langle e_{\text{a}} \rangle\!\rangle_{\bar{B},j} \{\!| loc_{\text{a}} / \text{this} |\!\}\{\!|(v_0, \ldots, v_n)/\, b|\!\}\rangle_{\delta,\hat{\gamma}} \\ readonly(\tau) &= \delta \\ S' &= S \end{aligned}$$

Because $S' = S$, consequents 3 and 4 hold. Examining the definitions of advice chaining and binding substitution, we see that no new locations are introduced. Some locations may be dropped if not all formals appear in e_{a}. So $locations\left(e_{\text{right}}\right) \subseteq locations\left(e_{\text{left}}\right)$, and consequent 2 holds.

Case 13—UNDER. Here $S' = S$ and $locations\left(e_{\text{left}}\right) = locations\left(e_{\text{right}}\right)$, so all the consequents hold.

Case 14—TAG. $e_{\text{left}} = \langle v \rangle_{\delta,\hat{\gamma}}$, $e_{\text{right}} = v_\delta$, and $S' = S$. Because $S' = S$, consequents 3 and 4 hold. The only way in which $locations\left(e_{\text{right}}\right) \neq locations\left(e_{\text{left}}\right)$ is if $v = loc'$ and $\delta = \text{readonly}$. Then

$$locations\left(e_{\text{right}}\right) = \left\{ loc'_{\text{readonly}} \right\}.$$

Clearly consequent 2 holds. □

Theorem 4.30 (Read-only Soundness). *Suppose the evaluation triple $\langle \mathbb{E}[e], J, S \rangle$ appears in the evaluation of a well-typed program P. Let loc be a location in $dom(S)$ such that $domains_S(loc) \subset \mathcal{G}$, i.e., $S(loc)$ only names public concern domains. Let $\mathbb{G}_S(loc) = (L, E)$, and let the following assumptions hold:*

Assumption 1. $\forall \delta \cdot \left(loc_\delta \in locations(e)\right) \implies \left(\delta = readonly\right)$.

Assumption 2. $\forall loc'_\delta \in locations(e) \cdot (\delta = \varepsilon) \implies \left(\forall loc'' \in rep_S(loc) \cdot \left(loc', loc''\right) \notin writeReach(S)\right)$

Assumption 3. $\forall loc' \in dom(S) \cdot S\left(loc'\right) = [\,t \,.\, F] \implies isClass(t)$.

If $\langle \mathbb{E}[e], J, S \rangle \overset{*}{\hookrightarrow} \left\langle \mathbb{E}[v], J', S' \right\rangle$, *then* $\mathbb{G}_S(loc) = \mathbb{G}_{S'}(loc)$.

Proof. Immediate by appealing to Lemma 4.29 (Read-only Preservation) at each step in the evaluation. □

Theorem 4.30 (Read-only Soundness) allows aliasing in the analyzed code, so long as all the aliases are read-only. With a more general ownership type system, such as that of Aldrich and Chambers [9], the restrictions on aliasing might be relaxed. But the theorem imposes another condition that is quite restrictive: no aspects can be used in the program!

The basic issue is that aspects can "leak" pointers into the computation without being explicitly referenced. (In the proof of Lemma 4.29 (Read-only Preservation) on page 191, this leakage would be in the advice list, $\bar{B}$; see Case 11.) Thus the restrictions on aliasing in assumption 2, which are sufficient without aspects, are not sufficient in the presence of aspects. One might think that a design that makes all pointers to aspects read-only might solve the leakage problem. Unfortunately, that would render all aspects immutable. Alternatively, assumption 2 of the theorem might be changed to let loc'_δ range over all locations included in e, plus the location of every aspect. This would prevent the problems that arise in the BIND case of the proof of the lemma and would correspond to the sort of reasoning required with explicitly accepted assistance (as discussed in

Chapter 2). In the subsequent chapter, I show how spectator aspects, appropriately defined, avoid the leakage problem. The problem will remain for regular (assistant) aspects, but that seems to be a price of their greater expressive power. Adding a full-blown alias control system to MiniMAO might allow more control over aliasing between assistant aspects and base program objects. On the other hand, the power of assistant aspects might break the more powerful alias control system also. I leave that investigation to future work.

4.5 Related Work

Aldrich and Chambers [9] present an ownership type system that is decoupled from the encapsulation relation in a program. Their system allows very fine-grained specification, and static typechecking, of the aliasing relationships in a program. The system replaces the traditional owners-as-dominators property of ownership type systems with a link soundness property. The link soundness property says that the only interdomain aliases are those between "ownership domains" that are explicitly given permission to hold such aliases. These permissions are closely related to my concern domain dependency declarations. The authors' ownership domains are significantly more fine-grained than my concern domains, with each *object* having its own member domains. Their system includes a single global domain, called "shared", to which objects belong by default. Other global domains can be declared in their system by using public member domains within singleton objects. Such global domains are necessary for the reasoning I want in MiniMAO$_2$. I considered just adopting their type system for MiniMAO$_2$. However, the design of MiniMAO$_2$, lacking static fields, does not allow singleton objects to be coded in a natural way. Furthermore, their system is not designed to control mutation and does not distinguish between read-only and write-enabled pointers. Ownership domains is, perhaps, the most elegant of a large class of ownership and alias control type systems [9, 10, 25, 35, 116, 117, 118, 121]. Aldrich and Chambers [9] provide a solid summary of the work in this area.

Dantas and Walker [48] present a calculus for "harmless advice", based on an extension of the typed lambda calculus, with references and objects added in the style of Abadi and Cardelli's imperative object calculus [1]. Dantas and Walker use a type system with "protection domains" to keep aspects from altering the data of the base program. In keeping with this non-interference property, they do not allow advice to change values when proceeding to the base program. They borrow the lattice-ordered protection domains from the secure programming languages community [119, 139] to prevent lower integrity data, such as that generated by advice, from interfering with higher integrity data, such as that from the base program. These protection domains in their core calculus are more expressive than my concern domains. Protection domains define a partial order on data, whereas concern domains define a partition.

While the protection domains in the core calculus of Dantas and Walker are more expressive than concern domains, this expressiveness is relinquished in their user-level calculus. Their user-level calculus generates a single protection domain for the base program and a separate protection domain for each declared aspect. Thus the protection system is tied to the program structure and, unlike MiniMAO$_2$, cannot represent designs where the protection domains cross-cut the modularity structure of the program. (In the terms of Chapter 2, they cannot represent surprising assistants.) They also do not include classes in either their core calculus or user-level language, making their calculus a poor match for studying reasoning issues in AspectJ-like languages.

Leino [97] introduces data groups as a mechanism for abstractly expressing frame conditions for methods in a way that is compatible with both modular verification and subclassing. Data groups are programmer-defined hierarchical sets. Fields are declared to belong to a data group. A method declared to modify a particular data group may modify all fields in that data group and in any data groups in the downward closure. Subclasses can add fields to existing data groups and override methods to modify the new fields.

Leino's data groups abstract from the set of possible side effects inside a given module. My concern domains decouple Leino's abstraction over effects from the module hierarchy. This is analogous to the decoupling in Aldrich and Chamber's ownership domains relative to previous ownership type systems.

Techniques like concern domains have been used to attack reasoning issues in object-oriented languages more directly. Leino and Nelson [99] describe the use of static and dynamic dependencies in the Extended Static Checker project for Modula-3. Their specification language allows abstract, behavioral specification of methods. Key to this is the use of abstract variables [96], which are directly related to data groups [97]. But rather than just restricting the state that may be mutated by a method, as in data groups and concern domains, Leino and Nelson's specification language allows pre- and postconditions to refer to the *value* of the specification variables. These values are calculated using representation functions, which describe how to calculate the value of an abstract variable from the concrete state of a module. Static and dynamic dependencies are the key to avoid exposing the representation function of an abstract variable. These dependencies are "abstraction(s) of abstraction function(s)" (§13). The dependencies describe which fields are used in the representation function of the abstract variable. This explicit declaration of dependencies seems to be necessary for modular verification. This paper introduces a notion of "scope monotonicity"—any property proven in some scope is provable in any larger scope—as the key property in its notion of modular reasoning.

The language feature of abstract variables, and their static and dynamic dependencies, is orthogonal to concern domains. Concern domains are used to describe the large-scale structure of a program. Abstract variables are used for finer grained specifications. It is interesting to consider combining the two features. For example, what are the implications of declaring abstract fields as belonging to a particular concern domain?

Leino et al. [100] describe a technique for "specifying and statically checking the side effects of methods" in a modular way. Their core language, OOLONG, includes annotations for specifying data groups (i.e., static dependencies, inclusions (i.e., dynamic dependencies), and modifies lists (i.e., frame axioms). Two rather draconian referencing restrictions, pivot uniqueness and owner exclusion, are used to avoid problems that arise because of aliasing. However, with these restrictions, they are able to modularly and soundly verify methods. The modularity property is based on scope monotonicity [99].

Leino and Müller [98] describe a small language, with specification constructs, in which it is possible to modularly verify object invariants in the presence of aliasing. The system builds on Universes [117] to separate objects into hierarchical ownership contexts. Representation objects are confined within the context of a single owner object. However, Leino and Müller add a notion of "ownership transfer" to allow code to move objects between ownership contexts. The ownership transfer mechanism is cumbersome, involving packing and unpacking object graphs so that whole subcontexts are moved. An object cannot be owned by two other objects simultaneously. It is interesting to consider how their work might be combined with MiniMAO$_2$, particularly since the readonly annotations introduced in MiniMAO$_2$ are a simplification of those from Universes.

Besides these approaches that, like concern domains, restrict the flow of data in programs, there are other approaches that focus on limiting the join points to which advice may attach. Among these are Open Modules [8] (discussed in Section 3.3) and Aspectual Collaborations [102] (discussed in Section 2.6).

Another approach is to analyze and classify the sorts of interaction that may occur between aspects and the base program, but without applying any *a priori* restrictions on the sorts of interaction. The approach is exemplified by the work of Rinard et al. [146]. That paper presents a system that uses a simple control flow analysis, and a global pointer and escape analysis, to classify the interactions between advice and advised methods. The pointer analysis generates a "scope" for each code block describing the fields that are read or written by that block. The granularity is per-class, not per-object. The paper is a direct extension of the

ideas presented by Leavens and me [39]. Its contributions are the implementation of a static analysis and a more fine-grained classification system. Concern domains and effects clauses in MiniMAO$_2$ can be viewed as lifting the per-method and per-aspect information from their static analysis into the type system. Dependency declarations lift their globally detected interference into the type system. Instead of detecting the interference as a global property of a program, MiniMAO$_2$ lets the programmer explicitly specify the planned interaction of concerns. The type system then verifies the programmer's intent. The analysis of Rinard et al. [146] also considers interference by analyzing the pattern of read access to the heap. It would be straightforward to add read clauses to methods and advice in MiniMAO$_2$ to lift this analysis into its type system. Though intuitively the concern domains of the self object for a method or piece of advice (modulo dependency closure) place an upper bound on the readable concern domains. Static typechecking ensures that no other concern domains may be mentioned in the code. I leave the proof of this property and an investigation of the utility of explicit reads clauses to future work.

Lam and Rinard [88] present an object-oriented system, implemented as a Java extension, that lets programmers annotate objects with "tokens" and methods with "subsystem" designations. Multiple objects may have the same token and separate instances of the same class may have different tokens. A simple whole-program analysis can determine token propagation and referencing patterns, and subsystem interactions. Based on this analysis several sorts of design summary graphs can be produced, including:

— an abstract object model, which abstracts away the details of component objects into single nodes;

— a call/return model;

— a subsystem access (to token) model; and

— a heap interaction model, which details how subsystems communicate through shared objects.

The generated models are sound: only interactions shown in the models may actually occur at runtime. The analysis is completely static: no token or subsystem annotations are used at runtime. Given their system's whole-program analysis, one would expect that it should accommodate subtyping, but the presented formal type system does not seem to address this.

The concern domains in MiniMAO$_2$ are a generalization of Lam and Rinard's design tokens. Their work does not focus on the enforcement of concern separation, but on generating abstract models of concern interactions. Their subsystems partition the control flow graph of a program. In its current incarnation, MiniMAO$_2$ does not address control flow graphs. However, one can imagine an extension in which concern domains are treated as a generalization of both tokens and subsystems. Subsystems in their work are not polymorphic; i.e., every instance of a class belongs to the same subsystem. It may be that subsystem polymorphism is necessary in an aspect-oriented system. For example, if we are advising collection class objects, we may need to differentiate between the subsystems to which these objects belong. Lam and Rinard [88] simply omit subsystem annotations on such general purposes classes; the subsystem attribute of a control flow simply retains its previous value when control passes to an instance of an annotation-free class.

Rajan and Sullivan [141] introduce "classpects", which confound classes and aspects, in their Eos-U language. Because classpects contain advice-like constructs and can be instantiated, Eos-U bears some similarities to MiniMAO$_2$ with its aspect instantiation instructions. However, because classpects can be dynamically instantiated throughout the execution of a program, they do not confer the modular reasoning benefits of MiniMAO$_2$.

In MiniMAO$_2$, I introduce the writes pointcut descriptor, which allows advice to match based on the data that a method might mutate. This provides a kind of "semantic" pointcut [78], matching more on the meaning

of the method than on the pattern of names in its signature. Several others have proposed mechanisms for more semantic pointcut descriptors [34, 56, 67, 107], though none of these considers the frame conditions as an advice matching criteria.

Another line of work related to concern domains is what Reynolds [143] termed "separation logic". Separation logic is related to the logic of "bunched implications", introduced by O'Hearn and Pym [126]. These two ideas were brought to bear on the problem of local reasoning by O'Hearn et al. [127]. Separation logic extends Hoare logic [71] to reason about programs with mutable storage. The central idea in separation logic is to separate the predicates in a "spatial conjunction" so that each refers to an unconnected, disjoint subset of the heap, where "unconnected" means the absence of pointers from one subset to the other. This unconnectedness requirement is related to the restrictions on aliasing in Theorem 4.30 (Read-only Soundness). Because concern domains make the connections between subsets in a partitioning of the heap explicit, it seems that they might provide a useful substrate for applying separation logic to aspect-oriented programming languages. I leave that investigation to future work. O'Hearn et al. [127, §8] also discuss a notion of "memory faults", run-time errors that are signaled when code accesses a portion of the heap outside of the subset described in the specification of the code. A proof of correctness for the code must ensure that such memory faults cannot occur. The static type system of MiniMAO$_2$ can ensure this property for write access. And, as discussed above, MiniMAO$_2$ also intuitively places an upper-bound on the set of domains that may be read by a piece of code. Work related to separation logic continues. Reynolds [144] provides a nice summary of the early work. More recent work related to concern domains includes that of Bornat et al. [23] and Parkinson and Bierman [131]. An interesting point made in the latter paper, is that pointers in separation logic can equivalently be treated as access permissions. This bolsters my assertion above that the concern domains of an object relate to methods of that object having "read" permission to those concern domains.

I discussed Kicazles and Mezini's "aspect-aware interfaces" [80] in Section 2.6. These interfaces are generated from a global configuration, which is outside the scope of the language. In the current work, concern domain declarations and concern annotations on fields and methods, serve to define this global configuration, but within the language. Unlike Kiczales and Mezini's work, concern domains also allow tool support to enforce the separation of concerns designated by the programmer. With explicit acceptance of assistance, via my proposed hierarchical concern maps, a finer-grained configuration would be obtained.

4.6 Conclusion

I conclude this chapter by revisiting some claims from its introduction.

CLAIM 1 MiniMAO$_2$ enables efficient static detection of tangled code by lifting cross-cutting concerns from the program implementation into the type system.

This claim rests on the assumption that programmers can define the concerns in a program by separating the program's state into concern domains. If that assumption is true, then cross-cutting concerns are tangled in exactly those declarations that reference multiple concern domains. It is not yet entirely clear that the assumption holds, however. Future work includes extending the ideas of MiniMAO$_2$ to a practical programming language so that full-scale case studies can be carried out.

CLAIM 2 The type system enforces a non-interference property so that a global, signature-level search can identify all the code that might mutate a particular concern domain.

As discussed in Section 4.4.3, Theorem 4.23 (Tag Frame Soundness) says that given

— the signature of a method or advice,

— the concern domains of the target object, and

— the configuration of aspects in the program (as represented by the aspect instantiation instructions and the dependency declarations),

one can determine all the concern domains that might be mutated. To do so, one must just take the dependency closure of the method or advice's effects clause. No additional code analysis is required, beyond the separate typechecking of the static type system. In my proposed language with concern maps and explicit acceptance of advice, the search scope could be further narrowed.

CLAIM 3 Read-only pointers serve as a proxy for the reasoning issues involved in combining more general alias-control type systems with an aspect-oriented language.

The read-only annotations in MiniMAO$_2$ are a simple alias-control mechanism. Rather than preventing aliasing, they prevent aliases from being used to "do harm". Lemma 4.29 (Read-only Preservation) on page 191 demonstrates that this simple alias-control system is sound in the base language. But the proof of the lemma provides convincing evidence that the system can fail in the presence of aspects if those aspects are ignored. This also provides theoretic motivation for spectators, which are the primary subject of the subsequent chapter.

CHAPTER 5. MiniMAO$_3$: SPECTATORS REALIZED

In the previous chapter, I introduced concern domains in MiniMAO$_2$ and proved that they effectively partition the store, allowing the type system to enforce the separation of concerns. I also introduced a simple alias-control mechanism in MiniMAO$_2$: readonly annotations. I proved that this alias-control mechanism is effective for programs without aspects. I argued that the problems in the alias-control system created by introducing aspects demonstrated the reasoning difficulties concomitant with AspectJ-style around advice.

In this chapter I describe *MiniMAO$_3$*. MiniMAO$_3$ formalizes spectator aspects as discussed in Chapter 2. I give the formal definition of MiniMAO$_3$ (as a set of differences versus MiniMAO$_2$) and prove that the meta-theory for MiniMAO$_2$ also holds for the new calculus. More importantly, I prove that the alias-control mechanism of MiniMAO$_2$, which is ineffective in the presence of regular aspects, is effective in the presence of spectator aspects. This demonstrates that spectator aspects, unlike regular aspect, do not interfere with this reasoning property. Furthermore, I demonstrate that because spectators belong to private concern domains, one does not need to know about the spectators present in a program in order to reason about the program. Spectators can be used non-invasively without sacrificing modular reasoning.

5.1 Differences Versus MiniMAO$_2$

MiniMAO$_3$ has three main additions as compared to MiniMAO$_2$: spectator aspects, "surround" advice, and private concern domains. Spectator aspects are as described in Chapter 2. They provide a restricted form of advice that is statically known to not affect the code that they advise, in a well-defined way. To distinguish regular aspects and spectator aspects in this chapter, I will refer to the former as *assistants* and the latter as *spectators*.

I call the restricted form of advice for spectators, *surround advice*.[1] Surround advice is a form of around advice with limited capabilities. These capabilities correspond closely to those of "harmless advice" [48]. The body of a piece of surround advice consists of a *before part* and an *after part*. The following shows a simple piece of surround advice:

```
surround() : call( Object⟨loggee⟩ *(..) ) {
    this.log.append(''before'');              // before part
    proceed;                                  // mandatory proceed to advised code
    this.log.append(''after: '' + reply)      // after part
}
```

As the names imply, the before-part expression in a piece of surround advice is evaluated before the advised code, while the after-part expression is evaluated after it. The operational semantics, discussed below, ensures that surround advice always proceeds exactly once to the advised code (or subsequent advice), unless the

[1] The name "surround advice" is due to Lisa Laxson.

$$
\begin{aligned}
\mathit{decl} &::= \ldots \mid \text{spectator } a\langle \text{self}, G^* \rangle \ \{\ \mathit{field}^*\ \mathit{surr}^*\ \} \\
\mathit{surr} &::= \text{surround } (\ \mathit{form}^*\ ) : \mathit{pcd}\ \{\ e;\ \text{proceed};\ e\ \} \\
\mathit{form} &::= t\ \mathit{var}, \text{ where } \mathit{var} \notin \{\text{this}, \text{reply}\} \\
\gamma &::= \ldots \mid \text{self} \\
\mathit{domains} &::= \text{domain } g;, \text{ where } g \notin \mathscr{G}_{\text{self}} \\
\mathit{asp} &::= \text{use } a\langle g^* \rangle; \mid \text{use } a\langle \text{self}, g^* \rangle;, \text{ where } g \notin \mathscr{G}_{\text{self}}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{var} &\in \{\text{this}, \text{reply}\} \cup \mathscr{V}, \text{ where } \mathscr{V} \text{ is the set of variable names} \\
g &\in \mathscr{G} \cup \mathscr{G}_{\text{self}}, \text{ where } \mathscr{G} \text{ is the set of public concern domain names} \\
\mathscr{G}_{\text{self}} &= \{\text{self}_{\mathit{loc}} \cdot \mathit{loc} \in \mathscr{L}\}, \text{ the set of private concern domain names}
\end{aligned}
$$

Figure 5.1 Differences in Syntax of MiniMAO$_3$ vs. MiniMAO$_2$

before-part expression diverges. Furthermore, the arguments passed to the advised code are the original arguments. Surround advice cannot mutate the arguments and cannot pass along new arguments. The result returned from executing a piece of surround advice is the result of the advised code (or subsequent advice). The after-part expression has read-only access to the result value. Surround advice may capture the arguments to, and results from, the advised code, but only in read-only fields. From these restrictions it follows that the before- and after-parts of surround advice are evaluated solely for their side effects. Another way to think of surround advice is as paired before and after advice.

MiniMAO$_3$ also includes private concern domains. The operational semantics places each instance of a spectator its own, unique private concern domain. Only the spectator instance and any objects it creates, whether directly or transitively, may refer to the spectator's private concern domain. This notion is formalized in Definition 5.15 (Privacy Respecting Store) on page 228.

As in previous chapters, I describe the syntax, operational semantics, and static semantics of MiniMAO$_3$ by reviewing the differences versus the previous calculus.

5.1.1 Syntax of MiniMAO$_3$

Figure 5.1 gives the differences in the user syntax from MiniMAO$_2$ to MiniMAO$_3$. As would be expected, the syntax includes declaration forms for spectators and surround advice. A spectator declaration looks like an assistant declaration except for two changes: (1) the first concern domain variable is the special self variable, which represents the private concern domain of an instance of the spectator, and (2) instead of declaring around advice, the spectator declares surround advice.

Surround advice declarations also look like their counterpart in MiniMAO$_2$ except for three differences. As mentioned above, the body of a piece of surround advice consists of two expression, the before- and after-part expressions. In the syntax, these are separated by a proceed. This proceed is *not* an expression. It merely serves to syntactically separate the before and after parts, and as mnemonic for the semantics of surround advice. The second difference between surround- and around-advice declarations, is the lack of a return type in surround advice. The result of evaluating a piece of surround advice is the result of the advised code. Since surround advice is only evaluated for side effects, it does not have its own return type. The third difference stems from the fact that surround advice may only mutate the private concern domain of its host spectator.

Thus, all surround advice has an implicit effects clause, writes ⟨self⟩, which is omitted from the syntax.

In MiniMAO$_3$, the meta-variable, *var*, ranges over all variable names, the special this variable, and a new special variable: reply. The reply variable may be used in the after part of surround advice to refer to the result of the advised code (or of any subsequent advice). The static type system, plus the restriction on *var* in the *form* non-terminal of Figure 5.1, ensure that reply is only used in the after-part of surround advice.

The meta-variable γ, which ranges over concern domain names and concern domain variables in MiniMAO$_2$, also may denote the special self concern domain variable in MiniMAO$_3$.

The *domains* non-terminal in MiniMAO$_3$ bears an additional restriction not present in MiniMAO$_2$. The meta-variable g, which ranges over public concern domain names in MiniMAO$_2$, may also range over $\mathcal{G}_{\text{self}}$, the set of private concern domain names, in MiniMAO$_3$. However, the *domains* non-terminal, and aspect instantiation instructions discussed next, restrict g to just range over public concern domains. This restriction is part of the mechanism for keeping private concern domains private. Only the operational semantics may introduce private concern domain names into a computation. Private concern domain names have the form self$_{loc}$, where *loc* is the location in the store of the spectator instance associated with the named domain.

Assistants are only applied if a concern map says so. Concern maps are represented by aspect instantiation instructions in MiniMAO$_2$. Spectators could be applied more generally, because the advised code does not need to be aware of them. But in MiniMAO$_3$, some mechanism is needed to instantiate any public concern domain variables in a spectator declaration. Aspect instantiation instructions provide this. Unfortunately this confounds the use of aspect instantiation instructions as a mechanism for instantiating spectators and as a formal representation for concern maps. However, accepting this confounding allows me to avoid adding yet another additional form to the syntax.

5.1.2 Operational Semantics of MiniMAO$_3$

Figure 5.2 on the next page gives the differences in the operational semantics and supporting definitions of MiniMAO$_3$ versus MiniMAO$_2$. I describe these differences in the following subsections.

5.1.2.1 Syntax Extensions for the Operational Semantics of MiniMAO$_3$

Like the previous calculi, MiniMAO$_3$ extends the user syntax with an additional expression used by the operational semantics to track machine state. The new expression form, termed a *leap expression*, has the form $e_1 \curvearrowright e_2$. The semantics first evaluates e_1, then e_2 for its side effects, replacing any occurrences of reply in e_2 with the value of e_1. The result of the whole expression is the value arrived at from evaluating e_1—the value of e_1 "leaps" over the value of e_2. I use the leap expression to express the meaning of surround advice.

If MiniMAO$_3$ had a let form, I could use that to express the desired semantics of surround advice. The semantics of $e_1 \curvearrowright e_2$ is the same as:

$$\text{let reply}{=}e_1 \text{ in } (\, e_2; \text{ reply}\,).$$

However, since MiniMAO$_2$ does not have let, I choose to introduce the new, more concise expression form rather than general let expressions. This option also avoids introducing local variables and makes the special semantics of surround advice more explicit.

I use a new advice body tuple form, $[\![b, loc, (e_b, e_a), t]\!]_S$, to represent surround advice in chain expressions. I will refer to these as *surround-advice body tuples,* and will refer to the advice body tuples of MiniMAO$_2$ as *around-advice* ones. Like around-advice body tuples, the ones for surround advice include a binding term, b, and a location, *loc*, pointing to the host aspect instance in the store. The expression pair (e_b, e_a) represents the before- and after-part expressions, respectively (hence the subscripts). Unlike around-advice body tuples,

Syntax extensions:

$$e ::= \ldots \mid e \curvearrowright e \qquad \hat{g} \in \mathscr{P}\left(\mathscr{G} \cup \mathscr{G}_{\text{self}}\right)$$
$$B ::= \ldots \mid [\![b, loc, (e, e), t]\!]_{\text{S}} \qquad \hat{\gamma} \in \mathscr{P}\left(\mathscr{G} \cup \mathscr{G}_{var} \cup \{\text{self}\}\right)$$
$$t, s, u ::= \ldots \mid \top$$

Evaluation contexts:

$$\mathbb{E} ::= \ldots \mid \mathbb{E} \curvearrowright e$$

Evaluation rules:

$$\langle \mathbb{E}[\text{chain } [\![b, loc, (e_{\text{b}}, e_{\text{a}}), \_]\!]_{\text{S}} + \bar{B}, j(\, v_0, \ldots, v_n\,)], J, S\rangle \hookrightarrow$$
$$\left\langle \mathbb{E}[\text{under } \left(\left\langle e'_{\text{b}}\right\rangle_{\varepsilon,\{\text{self}_{loc}\}}; \text{ chain } \bar{B}, j(\, v_0, \ldots, v_n\,)\right) \curvearrowright \left\langle e'_{\text{a}}\right\rangle_{\varepsilon,\{\text{self}_{loc}\}}\right)], j' + J, S\right\rangle \quad \textsc{Surround}$$
$$\text{where } e'_{\text{b}} = e_{\text{b}}\{\!|loc/\text{this}|\!\}\{\!|(v_0, \ldots, v_n)/b|\!\}, \; e'_{\text{a}} = e_{\text{a}}\{\!|loc/\text{this}|\!\}\{\!|(v_0, \ldots, v_n)/b|\!\}, \text{ and}$$
$$j' = (\!|\text{this}, loc, -, -, -, -|\!)$$

$$\langle \mathbb{E}[v \curvearrowright e], J, S\rangle \hookrightarrow \langle \mathbb{E}[e\{\!|v/\text{reply}|\!\};\; v], J, S\rangle \qquad \textsc{Leap}$$

Advice binding:

$$adviceBind(J, S) = \bar{B}, \text{ where } \bar{B} \text{ is a smallest list satisfying}$$
$$\forall \left\langle loc, pcd, \left(e_{\text{b}}, e_{\text{a}}\right), t\right\rangle_{\text{S}} \in AT \cdot \left(\left(matchPCD_{\text{S}}(J, pcd, S) = b \neq \bot\right) \implies [\![b, loc, \left(e_{\text{b}}, e_{\text{a}}\right), t]\!]_{\text{S}} \in \bar{B}\right) \text{ and}$$
$$\forall \left\langle loc, pcd, e, \hat{\gamma}, \tau, \tau'\right\rangle \in AT \cdot \left(\left(matchPCD(J, pcd, S) = b \neq \bot\right) \implies [\![b, loc, e, \hat{\gamma}, \tau, \tau']\!] \in \bar{B}\right)$$

Subtyping:

$$t \preccurlyeq \top \qquad \frac{CT(c) = \text{spectator } a\langle \text{self}, G_2, \ldots, G_q\rangle \; \{\; \ldots \;\}}{a\langle \text{self}_{loc}, \gamma_2, \ldots, \gamma_q\rangle \preccurlyeq \text{Object}\langle \text{self}_{loc}\rangle}$$

Figure 5.2 Differences in the Operational Semantics of MiniMAO$_3$ vs. MiniMAO$_2$

those for surround advice do not include a set of writable concern domains; it is always $\{\text{self}_{loc}\}$. Surround-advice body tuples also omit the function types representing the type of the advice and the type of any proceed expressions in the advice. These function types are not needed for the semantics or meta-theory of surround advice. In place of these function types, a surround-advice body tuple just records a type, t, representing the return type of the advised code. This type information is used in the static semantics for typing the special reply variable in e_{a}.

MiniMAO$_3$ includes the type $\top$, which is a supertype of every type.[2] I use it for typing the pointcut descriptors of surround advice, which can safely use more general pointcut matching rules than those for around advice. I discuss this more in Section 5.1.3.4. The $\top$ type is not part of the user syntax; it cannot be used for a formal parameter, field, or cast type.

[2]Because subtyping is positionally invariant for concern domains, Object cannot serve as a top type in MiniMAO$_2$ or MiniMAO$_3$.

5.1.2.2 Evaluation in MiniMAO$_3$

Program evaluation in MiniMAO$_3$ is essentially the same as in MiniMAO$_2$. Like the surround-advice body tuples introduced for chain expressions in MiniMAO$_3$, the advice table also includes special 4-tuples for surround advice, $\langle loc, pcd, (e_b, e_a), t_\top \rangle_S$. The elements represent the spectator location (*loc*), surround advice pointcut description (*pcd*), before-part (e_b) and after-part (e_a) expressions, and the expected result type ($t_\top$). The "$\top$" subscript on t here is just a mnemonic to remind the reader that the expect result type might be $\top$—any type.

The initial store for the evaluation of a MiniMAO$_3$ program includes assistant instances as in MiniMAO$_2$. The initial store also includes spectator instances. The home domain of a spectator instance, stored in location *loc*, is the private concern domain self_{loc}. This is formalized in Definition 5.1 (Store Validity) on page 213. Unlike in MiniMAO$_2$, where evaluation of a program may begin with any valid initial store, evaluation of a program in MiniMAO$_3$ must start with a valid initial store that "respects privacy". Definition 5.15 (Privacy Respecting Store) on page 228 formalizes this property. The evaluation rules maintain the property (see Theorem 5.16 (Respect for Privacy)), so starting with a store that has the property ensures that the property always holds.[3]

MiniMAO$_3$ uses the same evaluation contexts as MiniMAO$_2$, plus one additional context for leap expressions (see Figure 5.2 on the facing page). The right-hand side of a leap expression is not evaluated until the left-hand side has been reduced to a value.

MiniMAO$_3$ includes two new evaluation rules. One new rule handles leap expressions. The other handles the case of the current redex being a chain expression with a surround-advice body tuple at the head of the advice list. Although MiniMAO$_3$ uses the BIND rule, and assorted rules for casts, unchanged from MiniMAO$_2$, a new definition of *adviceBind* and an extended subtyping relation affect these rules.

THE LEAP RULE The LEAP evaluation rule is straightforward. The value, v, on the left-hand side is substituted for reply in the expression, e, on the right-hand side. A sequence expression is used to let v "leap" over e.

THE SURROUND RULE The SURROUND rule appears a bit daunting. However, it just composes concepts already discussed. The basic scaffolding for the generated expression is a leap expression. This leap expression allows the result of the advised code to leap over the (discarded) result of the after-part.

On the left-hand side of the leap expression, is a sequence. The first term in the sequence evaluates the before-part expression for side effects. The second (chain) term in the sequence evaluates the advised code, or any subsequent advice in the chain.

The rule replaces the formal parameters in both the before- and after-part expressions with the appropriate actuals according to the binding term, b. These β-converted expressions— e'_b and e'_a—are tagged to indicate that only the private concern domain of the spectator may be mutated. Finally, the rule wraps the whole sordid mess in an under expression, to record that the spectator location has been pushed onto the join point stack.

OTHER CHANGES The other differences in MiniMAO$_3$ versus MiniMAO$_2$ that affect the evaluation rules are simple. The *adviceBind* auxiliary function in MiniMAO$_3$ (see Figure 5.2 on the preceding page) calls the new $matchPCD_S$ pointcut matching function, described below, to handle any surround advice records in the advice table; *adviceBind* continues to use *matchPCD* for around advice.

[3] Technically, respect for privacy could be included in the definition of store validity. I keep the concepts separate for expository purposes.

MiniMAO$_3$ extends the subtyping relation of MiniMAO$_2$ to make spectator instances subtypes of Object, in the appropriate private concern domain. This could potentially affect the cast rules, since a spectator instance could be up-cast to Object (though this serves no purpose and causes no harm). The subtyping relation in MiniMAO$_3$ also makes every type a subtype of $\top$. This does not affect the evaluation rules, because $\top$ cannot appear in a cast expression and no value may have the actual type $\top$.

5.1.2.3 Pointcut Matching for Surround Advice

Because of the restrictions on the behavior of surround advice as compared to around advice, a more general pointcut matching mechanism can be used for surround advice without sacrificing type safety. Surround advice can match more because it does less.

Figure 5.3 on the facing page gives the rules that define pointcut matching for surround advice, denoted by the function $matchPCD_S$.

Consider the first rule in the figure. This rule handles the call pointcut descriptor. It matches a join point abstraction where the return type, u, of the advised code is a subtype of t, the type that the pointcut descriptor names. The corresponding rule for around advice requires $u = t$. Why the difference?

In both around advice and surround advice, the result of the advised code may be used in the advice. Since the advice treats the result as having type t, it cannot bind to code where the result type, u, is a proper supertype of t. Or else the advice might call a method defined for t but not defined for the supertype. Thus, the semantics must require $u \preccurlyeq t$.

But unlike surround advice, around advice may also return some value other than the original result to the calling code. The caller expects that the result is a subtype of u. But the type system can only ensure that the result is a subtype of t. So if u were allowed to be a proper subtype of t when matching around advice, then the original caller might try to call some method on the result that is defined on u, but not on t. So for around advice, the semantics must require $t \preccurlyeq u$, and hence $t = u$. Because surround advice is not able to change the original result, letting surround advice match when $u \preccurlyeq t$ does not create type safety problems.

The rule for execution pointcut descriptors is just like the one for calls. Similar subtyping considerations apply to the rules for target and args. Though for these rules, it is the prohibition on surround advice replacing the arguments to the advised code that allows the more relaxed matching.

The rule for matching writes pointcut descriptors in surround advice requires that the set of writable concern domains of the advised code be a superset of the one given in the pointcut descriptor. As discussed in Section 5.1.3.4 below, this has no implications for the type system. Using superset matching allows surround advice to match any code that may mutate a particular concern domain, even if that code also may mutate other concern domains. The example discussed below demonstrates this. (One consequence of this design decision is that there is no way to specify a piece of surround advice that only matches pure methods; using writes $\langle\rangle$ would match any superset of the empty set, i.e., any set. This could be resolved be adding a pointcut descriptor that uses subset matching for writable concern domains, say onlyWrites. I omit this because it is technically uninteresting.)

Finally, the surround advice matching rules for this pointcut descriptors and for pointcut union, intersection, and negation exactly mimic those for around advice matching.

Figure 5.4 on page 208 shows a Logger spectator that takes advantage of the more general pointcut matching in MiniMAO$_3$. Consider the semantics of the pointcut description in the figure, for the spectator instance created in line 17. This pointcut description will match any call or execution of a method that

— has any target type in the Products domain (line 5),

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

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

$$matchPCD_S(\langle\!|\_, \_, \_, \_, \_, \hat{\gamma}|\!\rangle + J, \text{writes}(\ \hat{\gamma}'\ ), S) = \begin{cases} \langle -, - \rangle & \text{if } \hat{\gamma}' \subseteq \hat{\gamma} \\ \bot & \text{otherwise} \end{cases}$$

$$matchPCD_S(\langle\!|\_, v, \_, \_, \_, \_|\!\rangle + J, \text{this}(\ t\ var\ ), S)$$
$$= \begin{cases} \langle var \mapsto v_{\delta'}, - \rangle & \text{if } v = loc_\delta,\ S(loc) = [s.F], \text{ and } \delta\ s \preccurlyeq t \quad (\text{where } readonly(t) = \delta') \\ \bot & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

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

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

Figure 5.3 Pointcut Descriptor Matching for Surround Advice

```
1  spectator Logger⟨self, loggee⟩ {
2     StringBuffer⟨self⟩ log;
3
4     surround(readonly Object⟨loggee⟩ targ, readonly Object⟨loggee⟩ newVal) :
5           target(readonly Object⟨loggee⟩ targ)
6           && args(readonly Object⟨loggee⟩ newVal)
7           && writes(loggee) {
8        this.log.append(''Entering '' + targ + '' with '' + newVal); // before part
9        proceed;
10       this.log.append(''Exited '' + targ) // after part
11    }
12 }
13
14 ...
15
16 {
17    use Logger⟨self, Products⟩;
18    ...
19 }
```

Figure 5.4 Example Illustrating Relaxed Pointcut Matching for Surround Advice

— takes a single argument, also in the Products domain (line 6), and

— might mutate the Products domain (line 7).

Compare this to the example in Figure 4.3 on page 126, which must use the exact matching of around advice.

5.1.3 Static Semantics of MiniMAO$_3$

Most of the machinery necessary for static type safety in MiniMAO$_3$ already exists in MiniMAO$_2$. The new calculus uses generalized notions of the evaluation dependency table and dependency closure to accommodate private concern domains. The calculus includes typing rules for the new spectator and surround advice declaration forms, and it uses a new program typing rule that considers spectator instantiation. The expression typing rules also get minor tweaks, to accommodate surround-advice body tuples in chain expressions and the new leap expression form. I discuss all of these changes in the following subsections.

5.1.3.1 General Differences

The evaluation dependency table in MiniMAO$_2$ includes a reflexive pair (g, g) for every declared, public concern domain in the program. Because MiniMAO$_3$ does not have declarations for private concern domains, I must extend the evaluation dependency table with a pair $(\mathsf{self}_{loc}, \mathsf{self}_{loc})$ for every implicitly declared private concern domain, i.e., for every spectator instance. Clearly this does not affect the reflexive nature of the evaluation dependency table. Note that the self domains are private and spectators lack dependency clauses. So the new pairs also do not affect the transitive nature of the evaluation dependency table. The type of dependency tables for MiniMAO$_3$ must admit pairs of private domains. It is:

$$DT : \left(\mathcal{G} \cup \mathcal{G}_{var} \cup \mathcal{G}_{\mathsf{self}}\right) \rightarrow \left(\mathcal{G} \cup \mathcal{G}_{var} \cup \mathcal{G}_{\mathsf{self}}\right).$$

Writable domains dependency closure:

$$depClose_{DT}(\hat{\gamma}) = \{\gamma' \cdot \exists \gamma \in \hat{\gamma} \cdot (\gamma, \gamma') \in DT\} \cup \{\mathsf{self}_{loc} \cdot (\exists loc \in \mathscr{L} \cdot (\mathsf{self}_{loc}, \mathsf{self}_{loc}) \in DT)\},$$
$$\text{where } DT : (\mathscr{G} \cup \mathscr{G}_{var} \cup \mathscr{G}_{\mathsf{self}}) \to (\mathscr{G} \cup \mathscr{G}_{var} \cup \mathscr{G}_{\mathsf{self}}) \text{ is reflexive and transitive, and } \forall \gamma \in \hat{\gamma} \cdot (\gamma, \gamma) \in DT$$

Spectator predicate:

$$\frac{CT(a) = \text{spectator } a\langle \mathsf{self}, G_2, \ldots, G_n \rangle \ldots}{isSpectator(\delta\ a\langle \gamma_1, \ldots, \gamma_n \rangle)}$$

Figure 5.5 Auxiliary Functions for the Static Semantics of MiniMAO$_3$

The dependency closure auxiliary function in MiniMAO$_3$ includes all the private concern domains from the dependency table (see Figure 5.5). This reflects the fact that an unseen spectator may always modify its private concern domain. However, because a private concern domain may only be named within the representation of the spectator, other code can still not mutate it.

The last general difference in the static semantics is in type environments. In MiniMAO$_3$, a type environment, Γ, allows mappings like $\Gamma(\mathsf{self}_{loc}) = \mathsf{domain}$ and $\Gamma(\mathsf{self}) = \mathsf{domain}$.

5.1.3.2 Declaration Typing

Figure 5.6 on the following page gives the differences in the typing rules of MiniMAO$_3$ versus MiniMAO$_2$.

The T-PROG rule in MiniMAO$_3$ must handle instantiation of both assistants and spectators. To do this it uses two "helper" rules. The T-ASSTINST rule is for assistant instantiation; it just includes the hypotheses from the T-PROG rule in MiniMAO$_2$ that are used to check aspect instantiation instructions there. The T-SPECINST rule is for spectator instantiation; it ensures that only spectator instances use private concern domains.[4]

The T-SPEC rule for spectators is just like the T-ASP one for assistants (see Figure 4.14 on page 142), but it omits checks on dependency declarations, which spectators lack.

The T-SURR rule for surround advice is similar to the T-ADV rule for around advice. T-SURR uses a relaxed pointcut declaration typing judgment (indicated by the subscripted turnstile "$\vdash_S$") that places no constraints on the this, target, or args types of *pcd*. Section 5.1.3.4 below discusses this in more detail. T-SURR checks that both the before- and after-part expressions are well-typed, using a set of writable concern domains that just includes the private self domain. Because these expressions are only evaluated for their side effects, T-SURR puts no constraints on the types given to them. It is enough that the expressions are well typed. When checking the formal parameter types, the T-SURR rule uses an empty set of writable concern domain variables. This forces all formal parameters of the surround advice to be read-only, preventing the advice from mutating any of the arguments to be passed to the advised code.

5.1.3.3 Expression Typing

The T-PROG rule in MiniMAO$_3$ uses helper rules to differentiate between spectators and assistants. Similarly, the T-CHAIN rule uses helper rules—T-BOD and T-BODS—to differentiate between around-advice and surround-advice body tuples in the advice chain, $\bar{B}$. When all body tuples in $\bar{B}$ represent around advice, then

[4] It may be that assistants could also be allowed to have their own private concern domains. I have not yet considered either the safety or the utility of this generalization.

Declaration typing rules:

T-PROG (replaces rule from MiniMAO$_2$)

$$\frac{\forall i \in \{1..n\} \cdot \vdash decl_i \text{ OK} \qquad \forall i \in \{1..r\} \cdot \{g_1,\ldots,g_p\} \vdash asp_i \text{ OK} \qquad g_1 : \text{domain}, \ldots, g_p : \text{domain} \,.\, \{g_1,\ldots,g_p\} \vdash_{\overline{DT}} e : t \qquad DT = depTable(\{g_1,\ldots,g_p\}, \emptyset)}{\vdash decl_1 \ldots decl_n \ \{\ \text{domain } g_1;\ \ldots;\ \text{domain } g_p;\ asp_1 \ldots asp_r\ e\ \}\ \text{OK}}$$

T-ASSTINST

$$\frac{CT(a) = \text{aspect } a\langle G_1,\ldots,G_n\rangle \ldots \qquad \forall i \in \{1..n\} \cdot g_i \in \hat{g}}{\hat{g} \vdash \text{use } a\langle g_1,\ldots,g_n\rangle \text{ OK}}$$

T-SPECINST

$$\frac{CT(a) = \text{spectator } a\langle \text{self}, G_2,\ldots,G_n\rangle \ldots \qquad \forall i \in \{2..n\} \cdot g_i \in \hat{g}}{\hat{g} \vdash \text{use } a\langle \text{self}, g_2,\ldots,g_n\rangle \text{ OK}}$$

T-SPEC

$$\frac{DT = depTable(\{\text{self}, G_2,\ldots,G_q\}, \emptyset) \qquad \forall i \in \{1..p\} \cdot DT \vdash surr_i \text{ OK in } a\langle \text{self}, G_2,\ldots,G_q\rangle \qquad q \geq 1 \qquad \forall i \in \{1..n\} \cdot \{\text{self}\} \vdash t_i \text{ OK in } a\langle \text{self}, G_2,\ldots,G_q\rangle}{\vdash \text{spectator } a\langle \text{self}, G_2,\ldots,G_q\rangle\ \{\ t_1\ f_1;\ \ldots;\ t_n\ f_n;\ surr_1 \ldots surr_p\ \}\ \text{OK}}$$

T-SURR

$$\frac{\Gamma \vdash_S pcd : \_ \,.\, \_ \,.\, \_ \,.\, u_\top \,.\, V \,.\, V \qquad V = \{\text{self}\} \qquad \Gamma, \text{this} : a\langle \text{self}, G_2,\ldots,G_q\rangle, \,.\, \{\text{self}\} \vdash_{\overline{DT}} e_b : s_b \qquad \Gamma, \text{this} : a\langle \text{self}, G_2,\ldots,G_q\rangle, \text{reply} : \text{readonly } u_\top \,.\, \{\text{self}\} \vdash_{\overline{DT}} e_a : s_a \qquad \forall i \in \{1..n\} \cdot \emptyset \vdash t_i \text{ OK in } a\langle \text{self}, G_2,\ldots,G_q\rangle \qquad \Gamma = var_1 : t_1,\ldots,var_n : t_n, \text{self} : \text{domain}, G_2 : \text{domain},\ldots,G_q : \text{domain}}{DT \vdash \text{surround}(\ t_1\ var_1,\ldots,t_n\ var_n\ ) : pcd\ \{\ e_b;\ \text{proceed};\ e_a\ \}\ \text{OK in } a\langle \text{self}, G_2,\ldots,G_q\rangle}$$

Expression typing rules:

T-CHAIN (replaces rule from MiniMAO$_2$)

$$\frac{\forall i \in \{0..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{\overline{DT}} e_i : u_i \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq t_i \qquad depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (readonly(u_0) = \text{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \forall i \in \{1..p\} \cdot \Gamma \,.\, \hat{\gamma}' \,.\, \tau \vdash_{\overline{DT}} B_i \text{ OK} \qquad \tau = t_0 \times \ldots \times t_n \to t}{\Gamma \,.\, \hat{\gamma} \vdash_{\overline{DT}} \text{chain } B_1 + \ldots + B_p + \bullet, (\!|\_, \_, \_, \_, \tau, \hat{\gamma}'|\!)(\ e_0,\ldots,e_n\ ) : t}$$

T-BOD

$$\frac{\Gamma \vdash b \text{ OK} \qquad \Gamma, \text{this} : \Gamma(loc), typeBind(\Gamma, b_i, \langle t_0,\ldots,t_n\rangle) \,.\, depClose_{DT}(\hat{\gamma}') \vdash_{\overline{DT}} e : s \qquad s \preccurlyeq t \qquad depClose_{DT}(\hat{\gamma}') \subseteq depClose_{DT}(\hat{\gamma}) \qquad \tau = t_0 \times \ldots \times t_n \to t}{\Gamma \,.\, \hat{\gamma} \,.\, \tau \vdash_{\overline{DT}} [\![b, loc, e, \hat{\gamma}', \tau', \tau]\!] \text{ OK}}$$

T-BODS

$$\frac{\Gamma \vdash b \text{ OK} \qquad \Gamma, \text{this} : \Gamma(loc), typeBind(\Gamma, b, \langle t_0,\ldots,t_n\rangle) \,.\, depClose_{DT}(\{\text{self}_{loc}\}) \vdash_{\overline{DT}} e_b : s_b \qquad \Gamma, \text{reply} : \text{readonly } u_\top, \text{this} : \Gamma(loc), typeBind(\Gamma, b, \langle t_0,\ldots,t_n\rangle) \,.\, depClose_{DT}(\{\text{self}_{loc}\}) \vdash_{\overline{DT}} e_a : s_a \qquad t \preccurlyeq u_\top \qquad \tau = t_0 \times \ldots \times t_n \to t}{\Gamma \,.\, \hat{\gamma} \,.\, \tau \vdash_{\overline{DT}} [\![b, loc, (e_b, e_a), u_\top]\!]_S \text{ OK}}$$

T-LEAP

$$\frac{\Gamma \,.\, \hat{\gamma} \vdash_{\overline{DT}} e_0 : t \qquad \Gamma, \text{reply} : t \,.\, \hat{\gamma} \vdash_{\overline{DT}} e_1 : s}{\Gamma \,.\, \hat{\gamma} \vdash_{\overline{DT}} e_0 \curvearrowright e_1 : t}$$

Figure 5.6 Differences in the Static Semantics of MiniMAO$_3$ vs. MiniMAO$_2$

T-CHAIN and T-BOD together are equivalent to the T-CHAIN rule for MiniMAO$_2$ (see Figure 4.17 on page 147). Recall that the hypotheses of T-BOD serve to propagate type information needed in the subject reduction proof. The hypotheses of T-BODS serve a similar purpose for surround advice. The differences between T-BOD and T-BODS reflect the more relaxed typing requirements on surround advice.

The new T-LEAP rule is straightforward, mirroring the operational semantics for leap expressions. T-LEAP ensures that the left-hand expression, e_0, is well typed and gives its type, t, to the whole expression. The rule checks the right-hand expression, e_1, assuming that reply has type t. Since the right-hand expression is only evaluated for side effects, T-LEAP does not place constraints on its type. It is enough that the expression is well typed.

5.1.3.4 Static Semantics of Pointcuts for Surround Advice

The typing judgment for a pointcut in surround advice is denoted

$$\Gamma \vdash_S pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, u \,.\, V \,.\, V'.$$

See Section 3.2.3.2 on page 94 for a description of the various elements in the type.

The main differences between the pointcut typing rules for around advice (see Figure 4.20 on page 150) and those for surround advice (Figure 5.7 on the next page) are that the latter (1) do not track the set of writable concern domains matched by the pointcut and (2) allow more general combination of result types, treating an unconstrained result type as $\top$.

Difference (1) is possible because the set of writable concern domains of a piece of surround advice is always $\{\mathsf{self}\}$. I do not consider the side effects of the advised code to be side effects of the surround advice. They would happen even in the absence of the surround advice and, unlike for around advice, can only happen once. (The operational semantics reflects this treatment of side effects. The SURROUND rule tags the before- and after-part expressions with the set $\{\mathsf{self}_{loc}\}$. But the rule does not tag the chain expression representing the proceed of the advice.)

Difference (2) is possible because for surround advice the result type of the advised code does not need to be determined exactly, or even determined at all. Why is this? In around advice, proceeding to the advised code is done with a proceed expression. The result of the advised code is available to be manipulated by the advice, so the type system must place an upper bound on the possible result type. But, as discussed above, around advice may also return any value, not just the result of the advised code. To avoid problems in client code, which expect the result to conform to the type of the advised code, the type system must ensure that the value actually returned by the around advice is a subtype of the expected result type. Thus, for around advice the result type of the advised code must be determined exactly.

On the other hand, for surround advice, the result of the advised code is automatically the result of the advice. So the type system must only place an upper bound on the result type; this bound allows the result to be used in the after-part of the surround advice. If the bound is just $\top$, then the reply special variable reference might be used in the after-part, but essentially nothing can be done with it. A reply reference in such a piece of advice would have type $\top$. It could not be used as a target: $\top$ does not have fields or methods. It could not be passed as a parameter or assigned to a field: $\top$ is not in the user syntax, so parameters and fields cannot have type $\top$. It could not be returned as a result: the operational semantics for surround advice only evaluates surround advice for side effects.

This more relaxed typing for results raises another question. Why are the typing rules not relaxed for target, this, and args pointcut descriptors? In fact, they are relaxed somewhat by virtue of the pointcut typing

$$U ::= \langle t^* \rangle \mid \bot \qquad \hat{u} ::= t \mid \bot \qquad V \in \mathscr{P}(\mathcal{V})$$

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

T-CallPCD$_S$

$$\frac{\forall i \in \{1..q\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \vdash_S \text{call}(\ \delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle\ idPat(..)\ ) : \bot.\bot.\bot.\delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle.\emptyset.\emptyset}$$

T-ExecPCD$_S$

$$\frac{\forall i \in \{1..q\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \vdash_S \text{execution}(\ \delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle\ idPat(..)\ ) : \bot.\bot.\bot.\delta\ T\langle\gamma_1,\ldots,\gamma_q\rangle.\emptyset.\emptyset}$$

T-WrtPCD$_S$

$$\frac{\forall i \in \{1..n\} \cdot \Gamma(\gamma_i) = \text{domain}}{\Gamma \vdash_S \text{writes}(\ \gamma_1,\ldots,\gamma_n\ ) : \bot.\bot.\bot.\top.\emptyset.\emptyset}$$

T-ThisPCD$_S$

$$\frac{\Gamma(var) = t}{\Gamma \vdash_S \text{this}(\ t\ var\ ) : t.\bot.\bot.\top.\{var\}.\{var\}}$$

T-TargPCD$_S$

$$\frac{\Gamma(var) = t}{\Gamma \vdash_S \text{target}(\ t\ var\ ) : \bot.t.\bot.\top.\{var\}.\{var\}}$$

T-ArgsPCD$_S$

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

T-UnionPCD$_S$

$$\frac{\Gamma \vdash_S pcd_1 : \hat{u}.\hat{u}'.U.u_1.V_1.V_1' \qquad \Gamma \vdash_S pcd_2 : \hat{u}.\hat{u}'.U.u_2.V_2.V_2' \qquad u_1 \preccurlyeq u_\top \qquad u_2 \preccurlyeq u_\top \qquad V = V_1 \cap V_2 \qquad V' = V_1' \cup V_2'}{\Gamma \vdash_S pcd_1 \ ||\ pcd_2 : \hat{u}.\hat{u}'.U.u_\top.V.V'}$$

T-NegPCD$_S$

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

T-IntPCD$_S$

$$\frac{\Gamma \vdash_S pcd_1 : \hat{u}_1.\hat{u}_1'.U_1.u_1.V_1.V_1' \qquad \Gamma \vdash_S pcd_2 : \hat{u}_2.\hat{u}_2'.U_2.u_2.V_2.V_2' \qquad \hat{u} = \hat{u}_1 \sqcup \hat{u}_2 \qquad \hat{u}' = \hat{u}_1' \sqcup \hat{u}_2' \qquad U = U_1 \sqcup U_2 \qquad u_1 \preccurlyeq u_\top \qquad u_2 \preccurlyeq u_\top \qquad V_1' \cap V_2' = \emptyset \qquad V = V_1 \cup V_2 \qquad V' = V_1' \cup V_2'}{\Gamma \vdash_S pcd_1\ \&\&\ pcd_2 : \hat{u}.\hat{u}'.U.u_\top.V.V'}$$

Figure 5.7 Static Semantics of Pointcuts for Surround Advice

hypothesis in T-SURR, which allows these positions to be left unchecked (i.e., they can be $\perp$). Also, the surround advice pointcut matching function, $matchPCD_S$, uses subtype matching instead of exact matching. But still, in Figure 5.7 on the facing page, the T-UNIONPCD$_S$ and T-INTPCD$_S$ rules are strict about combining the type information for target, this, and args—just as strict as are the corresponding rules for around advice. The reason is that the type system must still ensure that formal parameters bound by these pointcut descriptors are bound exactly once. The result value is bound to reply automatically, so these binding issues do not come into play there. If MiniMAO$_3$ were extended with non-binding forms of target, this, and args like those available in AspectJ, then more relaxed typing could likely be used. I leave the formalization of AspectJ's full menagerie of pointcut descriptors to future work.

5.2 Meta-Theory of MiniMAO$_3$

Rather than restating the entire meta-theory from MiniMAO$_2$, I just give the definitions, lemmas, and theorems that differ for MiniMAO$_3$. The rest of the meta-theory of MiniMAO$_2$ is included here by reference, with the understanding that the descriptions apply to MiniMAO$_3$ syntax and semantics—a sort of dynamic scoping of the meta-theory. As in the previous chapter, I first discuss supporting definitions and lemmas, then I cover the soundness of the static type system. A final subsection addresses the effectiveness of effects clauses, private concern domains, and read-only annotations.

5.2.1 Supporting Definitions and Lemmas

Definition 4.2 (Concern-Complete Environments); Definition 4.3 (Environment-Store Consistency), $\Gamma \approx S$; and Definition 4.4 (Stack-Store Consistency), $J \approx S$, are applicable to MiniMAO$_3$ as written. In MiniMAO$_3$, a valid store must instantiate spectators in the appropriate, private self_{loc} concern domains. The definition of store validity reflects this.

Definition 5.1 (Store Validity). Given a well-typed program P with aspect instantiation instructions

$$\text{use } a_1\langle g_{1,1},\ldots,g_{1,p_1}\rangle;\ \ldots;\ \text{use } a_n\langle g_{n,1},\ldots,g_{n,p_n}\rangle,$$

we say that a store S is *valid* if both of the following hold:

1. $\forall i \in \{1..n\} \cdot \exists loc \in \mathscr{L} \cdot \begin{cases} S(loc) = [a_i\langle \mathsf{self}_{loc}, g_{i,2},\ldots,g_{i,p_i}\rangle \centerdot F] & \text{if } CT(a_i) = \text{spectator } \ldots \\ S(loc) = [a_i\langle g_{i,1},\ldots,g_{i,p_i}\rangle \centerdot F] & \text{otherwise} \end{cases}$
2. $\exists \Gamma \cdot \Gamma \approx S$

The statement of the Dependency Closure Inclusion lemma for MiniMAO$_3$ is the same as Lemma 4.6 (Dependency Closure Inclusion) on page 153. I update the proof to consider both public and private concern domains.

Lemma 5.2 (Dependency Closure Inclusion). *Let P be a program with concern domains $\hat{g}$ and evaluation dependency table DT. If $\hat{\gamma} \subseteq \hat{g}$, $\hat{\gamma}' \subseteq \hat{g}$, and $\hat{\gamma}' \subseteq depClose_{DT}(\hat{\gamma})$, then $depClose_{DT}(\hat{\gamma}') \subseteq depClose_{DT}(\hat{\gamma})$.*

Proof. Because DT is constant throughout the proof, I elide it where practical. Let γ' be an arbitrary element of $depClose(\hat{\gamma}')$. There are two possibilities.

If $\gamma' \in \mathscr{G}$, then by definition of *depClose*, there exists $\gamma \in \hat{\gamma}'$ such that $(\gamma, \gamma') \in DT$. But $\gamma \in \hat{\gamma}'$ implies $\gamma \in depClose(\hat{\gamma})$ by the assumption of the lemma. So again by the definition of *depClose*, there exists $\gamma'' \in \hat{\gamma}$

such that $(\gamma'', \gamma) \in DT$. Now DT is reflexive and transitive, so $(\gamma'', \gamma') \in DT$. By the definition of *depClose*, $\gamma'' \in \hat{\gamma} \implies \gamma' \in depClose(\hat{\gamma})$.

On the other hand, suppose $\gamma' \in \mathcal{G}_{\text{self}}$. By the definitions of the evaluation dependency closure and *depClose*, all private concern domains are in every dependency closure over DT. Thus, $\gamma' \in depClose(\hat{\gamma})$.

So every element of $depClose(\hat{\gamma}')$ is also an element of $depClose(\hat{\gamma})$. □

The Dependency Table Extension lemma for MiniMAO$_3$ has the same statement as Lemma 4.7 (Dependency Table Extension) on page 153, however the proof here considers the new definition of *depClose*.

Lemma 5.3 (Dependency Table Extension). *If e includes only user syntax, $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$, $DT \subseteq DT_2$, and $\forall \gamma \in \hat{\gamma} \cdot (\gamma, \gamma) \in DT_2$, then*

$$\Gamma \centerdot depClose_{DT_2}(\hat{\gamma}) \vdash_{DT_2} e : t.$$

Proof. The proof is by structural induction on the derivation of $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$. The base cases are T-New, T-Obj, T-Var, and T-Null. (We do not need to consider T-Loc, because locations are not part of the user syntax.) For all of these, the judgment does not depend on DT, so the claim holds.

The remaining expression typing rules constitute the induction steps. The induction hypothesis is that the claim of the lemma holds for all derivations smaller than the one under consideration. For T-Get, T-Set, T-Cast, T-Seq, T-Proc, and T-Under, the claim immediate from the induction hypothesis.

All but one of the hypotheses of T-Call hold immediately by the induction hypothesis. The one hypothesis from the derivation of $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$ that we must consider is $depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}$, where $\hat{\gamma}'$ is the set of writable domains from the effects clause of the called method. The corresponding hypothesis from the derivation of $\Gamma \centerdot depClose_{DT_2}(\hat{\gamma}) \vdash_{DT_2} e : t$ is $depClose_{DT_2}(\hat{\gamma}') \subseteq depClose_{DT_2}(\hat{\gamma})$.

First, note that $depClose_{DT}(\hat{\gamma}') \subseteq \hat{\gamma}$ implies $\hat{\gamma}' \subseteq \hat{\gamma}$. To see this, take $\gamma' \in \hat{\gamma}'$. By definition,

$$\gamma' \in depClose_{DT}(\hat{\gamma}')$$

and thus $\gamma' \in \hat{\gamma}$.

Next, note that $\hat{\gamma}' \subseteq \hat{\gamma}$ implies $depClose_{DT_2}(\hat{\gamma}') \subseteq depClose_{DT_2}(\hat{\gamma})$. To see this, take $\gamma' \in depClose_{DT_2}(\hat{\gamma}')$. There are two possibilities, depending on whether or not γ' is private, i.e., $\gamma' \in \mathcal{G}_{\text{self}}$.

If $\gamma' \in \mathcal{G}_{\text{self}}$, then let $\gamma' = \text{self}_{loc}$. By the definition of *depClose*,

$$(\text{self}_{loc}, \text{self}_{loc}) \in DT_2 \text{ and } \gamma' \in depClose_{DT_2}(\hat{\gamma}).$$

On the other hand, suppose $\gamma' \notin \mathcal{G}_{\text{self}}$. Then there exists $\gamma \in \hat{\gamma}'$ such that $(\gamma, \gamma') \in DT_2$. But $\hat{\gamma}' \subseteq \hat{\gamma}$ then implies that there exists $\gamma \in \hat{\gamma}$ such that $(\gamma, \gamma') \in DT_2$. So $\gamma' \in depClose_{DT_2}(\hat{\gamma})$.

Thus, by T-Call $\Gamma \centerdot depClose_{DT_2}(\hat{\gamma}) \vdash_{DT_2} e : t$, and the claim holds for this case.

The remaining expression typing rules—T-Exec, T-Chain, T-Join, T-Tag, and T-Leap—do not apply to user syntax. Thus, the claim holds. □

The Substitution lemma for MiniMAO$_3$ requires a proof case for the new leap expression.

Lemma 5.4 (Substitution). *If* $\Gamma, var_1 : t_1, \ldots, var_n : t_n \,.\, \hat{\gamma} \vdash_{DT} e : t$ *and* $\forall i \in \{1..n\} \cdot \Gamma \,.\, \hat{\gamma} \vdash_{DT} e_i : s_i$ *where* $s_i \preccurlyeq t_i$ *then* $\Gamma \,.\, \hat{\gamma} \vdash_{DT} e\{\!|e_1 / var_1, \ldots, e_n / var_n|\!\} : s$ *for some* $s \preccurlyeq t$.

Proof. The set up for the proof is exactly like that for Lemma 4.8 (Substitution) on page 154. I just give here the induction step for the new T-LEAP case. (The differences in T-CHAIN between MiniMAO$_2$ and MiniMAO$_3$ are immaterial for this lemma.)

For the T-LEAP case, $e = e'_1 \curvearrowright e'_2$ and the last step in the type derivation is:

$$\frac{\Gamma' \,.\, \hat{\gamma} \vdash e'_1 : t \qquad \Gamma', \mathsf{reply} : t \,.\, \hat{\gamma} \vdash e'_2 : s}{\Gamma' \,.\, \hat{\gamma} \vdash e'_1 \curvearrowright e'_2 : t}$$

Now $e\{\!|\bar{e} / \overline{var}|\!\} = e'_1\{\!|\bar{e} / \overline{var}|\!\} \curvearrowright e'_2\{\!|\bar{e} / \overline{var}|\!\}$. By the induction hypothesis, $\Gamma \,.\, \hat{\gamma} \vdash e'_1\{\!|\bar{e} / \overline{var}|\!\} : t'$, $\Gamma, \mathsf{reply} : t \,.\, \hat{\gamma} \vdash e'_2\{\!|\bar{e} / \overline{var}|\!\} : s'$, for some $t' \preccurlyeq t$ and $s' \preccurlyeq s$ (where the application of the induction hypothesis for e'_2 uses the initial type environment $\Gamma, \mathsf{reply} : t$). Lemma 4.13 (Environment Subtyping) on page 162 gives $\Gamma, \mathsf{reply} : t' \,.\, \hat{\gamma} \vdash e'_2\{\!|\bar{e} / \overline{var}|\!\} : s''$, for some $s'' \preccurlyeq s'$. Thus, by T-LEAP, $\Gamma \,.\, \hat{\gamma} \vdash e\{\!|\bar{e} / \overline{var}|\!\} : t'$, $t' \preccurlyeq t$, and the claim holds. □

Lemma 4.9 (Environment Extension), Lemma 4.10 (Environment Contraction), and Lemma 4.11 (Replacement) from MiniMAO$_2$ apply to MiniMAO$_3$ as written.

The Replacement with Subtyping lemma for MiniMAO$_3$ introduces a proof case for the new leap evaluation context. The case for the chain evaluation context of the proof from MiniMAO$_2$ is sufficiently general as to apply here without change.

Lemma 5.5 (Replacement with Subtyping). *If* $\Gamma \,.\, \hat{\gamma} \vdash_{DT} \mathbb{E}[e] : t$, $\Gamma \,.\, \hat{\gamma}' \vdash_{DT} e : u$, *and* $\Gamma \,.\, \hat{\gamma}' \vdash_{DT} e' : u'$ *where* $u' \preccurlyeq u$ *and* $\hat{\gamma}' \subseteq \hat{\gamma}$, *then* $\Gamma \,.\, \hat{\gamma} \vdash_{DT} \mathbb{E}[e'] : t'$ *where* $t' \preccurlyeq t$.

Proof. The set up for the proof is exactly like that for Lemma 4.12 (Replacement with Subtyping) on page 159. I just give here the case for the new leap evaluation context. (The differences due to the new form of T-CHAIN for MiniMAO$_3$ are immaterial for this lemma.)

Suppose $\mathbb{E}_2 = - \curvearrowright e''$. The last step in the type derivation for $\mathbb{E}_2[e]$ must be T-LEAP (with $\hat{\gamma}' = \hat{\gamma}''$ and $s = u$):

$$\frac{\Gamma \,.\, \hat{\gamma}'' \vdash e : s \qquad \Gamma, \mathsf{reply} : s \,.\, \hat{\gamma}'' \vdash e'' : s''}{\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e] : s}$$

By assumption, $\Gamma \,.\, \hat{\gamma}'' \vdash e' : u'$, where $u' \preccurlyeq u = s$. By Lemma 4.13 (Environment Subtyping) on page 162, $\Gamma, \mathsf{reply} : u' \,.\, \hat{\gamma}'' \vdash e'' : u''$ for some $u'' \preccurlyeq s''$.

Thus, $\Gamma \,.\, \hat{\gamma}'' \vdash \mathbb{E}_2[e'] : u'$ where $u' \preccurlyeq s$. □

Lemma 4.13 (Environment Subtyping) from MiniMAO$_2$ applies to MiniMAO$_3$ as written. Lemma 4.14 (Binding Soundness) from MiniMAO$_2$, for around-advice body tuples, also applies to MiniMAO$_3$. I add a new binding soundness lemma to deal with surround-advice body tuples.

Lemma 5.6 (Surround Binding Soundness). *Let P be a well-typed program with evaluation dependency table DT. Let S be a valid store for P and* $J = (\!|\ldots, (t_0 \times \ldots \times t_n \to t), \hat{\gamma}|\!) + J'$ *be a stack consistent with S. If* $\bar{B} = adviceBind(J, S)$, *then* $\forall [\![b, loc, (e_b, e_a), u_\top]\!]_S \in \bar{B}$ *the following conditions hold:*

Consequent 1. $t \preccurlyeq u_\top$

Consequent 2. $\emptyset \vdash b$ *OK*

Consequent 3. For concern-complete $\Gamma \approx S$*, the judgments*

$$\Gamma, \mathit{this} : \Gamma(\mathit{loc}), \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \,.\, \mathit{depClose}_{DT}\left(\{\mathit{self}_{\mathit{loc}}\}\right) \vdash_{\overline{DT}} e_b : t_b$$

and

$$\Gamma, \mathit{reply} : \mathit{readonly}\ u_\top, \mathit{this} : \Gamma(\mathit{loc}), \mathit{typeBind}(\Gamma, b, \langle t_0, \ldots, t_n \rangle) \,.\, \mathit{depClose}_{DT}\left(\{\mathit{self}_{\mathit{loc}}\}\right) \vdash_{\overline{DT}} e_a : t_a$$

hold for some types t_b *and* t_a*.*

Proof. I will use a common setup and some common meta-variables throughout the proof.

Pick an arbitrary surround-advice element $[\![b, \mathit{loc}, (e_\mathrm{b}, e_\mathrm{a}), u_\top]\!]_S \in \bar{B}$. Let the surround advice corresponding to $[\![b, \mathit{loc}, (e_\mathrm{b}, e_\mathrm{a}), u_\top]\!]_S$ be

$$\text{surround}\ (\ s''_1\ \mathit{var}_1, \ldots, s''_p\ \mathit{var}_p\ )\ :\ \mathit{pcd}''\ \{\ e''_\mathrm{b};\ \text{proceed};\ e''_\mathrm{a}\ \}$$

with advice table entry $\langle b, \mathit{loc}, (e_\mathrm{b}, e_\mathrm{a}), u_\top \rangle$. Let this advice be declared in a spectator a with concern domain variables $\mathsf{self}, G_2, \ldots, G_{q'}$. Let $S(\mathit{loc}) = \left[a\langle \mathsf{self}_{\mathit{loc}}, g_2, \ldots, g_{q'} \rangle \,.\, F \right]$. We will consider the typing derivation for this advice, which must exist because the program is well typed. However, we will α-convert the entire derivation, replacing self with $\mathsf{self}_{\mathit{loc}}$ and G_i with g_i for all $i \in \{2..q'\}$.

To simplify the notation, I will write $\{\!| \bar{g} / \bar{G} |\!\}$ for $\{\!| \mathsf{self}_{\mathit{loc}} / \mathsf{self}, g_2 / G_2, \ldots, g_{q'} / G_{q'} |\!\}$. Let $\forall i \in \{1..p\} \cdot s_i = s''_i \{\!| \bar{g} / \bar{G} |\!\}$, and $\Gamma' = \mathit{var}_1 : s_1, \ldots, \mathit{var}_p : s_p, \mathsf{self}_{\mathit{loc}} : \mathsf{domain}, g_2 : \mathsf{domain}, \ldots, g_{q'} : \mathsf{domain}$. By the construction of AT, $e_\mathrm{b} = e''_\mathrm{b} \{\!| \bar{g} / \bar{G} |\!\}$, $e_\mathrm{a} = e''_\mathrm{a} \{\!| \bar{g} / \bar{G} |\!\}$, and $\mathit{pcd} = \mathit{pcd}'' \{\!| \bar{g} / \bar{G} |\!\}$. Let the dependency table of the advice typing be $DT_a = \mathit{depTable}\left(\left\{\mathsf{self}_{\mathit{loc}}, g_2, \ldots, g_{q'}\right\}, \emptyset\right)$. This comes from T-SPEC, with concern domain variables replaced by concern domain names.

Plugging this notation into the α-converted derivation from T-SURR gives:

$$\frac{\begin{array}{c} \Gamma' \vDash_S \mathit{pcd} : \_ \,.\, \_ \,.\, \_ \,.\, u_\top \,.\, V \,.\, V \qquad V = \{\mathit{var}_1, \ldots, \mathit{var}_p\} \\ \Gamma', \mathsf{this} : a\langle \mathsf{self}_{\mathit{loc}}, g_2, \ldots, g_{q'} \rangle \,.\, \{\mathsf{self}_{\mathit{loc}}\} \vdash_{\overline{DT}_a} e_\mathrm{b} : s_\mathrm{b} \\ \Gamma', \mathsf{this} : a\langle \mathsf{self}_{\mathit{loc}}, g_2, \ldots, g_{q'} \rangle, \mathsf{reply} : \mathsf{readonly}\ u_\top \,.\, \{\mathsf{self}_{\mathit{loc}}\} \vdash_{\overline{DT}_a} e_\mathrm{a} : s_\mathrm{a} \\ \forall i \in \{1..p\} \cdot \emptyset \vdash s_i\ \text{OK in}\ a\langle \mathsf{self}_{\mathit{loc}}, g_2, \ldots, g_{q'} \rangle \end{array}}{DT_a \vdash \text{surround}(\ s_1\ \mathit{var}_1, \ldots, s_p\ \mathit{var}_p\ )\ :\ \mathit{pcd}\ \{\ e_\mathrm{b};\ \text{proceed};\ e_\mathrm{a}\ \}\ \text{OK in}\ a\langle \mathsf{self}_{\mathit{loc}}, g_2, \ldots, g_{q'} \rangle} \tag{5.1}$$

For convenience, Figure 5.8 on the next page summarizes the setup of the proof and the use of these meta-variables.

Consequent 1 on the preceding page relates the expected return type of the matched code, from *pcd*, to the actual return type from the join point abstraction. This ensures that if reply is used in e_a, then it is treated as having the correct type. The following subclaim says that the consequent holds if *pcd* is well-typed, which it must be in a well-typed program.

Subclaim 1. Assume $\Gamma' \vDash_S \mathit{pcd} : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, u_\top \,.\, V' \,.\, V''$. Then

$$\mathit{matchPCD}_S(J, \mathit{pcd}, S) \neq \bot \implies t \preccurlyeq u_\top$$

Meta-variable Bindings:

$$[\![b, loc, (e_b, e_a), u_\top]\!]_S \in \bar{B}$$
$$S(loc) = \left[a\langle \text{self}_{loc}, g_2, \ldots, g_{q'}\rangle \,.\, F\right]$$
$$\Gamma' = var_1 : s_1, \ldots, var_p : s_p, \text{self}_{loc} : \text{domain}, g_2 : \text{domain}, \ldots, g_{q'} : \text{domain}$$

Advice Type Derivation (with domains reified):

$$\frac{\begin{array}{c}\Gamma' \vdash_S pcd : \_ \,.\, \_ \,.\, \_ \,.\, u_\top \,.\, V \,.\, V \qquad V = \{var_1, \ldots, var_p\} \\ \Gamma', \text{this} : a\langle \text{self}_{loc}, g_2, \ldots, g_{q'}\rangle \,.\, \{\text{self}_{loc}\} \vdash_{DT_a} e_b : s_b \\ \Gamma', \text{this} : a\langle \text{self}_{loc}, g_2, \ldots, g_{q'}\rangle, \text{reply} : \text{readonly } u_\top \,.\, \{\text{self}_{loc}\} \vdash_{DT_a} e_a : s_a \\ \forall i \in \{1..p\} \cdot \emptyset \vdash s_i \text{ OK in } a\langle \text{self}_{loc}, g_2, \ldots, g_{q'}\rangle\end{array}}{DT_a \vdash \text{surround}(\, s_1\ var_1, \ldots, s_p\ var_p\,) : pcd\ \{\ e_b;\ \text{proceed};\ e_a\ \}\ \text{OK in } a\langle \text{self}_{loc}, g_2, \ldots, g_{q'}\rangle}$$

Figure 5.8 Setup and Common Meta-variable Bindings Used in the Proof of Lemma 5.6

Proof of subclaim.

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

$$\begin{aligned} matchPCD_S(J, pcd, S) \neq \bot &\implies t \preccurlyeq t'' \\ &\implies t \preccurlyeq u_\top. \end{aligned}$$

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

— $pcd = \text{writes}(\ldots)$. By T-WRTPCD$_S$, $u_\top = \top$. By definition of subtyping, $t \preccurlyeq \top$.

— $pcd = \text{this}(\ldots)$. Here $t \preccurlyeq \top = u_\top$, by T-THISPCD$_S$.

— $pcd = \text{target}(\ldots)$. Here $t \preccurlyeq \top = u_\top$, by T-TARGPCD$_S$.

— $pcd = \text{args}(\ldots)$. Here $t \preccurlyeq \top = u_\top$, by T-ARGSPCD$_S$.

— $pcd = pcd_1 \ ||\ pcd_2$. By T-UNIONPCD$_S$, $\Gamma' \vdash_S pcd_1 : \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, u_1 \,.\, V_1 \,.\, V'_1$, $\Gamma' \vdash_S pcd_2 : \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, u_2 \,.\, V_2 \,.\, V'_2$, $u_1 \preccurlyeq u_\top$, and $u_2 \preccurlyeq u_\top$. By the induction hypothesis, $matchPCD_S(J, pcd_1, S) \neq \bot \implies t \preccurlyeq u_1 \preccurlyeq u_\top$ and $matchPCD_S(J, pcd_2, S) \neq \bot \implies t \preccurlyeq u_2 \preccurlyeq u_\top$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD_S(J, pcd, S) \neq \bot &\implies matchPCD_S(J, pcd_1, S) \neq \bot \text{ or } matchPCD_S(J, pcd_2, S) \neq \bot \\ &\implies t \preccurlyeq u_\top \end{aligned}$$

— $pcd = pcd_1 \ \&\&\ pcd_2$. By T-INTPCD$_S$, $\Gamma' \vdash_S pcd_1 : \hat{u}_1 \,.\, \hat{u}'_1 \,.\, U_1 \,.\, u_1 \,.\, V_1 \,.\, V'_1$, $\Gamma' \vdash_S pcd_2 : \hat{u}_2 \,.\, \hat{u}'_2 \,.\, U_2 \,.\, u_2 \,.\, V_2 \,.\, V'_2$, $u_1 \preccurlyeq u_\top$, and $u_2 \preccurlyeq u_\top$. By the induction hypothesis, $matchPCD_S(J, pcd_1, S) \neq \bot \implies t \preccurlyeq u_1 \preccurlyeq u_\top$ and $matchPCD_S(J, pcd_2, S) \neq \bot \implies t \preccurlyeq u_2 \preccurlyeq u_\top$. By the definition of *matchPCD*,

$$\begin{aligned} matchPCD_S(J, pcd, S) \neq \bot &\implies matchPCD_S(J, pcd_1, S) \neq \bot \text{ and } matchPCD_S(J, pcd_2, S) \neq \bot \\ &\implies t \preccurlyeq u_\top \end{aligned}$$

— $pcd = !\ pcd_1$. Here $t \preccurlyeq \top = u_\top$, by T-NEGPCD$_S$.

Subclaim-□

We next turn to consequent 2 on page 216. 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 4.15 on page 144).

Subclaim 2. Assume $\Gamma' \vDash_S pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, u_\top \,.\, V' \,.\, V''$. Then $matchPCD_S(J, pcd, S) = b = \langle \alpha, \beta_0, \ldots, \beta_x \rangle$ implies all of the following:

$$\emptyset \vdash b \text{ OK} \tag{5.2a}$$

$$V' \subseteq var(b) \subseteq V'' \tag{5.2b}$$

$$\hat{u} = \bot \iff \alpha = - \tag{5.2c}$$

$$\hat{u}' = \bot \iff \beta_0 = - \tag{5.2d}$$

$$U = \bot \implies x = 0 \tag{5.2e}$$

$$U \neq \bot \implies x = n \tag{5.2f}$$

$$U = \bot \iff \forall i \in \{1..x\} \cdot \beta_i = - \tag{5.2g}$$

Proof of subclaim. The proof of the subclaim exactly follows that of Subclaim 5 of Lemma 4.14 (Binding Soundness) for around advice (see page 168), but using surround advice pointcut typing rules and $matchPCD_S$. Neither the *pcd* result type nor the writable domains set from the typing judgment for around advice are material to the proof.

Subclaim-□

By T-SURR, the assumption of the subclaim holds. Therefore, consequent 2 on page 216 holds by (5.2a).

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

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle) = var_1 : s'_1, \ldots, var_p : s'_p \text{ where } \forall i \in \{1..p\} \cdot s'_i \preccurlyeq s_i \tag{5.3}$$

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 to this end.

Subclaim 3. Assume $\Gamma' \vDash_S pcd : \hat{u} \,.\, \hat{u}' \,.\, U \,.\, u_\top \,.\, V' \,.\, V''$, where $V'' \subseteq \{var_1, \ldots, var_p\}$. Then

$$\begin{aligned} &matchPCD_S(J, pcd, S) = b \neq \bot \\ &\implies \forall var \in var(b) \cdot \big(\exists i \in \{1..p\}, s'_i \in \mathcal{T} \cdot \big(var = var_i,\ typeBind(\Gamma, b, \langle t_0, \ldots, t_n \rangle)(var_i) = s'_i, \text{ and } s'_i \preccurlyeq s_i\big)\big) \end{aligned}$$

Proof of subclaim. As with the previous one, the proof of this subclaim closely follows that from the previous chapter for around advice (see Subclaim 6 on page 172). I choose to omit it.

Subclaim-□

With this last subclaim in hand we can now prove the final consequent of the lemma. The first two hypotheses of T-SURR (see (5.1) on page 216) are:

$$\Gamma' \vdash_{\mathrm{S}} pcd : \_ \centerdot \_ \centerdot \_ \centerdot u_\top \centerdot V \centerdot V$$
$$V = \{var_1, \ldots, var_p\}$$

By definition of *adviceBind*, $[\![b, loc, (e_\mathrm{b}, e_\mathrm{a}), u_\top]\!]_\mathrm{S} \in \bar{B}$ implies $matchPCD_\mathrm{S}(J, pcd, S) \neq \bot$. We first use Subclaim 2 and Subclaim 3 to prove equation (5.3) from page 218.

$$\begin{array}{lll} V = \{var_1, \ldots, var_p\} & & \text{by T-ADV} \\ \quad \Longrightarrow\ var(b) = \{var_1, \ldots, var_p\} & & \text{by (5.2b)} \\ \quad \Longrightarrow\ \forall i \in \{1..p\} \cdot \exists s'_i \in \mathscr{T} \cdot & & \\ \qquad \big(typeBind(\Gamma, b, \langle t_0, \ldots, t_n\rangle)(var_i) = s'_i,\ s'_i \preccurlyeq s_i\big) & & \text{by Subclaim 3} \end{array}$$

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

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

Thus, no additional variables are bound and (5.3) on the facing page holds:

$$typeBind(\Gamma, b, \langle t_0, \ldots, t_n\rangle) = var_1 : s'_1, \ldots, var_p : s'_p \text{ where } \forall i \in \{1..p\} \cdot s'_i \preccurlyeq s_i$$

Let $\Gamma'' = \mathsf{self}_{loc} : \mathsf{domain}, g_2 : \mathsf{domain}, \ldots, g_{q'} : \mathsf{domain}$. Then T-SURR gives:

$$\begin{array}{l} var_1 : s_1, \ldots, var_p : s_p, \mathsf{this} : a\langle \mathsf{self}_{loc}, g_2, \ldots, g_{q'}\rangle, \Gamma'' \centerdot \{\mathsf{self}_{loc}\} \vdash_{DT_a} e_\mathrm{b} : s_\mathrm{b} \\ \quad \Longrightarrow \qquad \text{by Lemma 4.13} \\ \quad var_1 : s'_1, \ldots, var_p : s'_p, \mathsf{this} : a\langle \mathsf{self}_{loc}, g_2, \ldots, g_{q'}\rangle, \Gamma'' \centerdot \{\mathsf{self}_{loc}\} \vdash_{DT_a} e_\mathrm{b} : t_\mathrm{b} \\ \qquad \text{where } t_\mathrm{b} \preccurlyeq s_\mathrm{b} \text{ and } \forall i \in \{1..p\} \cdot s'_i \preccurlyeq s_i \\ \quad \Longrightarrow \qquad \text{by (5.3)} \\ \quad \mathsf{this} : a\langle \mathsf{self}_{loc}, g_2, \ldots, g_{q'}\rangle, typeBind(\Gamma, b, \langle t_0, \ldots, t_n\rangle), \Gamma'' \centerdot \{\mathsf{self}_{loc}\} \vdash_{DT_a} e_\mathrm{b} : t_\mathrm{b} \\ \quad \Longrightarrow \qquad \text{by Lemma 4.9, with appropriate } \alpha\text{-conversion of } b \text{ and } e \\ \quad \Gamma, \mathsf{this} : a\langle \mathsf{self}_{loc}, g_2, \ldots, g_{q'}\rangle, typeBind(\Gamma, b, \langle t_0, \ldots, t_n\rangle), \Gamma'' \centerdot \{\mathsf{self}_{loc}\} \vdash_{DT_a} e_\mathrm{b} : t_\mathrm{b} \\ \quad \Longrightarrow \qquad \text{by concern-completeness of } \Gamma \\ \quad \Gamma, \mathsf{this} : a\langle \mathsf{self}_{loc}, g_2, \ldots, g_{q'}\rangle, typeBind(\Gamma, b, \langle t_0, \ldots, t_n\rangle) \centerdot \{\mathsf{self}_{loc}\} \vdash_{DT_a} e_\mathrm{b} : t_\mathrm{b} \end{array}$$

By the definition of evaluation dependency table, $DT_a \subseteq DT$ and $(\mathsf{self}_{loc}, \mathsf{self}_{loc}) \in DT$. The expression e_b contains only user syntax, by the construction of *AT*. Thus Lemma 5.3 (Dependency Table Extension) on page 214 gives:

$$\Gamma, \mathsf{this} : a\langle \mathsf{self}_{loc}, g_2, \ldots, g_{q'}\rangle, \mathsf{proceed} : \tau', typeBind(\Gamma, b, \langle t_0, \ldots, t_n\rangle) \centerdot depClose_{DT}\big(\{\mathsf{self}_{loc}\}\big) \vdash_{DT} e_\mathrm{b} : t_\mathrm{b}$$

So the first part of consequent 3 holds. The second part holds similarly. □

I restate the Advice Chaining lemma based on the T-CHAIN rule of MiniMAO$_3$.

Lemma 5.7 (Advice Chaining). *Let*

$$\begin{gathered}(\Gamma, \mathit{proceed}:\tau)\centerdot\hat{\gamma}\vdash_{DT} e:t,\\ j=\langle\!|\_,\_,\_,\_,\tau,\hat{\gamma}''|\!\rangle,\\ \tau=t_0\times\ldots\times t_n\to t,\\ \mathit{depClose}_{DT}(\hat{\gamma}'')\subseteq\hat{\gamma},\\ (\mathit{readonly}(t_0)=\mathit{readonly})\implies(\hat{\gamma}''=\emptyset),\end{gathered}$$

and for all $B\in\bar{B}$ *let* $(\Gamma, \mathit{proceed}:\tau)\centerdot\hat{\gamma}''\centerdot\tau\vdash_{DT} B$ *OK.*

Then $\Gamma\centerdot\hat{\gamma}\vdash_{DT}\langle\!\langle e\rangle\!\rangle_{\bar{B},j}:t$.

Proof. The statement of the lemma is made more concise by the T-BOD and T-BODS rules introduced for MiniMAO$_3$. However, the proof of the lemma is exactly like that for Lemma 4.15 (Advice Chaining) on page 175, so I omit it here. □

Lemma 4.16 (Join Point Abstractions) for MiniMAO$_2$ applies as written to MiniMAO$_3$.

5.2.2 Type Safety

I restate the Subject Reduction theorem to account for MiniMAO$_3$'s private concern domains. The set of writable concern domains used in the expression typing judgments in the theorem must include all private concern domains, because they may always be mutated. However, most of the proof of Theorem 4.17 (Subject Reduction) on page 177 can be incorporated here without change.

Theorem 5.8 (Subject Reduction). *Given a well-typed program P with public concern domains* $\hat{g}$ *and private concern domains* $\hat{g}'$*, for an expression e, a valid store S, a stack J consistent with S, a concern-complete type environment* Γ *consistent with S, a set of concern domains* $\hat{\gamma}$ *with* $\hat{g}'\subseteq\hat{\gamma}\subseteq(\hat{g}\cup\hat{g}')$*, and the evaluation dependency table, DT, of P, if* $\Gamma\centerdot\hat{\gamma}\vdash_{DT} e:t$ *and* $\langle e,J,S\rangle\hookrightarrow\langle e',J',S'\rangle$*, then* $J'\approx S'$*,* S' *is valid, and there exist concern-complete* $\Gamma'\approx S'$ *and* $t'\preccurlyeq t$*, such that* $\Gamma'\centerdot\hat{\gamma}\vdash_{DT} e':t'$.

Proof. To update the proof of Theorem 4.17 (Subject Reduction), I add two new cases for SURROUND and LEAP, update the BIND case to account for the revised T-CHAIN type rule, and explain why the ADVISE case does not need to be updated. All other cases from the proof on page 177 apply in MiniMAO$_3$, as does the set up for the proof.

Case 1—LEAP. Here $e=\mathbb{E}[v\curvearrowright e'']$, $e'=\mathbb{E}[e''\{\!|v/\,\mathrm{reply}|\!\};\ v]$, $J'=J$, and $S'=S$.

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

It remains to be shown that $\Gamma\centerdot\hat{\gamma}\vdash e':t$. Because e is well typed, it must be that $v\curvearrowright e''$ is also. Let $\Gamma\centerdot\hat{\gamma}\vdash v\curvearrowright e'':s$. This must be by T-LEAP with hypotheses $\Gamma\centerdot\hat{\gamma}\vdash v:s$ and $\Gamma,\mathrm{reply}:s\centerdot\hat{\gamma}\vdash e'':s'$ for some type s'.

From the second hypothesis and Lemma 5.4 (Substitution) on page 215 $\Gamma\centerdot\hat{\gamma}\vdash e''\{\!|v/\,\mathrm{reply}|\!\}:s''$ for some $s''\preccurlyeq s'$. By T-SKIP $\Gamma\centerdot\hat{\gamma}\vdash e''\{\!|v/\,\mathrm{reply}|\!\};\ v:s$. Thus, Lemma 4.11 (Replacement) on page 158 gives $\Gamma\centerdot\hat{\gamma}\vdash e':t$.

Case 2—SURROUND. Here

$$
\begin{aligned}
e &= \mathbb{E}[\mathsf{chain}\ [\![b, loc, (e_\mathrm{b}, e_\mathrm{a}), u_\top]\!]_\mathrm{S} + \bar{B}, j(\ v_0, \ldots, v_n\ )] \\
e' &= \mathbb{E}[\mathsf{under}\ \left(\left(\langle e'_\mathrm{b}\rangle_{\varepsilon,\{\mathsf{self}_{loc}\}};\ \mathsf{chain}\ \bar{B}, j(\ v_0, \ldots, v_n\ )\right) \curvearrowright \langle e'_\mathrm{a}\rangle_{\varepsilon,\{\mathsf{self}_{loc}\}}\right)] \\
j &= (\!|\_, \_, \_, \_, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}_\mathrm{m}|\!) \\
e'_\mathrm{b} &= e_\mathrm{b}\{\!|loc/\mathsf{this}|\!\}\{\!|(v_0, \ldots, v_n)/b|\!\} \\
e'_\mathrm{a} &= e_\mathrm{a}\{\!|loc/\mathsf{this}|\!\}\{\!|(v_0, \ldots, v_n)/b|\!\} \\
J' &= (\!|\mathsf{this}, loc, -, -, -, -|\!) + J \\
S' &= S
\end{aligned}
$$

Let $\Gamma' = \Gamma$. Clearly $\Gamma' \approx S'$. Because $[\![-]\!]_\mathrm{S}$ 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 $\Gamma \centerdot \hat{\gamma} \vdash e' : t$. Let

$$
\begin{aligned}
e_\mathsf{left} &= \mathsf{chain}\ [\![b, loc, (e_\mathrm{b}, e_\mathrm{a}), u_\top]\!]_\mathrm{S} + \bar{B}, j(\ v_0, \ldots, v_n\ ) \\
e_\mathsf{right} &= \mathsf{under}\ \left(\left(\langle e'_\mathrm{b}\rangle_{\varepsilon,\{\mathsf{self}_{loc}\}};\ \mathsf{chain}\ \bar{B}, j(\ v_0, \ldots, v_n\ )\right) \curvearrowright \langle e'_\mathrm{a}\rangle_{\varepsilon,\{\mathsf{self}_{loc}\}}\right)
\end{aligned}
$$

Because e is well typed, we know that e_left and all its subterms are also. Let $\Gamma \centerdot \hat{\gamma} \vdash e_\mathsf{left} : s$. This must be by T-CHAIN as follows:

$$
\dfrac{\dfrac{\begin{array}{c}\Gamma \vdash b\ \mathrm{OK} \qquad s \preccurlyeq u_\top \qquad \Gamma, \mathsf{this} : \Gamma(loc), typeBind(\Gamma, b, \langle s_0, \ldots, s_n\rangle) \centerdot depClose(\{\mathsf{self}_{loc}\}) \vdash e_\mathrm{b} : s_\mathrm{b} \\ \Gamma, \mathsf{reply} : \mathsf{readonly}\ u_\top, \mathsf{this} : \Gamma(loc), typeBind(\Gamma, b, \langle s_0, \ldots, s_n\rangle) \centerdot depClose(\{\mathsf{self}_{loc}\}) \vdash e_\mathrm{a} : s_\mathrm{a}\end{array}}{\Gamma \centerdot \hat{\gamma}_\mathrm{m} \centerdot (s_0 \times \ldots \times s_n \to s) \vdash [\![b, loc, (e_\mathrm{b}, e_\mathrm{a}), u_\top]\!]_\mathrm{S}\ \mathrm{OK}}\ \text{T-BODS} \qquad \begin{array}{c}\forall i \in \{0..n\} \cdot \Gamma \centerdot \hat{\gamma} \vdash v_i : u_i \qquad \forall i \in \{0..n\} \cdot u_i \preccurlyeq s_i \qquad depClose(\hat{\gamma}_\mathrm{m}) \subseteq \hat{\gamma} \\ (readonly(u_0) = \mathsf{readonly}) \implies (\hat{\gamma}_\mathrm{m} = \emptyset) \qquad \forall B \in \bar{B} \cdot \Gamma \centerdot \hat{\gamma}_\mathrm{m} \centerdot (s_0 \times \ldots \times s_n \to s) \vdash B\ \mathrm{OK}\end{array}}{\Gamma \centerdot \hat{\gamma} \vdash e_\mathsf{left} : s}\ \text{T-CHAIN}
\tag{5.4}
$$

In the derivation above I expanded the subderivation by T-BODS, because we will need some of its hypotheses to complete the case.

We want to show that $\Gamma \centerdot \hat{\gamma} \vdash e_\mathsf{right} : s$. This must be by a derivation like that shown in Figure 5.9 on page 222. There are five "leaf" hypotheses in this derivation (appearing in the top fringe of the proof tree), one of which appears twice. We must show that each of the four unique leaf hypotheses hold.

Two of the leaf hypotheses are easy to demonstrate. $\Gamma \centerdot \hat{\gamma} \vdash \mathsf{chain}\ \bar{B}, j(\ v_0, \ldots, v_n\ ) : s$ holds by T-CHAIN using all the hypotheses from derivation (5.4) except the judgment of T-BODS. By definition, $depClose(\{\mathsf{self}_{loc}\}) = \hat{g}'$, the set of all private concern domains. From the statement of the theorem, $\hat{g}' \subseteq \hat{\gamma}$, so $depClose(\{\mathsf{self}_{loc}\}) \subseteq \hat{\gamma}$. It remains to show the truth of the two leaf hypotheses for e'_b and e'_a.

From (5.4), we know

$$
\Gamma, \mathsf{reply} : \mathsf{readonly}\ u_\top, \mathsf{this} : \Gamma(loc), typeBind(\Gamma, b, \langle s_0, \ldots, s_n\rangle) \centerdot depClose(\{\mathsf{self}_{loc}\}) \vdash e_\mathrm{a} : s_\mathrm{a}. \tag{5.5}
$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \centerdot \mathit{depClose}(\{\mathsf{self}_{\mathit{loc}}\}) \vdash e'_{\mathrm{b}} : s'_{\mathrm{b}} \quad \mathit{depClose}(\{\mathsf{self}_{\mathit{loc}}\}) \subseteq \hat{\gamma}}{\Gamma \centerdot \hat{\gamma} \vdash \langle e'_{\mathrm{b}} \rangle_{\varepsilon,\{\mathsf{self}_{\mathit{loc}}\}} : s'_{\mathrm{b}}}\ \text{T-Tag} \quad \Gamma \centerdot \hat{\gamma} \vdash \mathsf{chain}\ \bar{B}, j(\, v_0, \ldots, v_n\,) : s}{\Gamma \centerdot \hat{\gamma} \vdash \left(\langle e'_{\mathrm{b}} \rangle_{\varepsilon,\{\mathsf{self}_{\mathit{loc}}\}}; \mathsf{chain}\ \bar{B}, j(\, v_0, \ldots, v_n\,)\right) : s}\ \text{T-Skip} \quad \dfrac{\dfrac{\Gamma, \mathsf{reply} : s \centerdot \mathit{depClose}(\{\mathsf{self}_{\mathit{loc}}\}) \vdash e'_{\mathrm{a}} : s'_{\mathrm{a}} \quad \mathit{depClose}(\{\mathsf{self}_{\mathit{loc}}\}) \subseteq \hat{\gamma}}{\Gamma, \mathsf{reply} : s \centerdot \hat{\gamma} \vdash \langle e'_{\mathrm{a}} \rangle_{\varepsilon,\{\mathsf{self}_{\mathit{loc}}\}} : s'_{\mathrm{a}}}\ \text{T-Tag}}{\Gamma \centerdot \hat{\gamma} \vdash \left(\langle e'_{\mathrm{b}} \rangle_{\varepsilon,\{\mathsf{self}_{\mathit{loc}}\}} \curvearrowright \langle e'_{\mathrm{a}} \rangle_{\varepsilon,\{\mathsf{self}_{\mathit{loc}}\}}\right) : s}\ \text{T-Leap}}{\Gamma \centerdot \hat{\gamma} \vdash e_{\mathsf{right}} : s}\ \text{T-Under}$$

Figure 5.9 Type Derivation for Result of SURROUND Rule in Subject Reduction Proof

Let $b = \langle \alpha, \beta_0, \ldots, \beta_q \rangle$. Assume $\alpha = var' \mapsto loc'_\delta$ and $\beta_0 = var_0$.[5] Then expanding the *typeBind* term in (5.5) gives

$$\Gamma, \text{reply}{:}\text{readonly } u_\top, \text{this}{:}\Gamma(loc), var'{:}\delta\,\Gamma\left(loc'\right), (var_i : t_i)_{i\in\{1..q\}\cdot\beta_i = var_i} \mathbf{.} \mathit{depClose}\left(\{\text{self}_{loc}\}\right) \vdash e_\text{a} : s_\text{a}. \quad (5.6)$$

Working the other direction, expanding the binding substitution in e'_a gives

$$e'_\text{a} = e_\text{a}\{\!|loc/\text{this}, loc'_\delta / var', (v_i / var_i)_{i\in\{1..q\}\cdot\beta_i = var_i}|\!\}. \quad (5.7)$$

By two hypotheses of T-CHAIN in (5.4) on page 221, we have

$$\forall i \in \{1..n\} \cdot \left(\Gamma \mathbf{.} \hat{\gamma} \vdash v_i : u_i \text{ where } u_i \preccurlyeq s_i\right).$$

For each value these judgments must be by T-LOC or T-NULL, neither of which use $\hat{\gamma}$ in its hypotheses. So we have $\forall i \in \{1..n\} \cdot \left(\Gamma \mathbf{.} \mathit{depClose}\left(\{\text{self}_{loc}\}\right) \vdash v_i : u_i \text{ where } u_i \preccurlyeq s_i\right)$. This fact, (5.6), and (5.7) satisfy the condition of Lemma 5.4 (Substitution) on page 215. Thus, $\Gamma, \text{reply}{:}\text{readonly } u_\top \mathbf{.} \mathit{depClose}\left(\{\text{self}_{loc}\}\right) \vdash e'_\text{a} : s''_\text{a}$ for some $s''_\text{a} \preccurlyeq s_\text{a}$. Finally, from (5.4), $s \preccurlyeq u_\top \preccurlyeq \text{readonly } u_\top$. So by Lemma 4.13 (Environment Subtyping) on page 162, $\Gamma, \text{reply} : s \mathbf{.} \mathit{depClose}\left(\{\text{self}_{loc}\}\right) \vdash e'_\text{a} : s'_\text{a}$ for some $s'_\text{a} \preccurlyeq s''_\text{a} \preccurlyeq s_\text{a}$.

The argument that e'_b is well typed is similar, but without the extra complications for dealing with reply.

So, all the leaf hypotheses in Figure 5.9 on the facing page hold. Therefore, $\Gamma \mathbf{.} \hat{\gamma} \vdash e_\text{right} : s$ and, by Lemma 4.11 (Replacement) on page 158, $\Gamma \mathbf{.} \hat{\gamma} \vdash e' : t$.

Case 3—BIND. Here:

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

This case is quite similar to the BIND case from the proof for MiniMAO$_2$ (see page 182). The essential difference is dealing with surround advice body tuples.

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

We will see that $J' \approx S'$. Let $e_\text{left} = \text{joinpt } (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\, v_0, \ldots, v_n\,)$. 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 with $\Gamma \mathbf{.} \hat{\gamma} \vdash e_\text{left} : s$. Thus, if v_{opt} is a location it must be in $dom(\Gamma)$ and so $J' \approx S'$.

It remains to show that $\Gamma \mathbf{.} \hat{\gamma} \vdash e' : t$. Let

$$e_\text{right} = \text{chain } \bar{B}, (\!|k, v_{opt}, m_{opt}, l_{opt}, (s_0 \times \ldots \times s_n \to s), \hat{\gamma}'|\!)(\, v_0, \ldots, v_n\,).$$

(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

[5] Handling the cases where either $\alpha = -$ or $\beta_0 = -$, or both, is a straightforward simplification.

satisfied for e_{right}.

By the well-typedness of e_{left} and its subterms, let $\Gamma \centerdot \hat{\gamma} \vdash v_i : t_i$ for all $i \in \{0..n\}$. By T-JOIN, we have $t_i \preccurlyeq s_i$ for all $i \in \{0..n\}$, $\mathit{depClose}(\hat{\gamma}') \subseteq \hat{\gamma}$, and $\left(\mathit{readonly}(u_0) = \text{readonly}\right) \implies \left(\hat{\gamma}' = \emptyset\right)$.

It remains to show the $\forall B \in \bar{B} \cdot \Gamma \centerdot \hat{\gamma}' \centerdot (s_0 \times \ldots \times s_n \to s) \vdash B$ OK. There are two cases to consider, depending on whether B is an around-advice body tuple, $[\![\ldots]\!]$, or a surround-advice one, $[\![\ldots]\!]_{\text{S}}$. Around-advice body tuples are treated in the BIND case of the proof for MiniMAO$_2$, so I omit that argument here. Let

$$B = [\![b, \mathit{loc}, \left(e_{\text{b}}, e_{\text{a}}\right), u_\top]\!]_{\text{S}}$$

be an arbitrary, surround-advice element of $\bar{B}$. By the definition of *adviceBind*, it must be the case that there exists a piece of advice with advice table entry $\left\langle \mathit{loc}, \mathit{pcd}, \left(e_{\text{b}}, e_{\text{a}}\right), u_\top \right\rangle_{\text{S}}$ such that $\mathit{matchPCD}_{\text{S}}(J', \mathit{pcd}, S) = b \neq \bot$.

By Lemma 5.6 (Surround Binding Soundness) on page 215 we have:

$$s \preccurlyeq u_\top$$
$$\emptyset \vdash b \text{ OK}$$
$$\Gamma, \text{this} : \Gamma(\mathit{loc}), \mathit{typeBind}(\Gamma, b, \langle s_0, \ldots, s_n \rangle) \centerdot \mathit{depClose}\left(\{\text{self}_{\mathit{loc}}\}\right) \vdash e_{\text{b}} : s_{\text{b}} \text{ for some } s_{\text{b}}$$
$$\Gamma, \text{reply} : \text{readonly } u_\top, \text{this} : \Gamma(\mathit{loc}), \mathit{typeBind}(\Gamma, b, \langle s_0, \ldots, s_n \rangle) \centerdot \mathit{depClose}\left(\{\text{self}_{\mathit{loc}}\}\right) \vdash e_{\text{a}} : s_{\text{a}} \text{ for some } s_{\text{a}}$$

By appropriate α-conversion of b, e_{b}, and e_{a}, we have $\Gamma \vdash b$ OK. The remaining hypotheses of T-BODS are satisfied directly by the results of the lemma. Thus, $\Gamma \centerdot \hat{\gamma} \vdash e_{\text{right}} : s$ by T-CHAIN. By T-UNDER and Lemma 4.11 (Replacement) on page 158 on page 158, $\Gamma \centerdot \hat{\gamma} \vdash e' : t$.

Case 4—ADVISE. Although the T-CHAIN rule in MiniMAO$_3$ differs from that for MiniMAO$_2$, the proof of this case is the same as that in Theorem 4.17 (Subject Reduction). This is because the ADVISE rule only applies when the first advice tuple in the chain is for around advice. The updated Lemma 5.7 (Advice Chaining) on page 220 is sufficient to show the claim. □

I restate the Progress theorem, as I did Theorem 5.8 (Subject Reduction), to account for MiniMAO$_3$'s private concern domains. The proof of the theorem differs in only minor ways from that of Theorem 4.18 (Progress) on page 185; progress is trivial for the new SURROUND and LEAP rules.

Theorem 5.9 (Progress). *Given a well-typed program, P, with public concern domains $\hat{g}$ and private concern domains $\hat{g}'$, for an expression e, a valid store S, a stack J consistent with S, a concern-complete type environment Γ consistent with S, a set of concern domains $\hat{\gamma}$ such that $\hat{g}' \subseteq \hat{\gamma} \subseteq \left(\hat{g} \cup \hat{g}'\right)$, and the evaluation dependency table DT, such that the triple $\langle e, J, S \rangle$ is reached in the evaluation of P, if $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$ then either:*

— *$e = \mathit{loc}_\delta$ for some δ and $\mathit{loc} \in \mathit{dom}(S)$,*

— *$e = \mathit{null}_\delta$ for some δ, or*

— *one of the following hold:*

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

Proof. The proof is the same as that for Theorem 4.18 (Progress) except for two differences:

— When the current redex matches the additional SURROUND or LEAP rules, progress is trivial; there are no side conditions.

— When the current redex is a chain expression (Case 2 of the original proof, on page 185), if the advice chain, $\bar{B}$, is non-empty then:

– $\langle e, J, S\rangle$ may evolve by ADVISE, as in the original proof, or

– $\langle e, J, S\rangle$ may evolve by SURROUND. □

The statement of the Type Safety theorem for MiniMAO$_3$ is exactly like that for MiniMAO$_2$, apart from referencing other meta-theory from the current chapter. But because of the centrality of the theorem, I repeat it here.

Theorem 5.10 (Type Safety). *Given a program P, with main expression e, concern domains $\hat{g}$, $\vdash P$ OK, and a valid store S_0, then either the evaluation of e diverges or else $\langle e, \bullet, S_0\rangle \overset{*}{\hookrightarrow} \langle x, J, S\rangle$ and one of the following hold for x:*

— *$x = loc_\delta$ for some δ and $loc \in dom(S)$,*

— *$x = null_\delta$ for some δ,*

— *$x = NullPointerException$, or*

— *$x = ClassCastException$*

Proof. If e diverges then the claim holds. If e converges, then note that the empty stack is consistent with any store, the validity of S_0 implies the existence of an initial type environment consistent with S_0, and $\vdash P$ OK implies $\Gamma \centerdot \hat{g} \vdash_{DT_P} e : t$ for some t, where $DT_P = \bigcup_{g \in \hat{g}} (g, g)$. Let DT be the evaluation dependency table for P. By the definition of the evaluation dependency table for MiniMAO$_3$ (see Section 5.1.3.1 on page 208), $DT_P \subseteq DT$ and $\forall g \in \hat{g} \cdot (g, g) \in DT$. Because e is the main expression of the program, it only contains user syntax. Also, because $\hat{g}$ includes every concern domain in P, $\hat{g} = depClose_{DT}(\hat{g})$. Thus, by Lemma 5.3 (Dependency Table Extension) on page 214, $\Gamma \centerdot \hat{g} \vdash_{DT} e : t$.

The proof (by induction on the number of evaluation steps) follows from Theorem 5.8 (Subject Reduction) on page 220 and Theorem 5.9 (Progress) on the preceding page. □

5.2.3 Effects

The more interesting meta-theory for MiniMAO$_3$ is that for its effects control mechanisms. As in MiniMAO$_2$, effects clauses are sound for public concern domains, but in MiniMAO$_3$ private concern domains may change without explicit permission. However, the very privacy of these domains keeps those changes from affecting other code. I also relax the conditions for the Read-Only theorem. The new conditions allow spectators, demonstrating that a simple alias control mechanism is effective in the presence of some sorts of aspects.

5.2.3.1 Effects Clauses

The definition of concern domains in the store is as written in Definition 4.20 (Concern Domain) on page 187. The Expression Typing Monotonicity lemma for MiniMAO$_3$ adds a lower bound on the set of writable domains used in a subderivation. This bound is the set of private concern domains. This lemma (apart from the lower bound) was immediate by inspection in MiniMAO$_2$. But because of the lower bound and the T-BODS rule, which does not make the monotonicity property explicit, the proof is marginally more complex here.

Lemma 5.11 (Expression Typing Monotonicity). *If* $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$, $\hat{g} = \{self_{loc} \cdot (self_{loc}, self_{loc}) \in DT\}$, *and* $\hat{g} \subseteq \hat{\gamma}$, *then for any subderivation* $\Gamma' \centerdot \hat{\gamma}' \vdash_{DT} e' : t'$, *it is the case that* $\hat{g} \subseteq \hat{\gamma}' \subseteq \hat{\gamma}$.

Proof. The proof is by structural induction on the derivation $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$. The base cases are T-NEW, T-OBJ, T-VAR, T-LOC, and T-NULL, which hold vacuously.

The remaining rules are the induction steps. The induction hypothesis is that the claim holds for all derivations smaller than the one being considered. We proceed by cases on the remaining typing rules. For each rule, we must consider the hypotheses that are expression typing judgments. For each such hypotheses, $\Gamma' \centerdot \hat{\gamma}' \vdash_{DT} e' : t'$, if we can show

$$\hat{g} \subseteq \hat{\gamma}' \subseteq \hat{\gamma}, \tag{5.8}$$

then the induction hypothesis is applicable and the claim holds.

The hypotheses of T-CALL, T-GET, T-SET, T-CAST, T-SEQ, T-PROC, T-UNDER, T-JOIN, and T-LEAP satisfy (5.8) trivially; for all such hypotheses $\hat{\gamma}' = \hat{\gamma}$.

Case 1—T-EXEC. For T-EXEC, there are two hypotheses that are expression typing judgments. For one, the set of writable concern domains is $\hat{\gamma}$, so (5.8) is satisfied. For the other hypothesis, the set of writable concern domains is $depClose_{DT}(\hat{\gamma}'')$, where $\hat{\gamma}''$ comes from the fun term in the judgment of T-EXEC. But T-EXEC also gives that $depClose_{DT}(\hat{\gamma}'') \subseteq \hat{\gamma}$. By the definition of *depClose*, $\hat{g} \subseteq depClose_{DT}(\hat{\gamma}'')$. Thus (5.8) is also satisfied for this hypothesis.

Case 2—T-CHAIN. For T-CHAIN, (5.8) is satisfied trivially for the argument expressions.

For around-advice body tuples, we have the advice body expression typed (in T-BOD) using a set of writable concern domains $depClose_{DT}(\hat{\gamma}'')$, where $\hat{\gamma}''$ comes from the around-advice body tuple. Similar to the T-EXEC case, we have $\hat{g} \subseteq depClose_{DT}(\hat{\gamma}'') \subseteq \hat{\gamma}$, where the upper bound is transitive through hypotheses of T-BOD and T-CHAIN.

For surround-advice body tuples, we have the before- and after-part expressions typed using the set of writable concern domains $depClose_{DT}(\{\mathrm{self}_{loc}\})$. By the definition of $depClose_{DT}$, $depClose_{DT}(\{\mathrm{self}_{loc}\}) = \hat{g}$. Thus, (5.8) is satisfied and the claim holds. □

The statements of Lemma 4.22 (Domain Preservation) and Theorem 4.23 (Tag Frame Soundness) are as written for MiniMAO$_2$. But there is a crucial difference in their meaning. The definition of *depClose* in MiniMAO$_3$ includes all private concern domains in every dependency closure. So the fact of the lemma and subsequent theorem—that $\forall g \in (\hat{g} \setminus depClose_{DT}(\hat{\gamma})) \cdot S|g = S'|g$—means that the meta-theory of effects clauses only holds the specified public concern domains constant; private concern domains may be mutated.

Lemma 5.12 (Domain Preservation). *Let P be a well-typed program with concern domains $\hat{g}$ and evaluation dependency table DT. Suppose the evaluation step $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle \hookrightarrow \langle \mathbb{E}[e'], J', S'\rangle$ occurs in an evaluation of P. Then $\forall g \in \left(\hat{g} \setminus depClose_{DT}\left(\hat{\gamma}\right)\right) \cdot S|g = S'|g$.*

Proof. By $\vdash P$ OK, Theorem 5.8 (Subject Reduction), and Theorem 5.9 (Progress) we know that $\mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}]$ is well typed. Therefore $\langle e\rangle_{\delta,\hat{\gamma}}$ is also well typed. This must be by T-TAG. By the hypotheses of that rule, there must be some Γ and t such that $\Gamma \centerdot depClose_{DT}\left(\hat{\gamma}\right) \vdash_{DT} e : t$.

Let e'' be the current redex of $\mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}]$. Let $\mathbb{E}'[-]$ be defined such that $\mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}] = \mathbb{E}[\langle \mathbb{E}'[e'']\rangle_{\delta,\hat{\gamma}}]$. Let $\hat{g}'$ be the set of private concern domains of P. By the definition of $depClose_{DT}$, $\hat{g}' \subseteq depClose_{DT}\left(\hat{\gamma}\right)$. So, $\Gamma \centerdot depClose_{DT}\left(\hat{\gamma}\right) \vdash_{DT} e : t$ implies, by Lemma 5.11 (Expression Typing Monotonicity), that there exists $\hat{\gamma}' \subseteq depClose_{DT}\left(\hat{\gamma}\right)$ such that $\Gamma \centerdot \hat{\gamma}' \vdash_{DT} e'' : s$ for some type s.

The remainder of the proof proceeds exactly like that for Lemma 4.22 (Domain Preservation). □

Theorem 5.13 (Tag Frame Soundness). *Let P be a well-typed program with concern domains $\hat{g}$ and evaluation dependency table DT. Suppose the evaluation triple $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle$ appears in an evaluation of P. Then either the evaluation diverges or $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle \overset{*}{\hookrightarrow} \langle \mathbb{E}[v], J', S'\rangle$, where $\forall g \in \left(\hat{g} \setminus depClose_{DT}\left(\hat{\gamma}\right)\right) \cdot S|g = S'|g$.*

Proof. By inspection of the semantics, to reach a value with the tagged expression removed the evaluation must be

$$\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle \overset{*}{\hookrightarrow} \langle \mathbb{E}[\langle v\rangle_{\delta,\hat{\gamma}}], J', S'\rangle \hookrightarrow \langle \mathbb{E}[v_\delta], J', S'\rangle .$$

The claim holds for the last step of this evaluation since the store is unchanged by the TAG rule. The claim holds for each of the other steps in this evaluation by Lemma 5.12 (Domain Preservation). □

5.2.3.2 Privacy

This subsection describes what it means for a concern domain to be private. The following lemma relates the extent of an object's graph to its home domain. It says that if one object can mutate another, then they must have the same home domain. (The lemma also holds for MiniMAO$_2$, though it is not needed there.)

Lemma 5.14 (Write Home). *Let S be a valid store occurring in the evaluation of a well-typed program P. Let $loc \in dom(S)$ and $\mathbb{G}_S(loc) = (L, E)$. Then*

$$\forall loc' \in L \cdot \left(\left(loc, loc'\right) \in writeReach(S)\right) \implies \left(home_S(loc') = home_S(loc)\right).$$

Proof. I prove the contrapositive.

Pick an arbitrary location, $loc' \in L$. If $home_S(loc') = home_S(loc)$, then the claim holds trivially. So assume not. By the definition of $home_S$, $home_S(loc') \neq home_S(loc)$ implies $loc' \neq loc$. There must exist some sequence of n annotated pairs, $\left(loc_{i-1} \overset{f_i}{\mapsto} loc_i\right) \in E$, where $loc_0 = loc$ and $loc_n = loc'$.

Suppose, for the purpose of showing a contradiction, that $\left(loc, loc'\right) \in writeReach(S)$. Then there exists a such a sequence of annotated pairs such that none of the f_i are read-only. Choose any such sequence. Since $home_S(loc') \neq home_S(loc)$, there must exist some $j \in \{1..n\}$ such that $home_S(loc_{j-1}) \neq home_S(loc_j)$. Let $S\left(loc_{j-1}\right) = \left[T\langle g_1, \ldots, g_p\rangle \centerdot F\right]$ and let $domains(T) = \langle G_1, \ldots, G_p\rangle$.

Consider field f_j and let $\delta\ U\langle G'_1,\ldots,G'_q\rangle$ be its declared type. From the typing of field declarations in T-CLASS, T-ASP, and T-SPEC, we see that either $G'_1\{\!|g_1/G_1,\ldots,g_p/G_p|\!\} = g_1$ or else $\delta = \mathsf{readonly}$.

We have assumed that $\delta \neq \mathsf{readonly}$. By Theorem 5.10 (Type Safety) and the positional invariance of domains under subtyping, $home_S(loc_j) = G'_1\{\!|g_1/G_1,\ldots,g_p/G_p|\!\} = g_1$. But $home_S(loc_{j-1}) = g_1$, contradicting $home_S(loc_{j-1}) \neq home_S(loc_j)$. So it must be the case that $\big(loc, loc'\big) \notin writeReach(S)$.

Thus,

$$\forall\, loc' \in L \cdot \big(home_S(loc') \neq home_S(loc)\big) \implies \big((loc, loc') \notin writeReach(S)\big),$$

and so the claim holds. □

The following definition formalizes the notion of private concern domains as a property of the store. The definition says that any object in the home domain of some spectator may not be in the object graph of another location, unless that location names the private home domain of the spectator. This corresponds to privacy because only a spectator, and any objects it dynamically creates, may mention the spectator's private home domain name.

Definition 5.15 (Privacy Respecting Store). Let S be a store and P a program. Let

$$S_S = \big\{(loc \mapsto [t\,.\,F]) \in S \cdot isSpectator(t)\big\}.$$

The store S *respects privacy* if

$$\forall\, loc_1, loc_2 \in S \cdot \big(\big(\exists loc_S \in S_S \cdot \mathsf{self}_{loc_S} = home_S(loc_1) \notin domains_S(loc_2)\big) \implies loc_1 \notin rep_S(loc_2)\big).$$

The following theorem states that all "naturally occurring" stores in MiniMAO$_3$ respect privacy.

Theorem 5.16 (Respect for Privacy). *For any valid store S occurring in the evaluation of a well-typed program P, S respects privacy.*

Proof. The proof is by induction on the number of evaluation steps up to the occurrence of the store S. The base case is the initial store, S_0, for the evaluation of P. By the definition of program evaluation in MiniMAO$_3$, S_0 is a valid store that respects privacy. To see that such a store exists, let S_0 be the minimal valid store for P. Pick any two locations $loc_1, loc_2 \in dom(S_0)$. Suppose $loc_1 \in rep_{S_0}(loc_2)$. Because

$$\forall [t\,.\,F] \in rng(S) \cdot rng(F) = \{\mathsf{null}\},$$

it must be that $loc_1 = loc_2$. Thus

$$home_{S_0}(loc_1) = home_{S_0}(loc_2) \in domains_{S_0}(loc_2),$$

and S_0 respects privacy.

For the induction, consider an evaluation step $\langle \mathbb{E}[e], J, S\rangle \hookrightarrow \langle \mathbb{E}[e'], J, S'\rangle$ in the evaluation of P. The induction hypothesis is that S respects privacy. For each possible evaluation rule, we prove that S' also respects privacy. For all evaluation rules except NEW and SET, $S' = S$, so this holds trivially. Consider the two non-trivial cases and let e_{left} be the current redex and e_{right} be the result such that $e = \mathbb{E}'[e_{\text{left}}]$ and $e' = \mathbb{E}'[e_{\text{right}}]$ for some $\mathbb{E}'$.

Case 1—NEW. $e_{\text{left}} = \text{new } t()$, $e_{\text{right}} = loc''$, $loc'' \notin dom(S)$, $S' = S \oplus \left(loc'' \mapsto [t \,.\, F]\right)$, and $rng(F) = \{\text{null}\}$.

Because loc'' is fresh and $rng(F) = \{\text{null}\}$, there is no $loc \in dom\left(S'\right)$ such that either $loc'' \in rep_{S'}(loc)$ or $loc \in rep_{S'}(loc'')$. Thus, by the induction hypothesis, S' respects privacy.

Case 2—SET. $e_{\text{left}} = \left(loc''.f{=}v\right)$, $e_{\text{right}} = v$, $S\left(loc''\right) = [t \,.\, F]$, $S' = S \oplus \left(loc'' \mapsto \left[t \,.\, F \oplus \left(f \mapsto v'\right)\right]\right)$, and

$$v' = \begin{cases} loc' & \text{if } v = loc'_{\delta'} \\ \text{null} & \text{otherwise} \end{cases}$$

(Since e_{left} is well typed, we can omit any δ-subscript on loc''.)

The only object changed in S' versus S is $S\left(loc''\right)$. It $v' = \text{null}$, then $\forall\, loc \in dom\left(S'\right) \cdot rep_{S'}(loc) \subseteq rep_S(loc)$, and S' respects privacy by the induction hypothesis.

So assume $v' = loc'$. After the assignment, $loc' \in rep_{S'}(loc'')$. Suppose there is some mapping

$$\left(loc_{\text{S}} \mapsto \left[s \,.\, F'\right]\right) \in S'$$

such that $isSpectator(s)$ and $home_{S'}(loc') = \text{self}_{loc_{\text{S}}} \notin domains_{S'}(loc'')$. We will see that this leads to a contradiction. By Theorem 5.10 (Type Safety), e_{left} is well typed. Therefore, $home_{S'}(loc') \in domains(fieldsOf(t)(f))$ by T-SET and the definition of subtyping. But that implies, by the definition of *fieldsOf*, that $\text{self}_{loc_{\text{S}}} \in domains_{S'}(loc'')$, a contradiction. Thus there can be no spectator instance that witnesses to the violation of respect for privacy. S' respects privacy.

So for all possible evaluation steps, respect for privacy is maintained. Thus, by induction, the claim holds. □

Theorem 5.16 (Respect for Privacy) on the facing page and Lemma 5.14 (Write Home) on page 227 imply that any object writable by a spectator may not be referenced by another aspect or by code in the base program. This is what allows reasoning about the program while ignoring spectators.

Rather than treating respect for privacy as a separate property, an alternative way of handling this issue would be to define store validity to include the domain privacy property stated in the lemma. However, that would require threading the proof of domain privacy maintenance through the subject reduction proof. In order to use as much meta-theory of MiniMAO$_2$ as possible for MiniMAO$_3$, I choose to separate the domain privacy property.

5.2.3.3 Read-Only Annotations

Definition 4.24 (Reach), Definition 4.25 (Writable Reach), and Definition 4.26 (Object Graph) apply to MiniMAO$_3$ as written. The statement of Definition 4.27 (Included Locations) on page 191 also applies, but the *locations* function must handle the new syntax of MiniMAO$_3$. For clarity, I restate the definition.

Definition 5.17 (Included Locations). Given an expression e, the set of locations *included* in e, denoted $locations(e)$, is given by the recursive definition in Figure 4.25 on page 192 plus the following:

$$locations(e_0 \curvearrowright e_1) = locations(e_0) \cup locations(e_1)$$

$$locations\left(\llbracket b, loc, \left(e_{\text{b}}, e_{\text{a}}\right), \text{␣} \rrbracket_{\text{S}} + \bar{B}\right) = locations(b) \cup \{loc\} \cup locations\left(e_{\text{b}}\right) \cup locations(e_{\text{a}}) \cup locations\left(\bar{B}\right)$$

Definition 4.28 (Home Domain) on page 191 also applies as written to MiniMAO$_3$.

The assumptions of Lemma 4.29 (Read-only Preservation) and Theorem 4.30 (Read-only Soundness), which disallow all aspects, can be relaxed for MiniMAO$_3$ to allow spectators. This is a consequence of Theorem 5.16 (Respect for Privacy) on page 228. Only the BIND case of the original proof for MiniMAO$_2$ relies on the strongest assumptions, so the proof below is updated without much trouble.

Lemma 5.18 (Read-only Preservation). *Suppose the evaluation triple* $\langle \mathbb{E}[e], J, S\rangle$ *appears in the evaluation of a well-typed program P. Let loc be a location in* $dom(S)$ *such that* $domains_S(loc) \subset \mathscr{G}$*, i.e.,* $S(loc)$ *only names public concern domains. Let* $\mathbb{G}_S(loc) = (L, E)$*, and let the following assumptions hold:*

Assumption 1. $\forall \delta \cdot \left(loc_\delta \in locations(e)\right) \implies \left(\delta = readonly\right)$*. (Intuitively, no write-enabled pointers to the object of interest appear in the expression.)*

Assumption 2. $\forall loc'_\delta \in locations(e) \cdot (\delta = \varepsilon) \implies \left(\forall loc'' \in rep_S(loc) \cdot \left(loc', loc''\right) \notin writeReach(S)\right)$*. (Intuitively, the expression does not contain any write-enabled pointers that reach into the graph of the object of interest.)*

Assumption 3. $\forall loc' \in dom(S) \cdot S\left(loc'\right) = [t.F] \implies isClass(t) \vee isSpectator(t)$*. (No assistant instances appear in the store.)*

If $\langle \mathbb{E}[e], J, S\rangle \hookrightarrow \left\langle \mathbb{E}[e'], J', S'\right\rangle$*, then*

Consequent 1. $\forall \delta \cdot \left(loc_\delta \in locations\left(e'\right)\right) \implies \left(\delta = readonly\right)$

Consequent 2. $\forall loc'_\delta \in locations\left(e'\right) \cdot (\delta = \varepsilon) \implies \left(\forall loc'' \in rep_S(loc) \cdot \left(loc', loc''\right) \notin writeReach(S')\right)$

Consequent 3. $\forall loc' \in dom\left(S'\right) \cdot S'\left(loc'\right) = [t.F] \implies isClass(t) \vee isSpectator(t)$

Consequent 4. $\mathbb{G}_S(loc) = \mathbb{G}_{S'}(loc)$*,*

Proof. The set up for the proof is exactly like that for Lemma 4.29 (Read-only Preservation) on page 191, as are all cases except that for BIND. I give the BIND case, and the new SURROUND and LEAP cases, for MiniMAO$_3$.

Case 1—BIND. $e_{\text{left}} = \text{joinpt } j\ (\ v_0, \ldots, v_n\ )$, $e_{\text{right}} = \text{under chain } \bar{B}, j\ (\ v_0, \ldots, v_n\ )$, $\bar{B} = adviceBind(j + J)$, and $S' = S$. Because $S' = S$, consequents 3 and 4 hold.

Now $locations\left(e_{\text{right}}\right)$ contains all elements of $locations\left(e_{\text{left}}\right)$. By the definitions of *locations* and *adviceBind*, $locations\left(e_{\text{right}}\right)$ also includes the locations of the aspects of any matching advice. Advice body expressions in $\bar{B}$ do not contribute any locations, by a similar argument to that for method bodies in Case 3 of the MiniMAO$_2$ proof. The other possible source of new locations for $locations\left(e_{\text{right}}\right)$ is the binding terms in $\bar{B}$. In particular, the left-most join point abstraction in $j + J$ of the form $(\!|\_, v, \_, \_, \_, \_|\!)$ may contribute v to $locations\left(e_{\text{right}}\right)$ because of a this pointcut descriptor.

By assumption 3 and the validity of S, there can be no matching around advice. There may, however, be surround advice. Let $[\![b, loc_S, \_, \_]\!]_S$ be an arbitrary surround-advice body tuple in $\bar{B}$.

By construction of the advice table, loc_S is not read-only and $home_S(loc_S) = \text{self}_{loc}$, a private concern domain. Let loc'' be an arbitrary element of $rep_S(loc)$. Assume for the purpose of showing a contradiction that $\left(loc_S, loc''\right) \in writeReach(S)$. Then Lemma 5.14 (Write Home) gives $home_S(loc'') = home_S(loc_S) = \text{self}_{loc_S}$. But by the statement of the lemma, $\text{self}_{loc_S} \notin domains_S(loc)$. So Theorem 5.16 (Respect for Privacy) implies that $loc'' \notin rep_S(loc)$, a contradiction. Thus, $\left(loc_S, loc''\right) \notin writeReach(S)$.

Finally, consider the binding term, b. If it does not include a this pointcut binding, then b does not introduce a new location. Also, if the this-bound location is read-only, then assumption 2 holds trivially for it. So assume $b = \langle var \mapsto loc', \ldots \rangle$. Let the pointcut that binds loc' be this(t var). By the definition of $matchPCD_S$ (see the this-binding rule), loc' being write-enabled implies $readonly(t) = \varepsilon$. But by T-SURR, all formal parameters have read-only types. Thus, the assumption that loc' is not read-only leads to a contradiction. For all possible locations in $\left(locations\left(e_{\text{right}}\right) \setminus locations\left(e_{\text{left}}\right)\right)$, assumption 2 holds.

Case 2—SURROUND. Here

$$
\begin{aligned}
e_{\text{left}} &= \text{chain } [\![b, loc_S, \left(e_b, e_a\right), \sqcup]\!]_S + \bar{B}, j \text{ (} v_0, \ldots, v_n \text{)} \\
e_{\text{right}} &= \text{under } \left(\left(\langle e'_b \rangle_{\varepsilon, \left\{\text{self}_{loc_S}\right\}}\text{; chain } \bar{B}, j \text{ (} v_0, \ldots, v_n \text{)}\right) \frown \langle e'_a \rangle_{\varepsilon, \left\{\text{self}_{loc_S}\right\}}\right) \\
e'_b &= e_b \{\!| loc_S / \text{this} |\!\}\{\!|(v_0, \ldots, v_n) / b|\!\} \\
e'_a &= e_a \{\!| loc_S / \text{this} |\!\}\{\!|(v_0, \ldots, v_n) / b|\!\} \\
S' &= S
\end{aligned}
$$

Because $S' = S$, consequents 3 and 4 hold. Examining the definition of binding substitution, we see that no new locations are introduced. Some locations may be dropped if not all formals appear in the before- and after-part expressions. So $locations\left(e_{\text{right}}\right) \subseteq locations\left(e_{\text{left}}\right)$, and consequent 2 holds.

Case 3—LEAP. Here $S' = S$ and $locations\left(e_{\text{left}}\right) = locations\left(e_{\text{right}}\right)$, so all the consequents hold. □

Theorem 5.19 (Read-only Soundness). *Suppose the evaluation triple $\langle \mathbb{E}[e], J, S \rangle$ appears in the evaluation of a well-typed program P. Let loc be a location in $dom(S)$ such that $domains_S(loc) \subset \mathcal{G}$, i.e., $S(loc)$ only names public concern domains. Let $\mathbb{G}_S(loc) = (L, E)$, and let the following assumptions hold:*

Assumption 1. $\forall \delta \cdot \left(loc_\delta \in locations(e)\right) \implies \left(\delta = readonly\right)$.

Assumption 2. $\forall loc'_\delta \in locations(e) \cdot (\delta = \varepsilon) \implies \left(\forall loc'' \in rep_S(loc) \cdot \left(loc', loc''\right) \notin writeReach(S)\right)$

Assumption 3. $\forall loc' \in dom(S) \cdot S\left(loc'\right) = [t \,.\, F] \implies isClass(t) \vee isSpectator(t)$.

If $\langle \mathbb{E}[e], J, S \rangle \overset{*}{\hookrightarrow} \left\langle \mathbb{E}[v], J', S' \right\rangle$, *then* $\mathbb{G}_S(loc) = \mathbb{G}_{S'}(loc)$.

Proof. Immediate by appealing to Lemma 5.18 (Read-only Preservation) at each step in the evaluation. □

5.3 Discussion

By Theorem 5.19 (Read-only Soundness), spectators can be used in a program without breaking the alias-control mechanism. The first two assumptions of the theorem are local properties of an expression. The other assumption just restricts the sorts of programs that are considered. So the statement of the theorem can be viewed as a formalization of local reasoning about the expression. In the last chapter, I argued that allowing assistants in the program would break this local reasoning property. In this chapter, I have demonstrated that spectators, surround advice, and private concern domains can be used to allow some kinds of aspects while maintaining the local reasoning property. I also noted in the last chapter that Read-only Preservation holds

with assistants if we consider all applicable assistants in assumption 2 of the theorem. This corresponds to the notion that applicable assistants must be considered for sound reasoning.

Theorem 5.13 (Tag Frame Soundness) on page 227 allows unseen, private concern domains to be modified during method or advice execution. However, because of Theorem 5.16 (Respect for Privacy) on page 228, one can still reason about the effects of a method or piece of advice. To reason about the execution of a method or piece of advice one must know its signature including its effects clause, the concern domains of the target object, and the configuration of assistants in the program, as represented by the aspect instantiation instructions and dependency declarations. By Theorem 5.16 (Respect for Privacy), if the concern domains of the target object do not include any private concern domains, then no changes made by unseen spectators will be visible in the code being considered. The side effects of spectators are effectively sequestered. Thus, spectators can be used non-invasively, as claimed in Chapter 2. Only the configuration of assistants must be known to reason about the effects of a block of code.

5.4 Related Work

Surround advice as formalized here is very similar to "harmless advice" [48]. Both restrict advice to always proceeding and never mutating arguments. Unlike the harmless advice design, MiniMAO$_3$ also includes more powerful around advice—Dantas and Walker [48] might call it "harmful"—that is not restricted. A more complete discussion of their work appears in Section 4.4.3.2. I also refer the reader to that section for a discussion of the other work related to MiniMAO$_3$.

5.5 Conclusion

In this chapter, I presented MiniMAO$_3$. MiniMAO$_3$ uses concern domains and read-only annotations to formalize spectator aspects as conceived in Chapter 2. Spectator aspects include *surround* advice that obeys the control flow restrictions proposed in Section 2.2.2. Surround advice also obeys the restriction on mutation proposed there.

In MiniMAO$_3$, even with unseen spectator aspects, the reasoning properties of MiniMAO$_2$ still hold. One does not need to know about the spectators present in a program in order to reason about the base program. Spectators can be used non-invasively without sacrificing modular reasoning. Only the configuration of assistants must be known to reason about the effects of a block of code. The key to this is my proof of "respect for privacy": the portion of the store mutable by a spectator cannot be observed by any objects that are not part of the spectator's representation.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

In the introduction to this dissertation, I stated my thesis that *there exists a discipline for programming in aspect-oriented languages with dynamic-context pointcut descriptors that (1) allows modular reasoning, (2) permits the use of existing aspect-oriented idioms for separation of concerns, (3) can be verified by a combination of static typechecking and simple verification conditions, and (4) can be incorporated into a practical, aspect-oriented language.*

I conclude my dissertation by reflecting on how this work supports the four claims of my thesis, and outlining the main open problems. I sketch directions for future work on modular reasoning for aspect-oriented languages and conclude with some more general reflections.

6.1 Support for the Thesis

Chapter 2 presented the MAO discipline for modular aspect-oriented reasoning. The MAO discipline addresses the twin problems of modular reasoning in aspect-oriented languages: unseen aspects may apply to the code, and aspects may be developed without complete knowledge of the code that will be advised. The discipline addresses these problems by separating aspects into two sorts: spectators and assistants. The discipline also requires that the aspect author and the programmer of advised code share the burden of ensuring modular reasoning.

Chapter 2 also described additional language features to facilitate the MAO discipline. These new features include accepts clauses and concern maps for explicit acceptance of assistance. Explicit acceptance allows assistant aspects, which have the full power of AspectJ's aspects, to be modularly identified, and thus considered when reasoning. The new features also provide for spectator declarations. Spectators are statically constrained to not modify the behavior of the modules that they view. This allows modular reasoning about the advised code, even if spectators remain unseen.

To demonstrate how explicit acceptance of assistance enables modular reasoning, Chapter 2 presented my extensions to the Java Modeling Language that allow one to write specifications for advice. These features allow one to write abstract specifications for around advice that model most compositions possible with proceed expressions. Based on the specification constructs, I presented an algorithm for composing specifications that allows one to determine the effective specification for any method call or execution in the presence of advice from accepted assistance. This composition process requires only modularly identifiable specifications and demonstrates the reasoning process that would be used even in the absence of formal specifications, thus supporting claim 1 of my thesis.

Because my proposed language features change the semantics of advice binding in AspectJ, Chapter 2 also included my evaluation of the practical effects of my proposal. An analysis of existing code samples showed that current aspect-oriented idioms could be coded within my proposal, supporting claim 2 of my thesis, and providing anecdotal support for claim 4. The ready identification of places to accept assistance from

client or implementation utilities in these samples supports my contention that experienced aspect-oriented programmers are already using disciplines, like the MAO discipline, that enable modular reasoning.

To validate the results of Chapter 2, I had to demonstrate that the claimed non-interference property of spectators could be statically checked. The remaining chapters built the formalism required to prove this claim, introducing several novel techniques in the process. The meta-theory of my formalism also provides partial support for claim 3 of my thesis.

Chapter 3 introduced MiniMAO$_1$, my core calculus for AspectJ. MiniMAO$_1$ faithfully explains the semantics of AspectJ's around advice on method call and execution join points. In particular, MiniMAO$_1$ is the first aspect-oriented formalism to model the possibility that advice can change the target object at a join point and affect method dispatch. MiniMAO$_1$ models the fact that, in AspectJ, advice that changes the target object at a call join point may change the method dispatched to, while advice that changes the target object at an execution join point will not affect the dispatched method.

AspectJ is not statically type safe [74]. With MiniMAO$_1$, I demonstrated that the type safety problems extend to the ability to change target objects in advice. To provide a solid foundation for formalizing the reasoning issues that I am concerned with, I devised changes in advice matching and pointcut typing in MiniMAO$_1$ that allow for static type safety. I introduced the concept of *binding soundness* for proving the soundness of my static type system.

Chapter 4 extended MiniMAO$_1$ with concern domains and a simple alias-control system using read-only references. The result, MiniMAO$_2$, enables efficient static detection of tangled code by lifting cross-cutting concerns from the program implementation into the type system. The type system enforces a non-interference property so that a global, signature-level search can identify all the code that might mutate a particular concern domain. Read-only references in MiniMAO$_2$ serve as a proxy for the reasoning issues involved in combining more general alias-control type systems with an aspect-oriented language. I proved that knowing the set of writable concern domains for a method or piece of advice (a modular property given the explicit acceptance of assistance) allows one to reason about the possible side effects of the code.

Chapter 5 presented MiniMAO$_3$. MiniMAO$_3$ uses concern domains and read-only annotations to formalize spectator aspects as conceived in Chapter 2. Spectator aspects include *surround* advice that obeys the control flow restrictions proposed in Section 2.2.2. Surround advice also obeys the restriction on mutation proposed there.

In MiniMAO$_3$, even with unseen spectator aspects, I proved that the reasoning properties of MiniMAO$_2$ still hold. One does not need to know about the spectators present in a program in order to reason about the program. Spectators can be used non-invasively without sacrificing modular reasoning. Only the configuration of assistants must be known to reason about the effects of a block of code. The key to this is my proof of "respect for privacy": the portion of the store mutable by a spectator cannot be observed by any objects that are not part of the spectator's representation.

6.2 Open Problems

My formal study provides strong, theoretical support for my thesis. My evaluation of existing AspectJ examples in Chapter 2 provides some evidence as to the practicality of my proposal. However, some open problems remain.

— To fully support claim 3, I must formalize the verification conditions entailed by my proposed specification language constructs, and prove that reasoning using effective specifications is sound.

— To fully support claim 4, I must demonstrate the incorporation of my proposed language features into a practical programming language. I must use that language to implement realistic-scale programs.

Each of these open problems entails a significant research program. I consider some of the issues involved in the following subsections.

6.2.1 Verification

In Chapter 2, I presented an algorithm for forming the effective specification for a method call in the presence of accepted assistance. I argued that this effective specification was intuitively correct, and that properly constrained spectators could be soundly neglected in reasoning about the method call. The meta-theory of MiniMAO$_2$ and MiniMAO$_3$ provides more supporting evidence for this argument.

However, more work is needed to support my claim of static verification in the MAO discipline. To prove the claim, I envision developing MiniMAO, as an extension to MiniMAO$_3$. In MiniMAO I would

— add constructs to MiniMAO$_3$ for giving specifications in the desugared form of Section 2.4.2.1 (i.e., as quantified variables, pre- and postconditions, and frame axioms);

— develop an axiomatic semantics for MiniMAO and prove it sound with respect to the operational semantics [70, 71]; and

— prove that the axiomatic semantics of an advised method call corresponds to the effective specification formed according to my algorithm.

I would need to show that in MiniMAO the actual code executed at run time, including spectators, corresponds to the effective specification, which excludes spectators. The key to this is that the effective specification is considered relative to the named, public concern domains of the method. Thus, the axiomatization of MiniMAO must accommodate concern domains.

It seems that separation logic may provide the leverage to make this work [126, 127, 143, 144]. As discussed in Section 4.5, the central idea in separation logic is to separate specification predicates in a "spatial conjunction" so that each refers to an unconnected, disjoint subset of the heap, where "unconnected" means the absence of pointers from one subset to the other. This unconnected-ness requirement is related to the restrictions on aliasing in Theorem 5.19 (Read-only Soundness). Concern domains, and private spectator domains, should provide the necessary substrate for applying separation logic to the verification problem in MiniMAO. O'Hearn et al. [127, §8] also discuss a notion of "memory faults", run-time errors that are signaled when code accesses a portion of the heap outside of the subset described in the specification of the code. A proof of correctness for the code must ensure that such memory faults cannot occur. The static type system of MiniMAO can ensure this property for write access. It may be that MiniMAO would also need *readable* domain sets, which would place a static bound on the set of domains that may be read by a piece of code.

6.2.1.1 Dynamic-Context Pointcut Descriptors

Another challenge in formally verifying aspect-oriented programs is dynamic-context pointcut descriptors. With pointcut descriptors like cflow in a language, static determination of whether a piece of advice will be executed at a given join point would seem to be undecidable.

As mentioned in Section 2.5.2, one potential solution to this problem would be to include dynamic-context predicates in the specification language. This would allow effective specifications to use guarded specification cases for advice that includes dynamic-context pointcut descriptors.

It might also be possible to simplify the set of dynamic-context pointcut descriptors. For example, the following combination of general, dynamic-context pointcut descriptors:

```
cflow(execution(* m(..))) && !cflowbelow(execution(* m(..))))
```

is a common idiom in AspectJ for matching the first execution of a method m. Instead of using general dynamic-context pointcut descriptors, perhaps a simpler, more specific descriptor could be used, like

```
firstExecution(* m(..)).
```

Other common idioms might require other simple pointcut descriptors. This more specific, but less powerful, set of dynamic-context pointcut descriptors may simplify the static analysis. For example, they might allow the axiomatic semantics of MiniMAO to include a simple abstraction of the call stack. If so, this abstraction could be used to choose between dynamic-context-guarded specification cases at join points. On the other hand, it may be that sacrificing the power of general, dynamic-context pointcut descriptors is too great a sacrifice in expressiveness. After all, these pointcut descriptors seem to be one of the unique contributions of AspectJ. (See my discussion of the work of Kiczales and Mezini [80] in Section 2.6.)

6.2.1.2 Multiple Proceeds

One benefit of detailed method specifications in a language like Java with JML, or Eiffel [110, 111], is that individual methods may be separately verified against their specifications. My proposed specification constructs should provide similar benefits for separate verification of advice. However, it is not precisely clear how to map the proceed-clause-separated subcases of advice specifications to the actual code of the advice. When the code of the advice is separated into neat blocks separated by proceed expressions that are always reached, the verification problem seems straightforward. However, proceed expressions in an advice body may be embedded in various looping and branching constructs. Because the specification for a piece of advice makes promises not just about the pre- and postconditions, but about the control flow, the proof obligations for showing that around advice is correct are stronger.

Furthermore, as mentioned in Section 2.4.1, some advice may proceed an indefinite number of times. Thus, future work must also consider specification constructs for describing this situation. Perhaps some of the JML techniques for specifying loops and loop invariants might apply [94, §12.2].

6.2.1.3 Mechanized Meta-theory[1]

One concern in extending MiniMAO_3 for formal verification is that proofs of the meta-theory are already quite involved. Thus, it may be sensible to use an automated reasoning system, such as Twelf [134], for any extensions. (Other potential systems are enumerated by Aydemier et al. [16, §1] in their POPLMARK challenge.) Although I am confident in the veracity of the proofs given in this work, only the most dedicated reviewer could check all the details presented therein. Machine checked proofs would increase others confidence in this work.

Using an automated system for proving the soundness of verification in MiniMAO would also open the door to investigating other features of AspectJ where the tedium of manual proofs would otherwise pose an obstacle. Among these would be the formalization of before and after advice, introductions, and the rest of AspectJ's pointcut descriptors. The automated system could also be used to investigate the use of concern

[1] This section heading comes from the tag line of the POPLMARK project,"Mechanized Metatheory for the Masses" [16].

domains for providing abstract descriptions of control flow, analogous to the subsystem annotations of Lam and Rinard [88] discussed in Section 4.5.

6.2.2 MAO

In Chapter 2, I described an evaluation of my proposed language features based on studying small examples of existing programs written in AspectJ. This provides some support for claim 4 of my thesis, that the MAO discipline can be incorporated into a practical, aspect-oriented language. However, to fully support my claim, I must actually incorporate the discipline into a practical language.

So, having laid the theoretical foundation with MiniMAO, an obvious next step is to develop a full-scale programming language—a variant of AspectJ—that provides my type safety and reasoning properties. I call this yet-to-be-developed language *MAO*. I would also like to extend JML [93, 94] for use in specifying other features of MAO programs beyond just around advice.

6.2.2.1 Approach

There are two approaches to implementing MAO that have complementary advantages and disadvantages.

The first approach would be to develop MAO as an extension of the Aspect Bench Compiler (ABC) [54]. ABC is a full implementation of a compiler for AspectJ, based on the Polyglot compiler framework [122] for its front end and the Soot optimization framework [155] for its back end. One of the main advantages of using ABC to implement MAO is that, being a Polyglot extension, my prototype implementation of concern maps and accepts clauses (discussed in Section 2.2.1.3) could be ported without much difficulty. The other advantage of building on ABC is that it was specifically designed as a framework for investigating extensions to AspectJ. ABC is strictly a compiler; it translates program text to machine code (or Java virtual machine code in this case). Merely implementing the translation semantics and typechecking for MAO is of relatively little technical interest. Although such an exercise is bound to uncover issues not considered in my formal study, my main motivation for implementing MAO is to study the issues that arise in using the language for real software development.

Most large-scale software development is presently done using integrated development environments. Thus, another approach to implementing MAO would be to extend the AspectJ Development Toolkit (AJDT) for ECLIPSE.[2] This approach would allow programmers to use MAO within a professional integrated development environment. It would also link the implementation of MAO to the reference implementation of AspectJ, allowing (forcing?) MAO to keep pace with the evolution of the core language. Furthermore, if the MAO extensions proved useful in practice, they could be more easily adopted into the core AspectJ language under this approach. Another advantage of this approach is that it provides the right environment for my proposed tool support for automatic generation of effective specifications (see Section 2.5.3). The major disadvantage of this approach is that the AJDT is not primarily designed to provided an extensible language platform, although the AJDT is open source, and so extensions are possible. However, because its primary goal is as a reference implementation, the AJDT is much more likely to change in ways that break extensions than is an extension framework like ABC.

[2] The AJDT is available from http://www.eclipse.org/aspectj, URL valid as of July 17, 2005.

6.2.2.2 Evaluation

Having developed an implementation of MAO, I would like to program and specify non-trivial systems using the MAO language and discipline. Questions to be answered include:

— Are concern domains that partition the heap an effective mechanism for statically separating concerns? Or are concerns, even in aspect-oriented programming, sufficiently tangled so that no statically-enforced separation is reasonable?

— Does the MAO discipline, like behavioral subtyping, help in guiding programmers' thinking and design efforts?

— Do effective specifications help programmers to reason about advised code?

— What proportion of aspects in large-scale systems are spectators? assistants?

— What tool support is necessary to help developers write useful specifications of advice?

Like many practical questions in programming languages, it is difficult and expensive to perform sufficiently large experiments to get significant answers to these questions. One way to achieve a larger sample is to make the tools freely available and encourage the development of a user community. Such an approach was key to the early success of AspectJ, and has also been beneficial for the JML project.

6.2.2.3 Other Features

Other interesting issues would arise with the development of a full-scale MAO language. One issue is adding concern domain annotations to the Java API. For this, I could borrow the technique of JML's .spec files [94]. These files allow one to write a specification for a module separately from the module's declaration. They are used in the JML project to provide behavioral specifications for a subset of the Java API.

Another issue that I would have to address is I/O. Disk input and output can be used as storage to pass data between domains that would otherwise be separated by the static type system. It may be sensible to consider such uses to be outside the scope of the type system's safety properties. Another possibility is to develop a locking I/O library that could confine certain files to certain concern domains. Such a library would have to interact more closely with the file system than Java's does. Also related to I/O is an investigation of how concern domains interact with graphical user interfaces. In my evaluation of proposed language features (see Section 2.3), I discussed the Debug aspect of AspectJ's spacewar example. This aspect attaches an additional menu item to the game's interface when the aspect is included in the system. To accommodate this with concern domains, it may be that all user interface elements must belong to a common concern domain. But how would this interact with my requirement that objects not maintain write-enabled pointers to other concern domains? It may be that the call-back architecture of most GUI frameworks would allow my restriction on write-enabled pointers to stand. Or it may be that the restriction would have to be loosened. I conjecture that it would be sufficient to restrict spectators to obeying the read-only restriction, but allow assistants and other classes to violate it. But it is not clear how this would affect regular objects in the representation of a spectator. Another possibility for concern domain references is to allow hierarchical concern domains, like the domains of Aldrich and Chambers [9].

To maintain the static type safety of MiniMAO, I would also have to investigate the use of introductions in AspectJ. Introductions in AspectJ are not statically type safe without a whole-program analysis. This is because two different aspects may introduce colliding methods to a class. The one that survives to run time is whichever

is added last. But a client may expect the other method. In MultiJava, we demonstrate how open classes can be used to achieve modular, static typechecking for introductions [38, 43, 46]. In fact, with open classes two different methods can be safely introduced to the same class, with a client deciding which of the two should be in scope. This lexical selection of applicable extensions was an early motivation for the current work.

6.3 Future Work

In additional to the open problems discussed above, the current work also suggests several other interesting lines of investigation. I discuss these briefly here.

6.3.1 Alias Control

I showed in Section 4.4.3.2 that the restrictions of my simple alias-control system could be violated in the presence of assistants. The private concern domains of spectators prevent them from violating the system in this way. The basic issue is that assistants can "leak" pointers into a computation. Adding a more expressive alias-control system to MiniMAO might allow more control over aliasing between assistant aspects and base program objects. On the other hand, the power of assistant aspects might break the more powerful alias-control system also.

Here are a couple of promising approaches to investigating this issue:

— The per-object domains of Aldrich and Chambers [9] might allow my concern domains and an alias-control system to be unified. In their work they control aliases in order to understand the aliasing patterns used in program architecture. They do not consider side-effect control.

— Rinard et al. [146] propose "abstract fields", which seem to be equivalent to Leino's data groups [97]. They use abstract fields to detect (through a global analysis) object-aspect interference without breaking data encapsulation. Abstract fields are also useful in providing a more abstract representation of any interference, uncluttered by the specific details of all the concrete fields. My modular, static type system uses concern domains to limit interference to that specifically allowed by the programmer. Perhaps concern domains closer in granularity to data groups could be used to statically prevent interference at a finer granularity.

6.3.2 Late Binding and Aspect-Oriented Virtual Machines

It seems that the non-interference properties of spectators might have implications for dynamic weaving [57, 112, 113, 135, 138, 151]. Because spectators cannot change the behavior of other modules, it seems natural to be able to apply them to a program that is already running, for example to diagnose a problem in a long-running server application. The generality of spectator application means that they can potentially be dispatched to at any join point. Thus, it seems that some form of virtual machine support for spectators might be interesting.

The Steamloom virtual machine includes dynamic aspect dispatch [19]. The CeasarJ aspect-oriented language uses Steamloom as its virtual machine to enabled dynamic deployment (and "undeployment") of aspects [113]. (The JRockit JVM from BEA Weblogic also includes support for dynamic weaving.) An aspect-oriented virtual machine like Steamloom would include facilities for applying and removing spectators from already running programs and for dispatching to spectators at the appropriate join points.

```
public class ArrayCoCon {
   public static void main(String[] args) {
      Integer[] i = new Integer[1];
      m(i); // triggers an ArrayStoreException at run time
   }

   public static void m(Object[] o) {
      System.out.println(o instanceof Object[]); // true for call above
      System.out.println(o instanceof Integer[]); // true for call above
      o[0] = new Object(); // no way to avoid an exception!
   }
}
```

Figure 6.1 Array Co- and Contravariance Problem in Java

6.3.3 Concurrent Aspect-oriented Programming

My study focused on sequential aspect-oriented programs. This focus excludes some interesting techniques. For example, Laddad's worker object creation pattern uses proceed closures [86, §8.1]. In this pattern, advice captures a proceed expression inside an instance of an anonymous Runnable class. This allows the advised code to be postponed, or executed immediately but in a new thread. Such use of proceed is fascinating, but to study it I would need a formalism that models concurrent processes. It may be that some variant of the π-calculus would be appropriate for this study [115].

6.3.4 Subtype Matching in Around? Unsound!

It is interesting to note that the semantics of $matchPCD_S$ for surround advice, described in Section 5.1.2.3, corresponds to that used by AspectJ for *all* advice. Before and after advice in AspectJ share many of the properties of surround advice, and I conjecture that AspectJ's matching semantics is statically type safe for before and after advice. But as discussed in Section 3.2.2.4, using a semantics like $matchPCD_S$ for around advice is not statically type safe.

AspectJ was designed for expressiveness and to be Java-like, in order that it might be readily adopted by Java programmers. It is common for Java programs, especially prior to Java 5 [65], to include many type casts. Prior versions of Java did not include parametric polymorphism. Container classes in these versions of Java treated all contained objects as having type Object. Downcasts had to be used to perform any interesting operations on an object extracted from such a collection. Java also has a co- and contravariance problem for arrays [13, §6.4.4] that is similar to the one for arguments in around advice in AspectJ. (See sample code in Figure 6.1.) Because of these static type safety issues in Java, AspectJ's designers did not treat static type safety as their highest goal.[3] With the array co- and contravariance problem in Java, there is no way to avoid the exception. This differs from the situation in AspectJ where the compiler automatically inserts a cast. Instead of inserting the cast, the AspectJ compiler could issue an error and require that the programmer manually insert the appropriate typecase and cast to suppress the compiler warning, and avoid any undesired run-time type errors, based on her knowledge of the desired semantics.

[3] Personal communication, Erik Hilsdale, June 17, 2005.

On the other hand, because Java 5 has dramatically reduced the need for type casts, it is interesting to consider how around advice in an AspectJ-like language might be made statically type safe without giving up expressiveness. Some possible alternatives are:

Exact matching. This is the MiniMAO$_1$ solution for soundness of its static type system, though it means that before, after, and surround advice use a different join point matching semantics. This is likely to be confusing to programmers. Would exact matching on interface types solve the expressiveness problem? I conjecture that it would but have not thought deeply about this. Even so, this would still leave the problem of having different semantics for pointcut matching in different kinds of advice. This could be mitigated by introducing new pointcut descriptors, like exactCall, that use exact type matching. Around advice would only be statically type safe if it used exact-matching pointcut descriptors.

Whole-program analysis. Rather than requiring a program-inserted type cast, a whole-program analysis could be performed to see if any unsafe uses appear. With concern maps the scope of this analysis could be restricted. However, once we admit dynamic weaving of aspects, then any analysis based on visible, unsafe uses would no longer be safe. Perhaps this could be avoided by only allowing dynamic weaving of before, after, and surround advice.

Final parameters. It might seem that the co- and contravariance problem for around advice could be avoided by making all pointcut-bound parameters final. However, this does not prevent swapping parameter order or substituting local state of the aspect for parameters, so it is a non-solution.

Revised proceed. Another possibility would be to make proceed in around advice behave something like the automatic proceed in surround advice. It could be restricted to not allowing changing of the values passed to the advised code in reference type arguments. It could still allow mutation of those arguments, zero or more proceeds, and mutation of results. This avoids the co- and contravariance problem because the arguments would point to the same objects as in the original call, thus preserving type invariance, while still allowing more flexibility than can be allowed for surround advice.

More expressive types. Would a more expressive type system for matching pointcut descriptors, perhaps using bounded parametric types [26, 136, 145] or dependent types [125, 161], allow less restrictive pointcut typing and matching while maintaining soundness? It is appealing to consider these more sophisticated type systems. On the other hand, would such a type system be accessible for the typical programmer?

6.3.5 Component-based Programming

It would also be interesting to compare the reasoning problem in aspect-oriented programming to reasoning in component-based programming [63, 152].

Component-based programming requires components to specify their expectations of external modules. This allows separate verification of the components, provided that expectations are checked at composition time. In the MAO discipline, the steps are performed in a different order. Each piece of advice is separately verified. Then, for a given composition, the specifications are composed to determine the effective specification of an advised join point. This effective specification is used to verify the code that triggers the join point. A problem that would surface as an unsatisfied expectation in component-based programming, appears as an effective specification like requires false; ensures true;. It is essentially useless for verifying the client code.

It seems that the essential trade-off is that component-based programming provides simpler compositional reasoning, but less expressive composition mechanisms versus aspect-oriented programming. Aspectual

collaborations [102] can be considered a mid-point between these extremes, though that system does not include formal specifications.

6.4 Postscript

My work on MultiJava [38, 43, 46] initially attracted me to the problems explored here. Open classes in MultiJava preserve static type safety, while the similar introductions of AspectJ do not. I was troubled by this mismatch. At the time it seemed incongruous that an extension to a statically type safe language would not try to maintain this property. In retrospect, Java is not so statically type safe as I thought. Though with parametric polymorphism, Java 5 comes closer to this ideal (especially if one eschews arrays of reference types for generic ArrayLists).

But still, I was and am bothered that a language that claims to improve modularity properties actually requires a whole program analysis for type checking and compilation.

My contention was that if some formal notion of modular reasoning was not to be found for AspectJ, then it was simply a programming language for specifying global mutations over the syntax of a program. Aspects instruct the compiler to edit code that matches certain quantified predicates. For dynamic-context pointcut descriptors, the compiler is instructed to insert branching instructions also.

However, having demonstrated that modular reasoning is, in fact, possible in AspectJ, I am now convinced that aspect-oriented programming really is a new paradigm for separating cross-cutting concerns. I am still troubled by the lack of static type safety and other properties of AspectJ. However, I believe that these problems can be remedied in future aspect-oriented languages and I look forward to making contributions in that area.

Is aspect-oriented programming as radical a departure from the past as object-oriented programming was? I do not think so. Aspect-oriented programming is more evolutionary in nature. While object-oriented programming really requires an entirely different approach to conceptualizing a problem domain, aspect-oriented programming requires more of an augmentation of the approach. Aspects serve as abstractions to separate the code for cross-cutting concerns, and as glue to connect concerns together. However, I suspect that within those concerns, regular object-oriented designs will prevail.

I am left with one main question: Does the complexity in reasoning about aspect-oriented programs make the paradigm inaccessible to the typical programmer? It may be that the weight of the machinery required for modular, aspect-oriented reasoning is just as heavy as the code needed to modularize cross-cutting concerns in an object-oriented program. If so, then perhaps something like spectators, or the harmless advice of Dantas and Walker [48], is sufficient, and everything else can be done with plain old objects. Only time may provide an answer, and given the vagaries of technological evolution, we may never know. Still, I am sure that it will be an interesting journey.

BIBLIOGRAPHY

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

[2] FOOL-12. *The 12th international workshop on Foundations of object-oriented languages*, Long Beach, California, 2005. ACM.

[3] ICSE '05. *Proc. of the 27th international conference on Software engineering*, St. Louis, Missouri, USA, 2005. ACM.

[4] OOPSLA '02. *Proc. of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seattle, USA, 2002. ACM.

[5] POPL '05. *Proc. of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Long Beach, California, USA, 2005. ACM.

[6] AOSD '03. *Proc. of the 2nd international conference on Aspect-oriented software development*, Boston, Mass., USA, 2003. ACM.

[7] AOSD '04. *Proc. of the 3rd international conference on Aspect-oriented software development*, Lancaster, UK, 2004. ACM.

[8] 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.

[9] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Odersky [124], pages 1–25.

[10] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In OOPSLA '02 [4], pages 311–330.

[11] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242. Springer-Verlag, 1987. Lecture Notes in Computer Science, volume 276.

[12] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489, pages 60–90. Springer-Verlag, 1991.

[13] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, Reading, MA, 2000.

[14] AspectJ Team. The AspectJ programming guide. Available from http://eclipse.org/aspectj on March 1, 2003.

[15] Enis Avdičaušević, Marjan Mernik, Mitja Lenič, and Viljem Žumer. Experimental aspect-oriented language - AspectCOOL. In *Proc. of the 2002 ACM Symposium on Applied Computing*, Madrid, Spain, 2002. ACM.

[16] Brian E. Aydemier, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, Lecture Notes in Computer Science, Oxford, UK, June 2005. Springer-Verlag. To appear.

[17] Lodewijk Bergmans and Mehmet Akşits. Composing crosscutting concerns using composition filters. *Comm. of the ACM*, 44(10):51–57, 2001.

[18] A. Michael Berman, editor. *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Denver, Colorado, USA, 1999. ACM Press.

[19] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In AOSD '04 [7], pages 83–92.

[20] Ron Bodkin, Don Almaer, and Ramnivas Laddad. aTrack: an enterprise bug tracking system using AOP. A demonstration at AOSD 2004, available from https://atrack.dev.java.net/ on July 17, 2005, March 2004.

[21] Jonas Bonér and Alexandre Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org/index.html, valid July 17, 2005.

[22] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[23] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In POPL '05 [5].

[24] L. Bougé and N. Francez. A compositional approach to superimposition. In *Proc. of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 240–249, San Diego, California, USA, 1988. ACM Press.

[25] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New Orleans, Louisiana, USA, 2003. ACM Press.

[26] K. B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4(2):127–206, April 1994.

[27] 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.

[28] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[29] Bill Burke, Austin Chau, Marc Fleury, Adrian Brock, Andy Godwin, and Harald Gliebe. JBoss aspect oriented programming. http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop, valid March 1, 2004.

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

[31] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. on Prog. Lang. and Systems*, 17(3):431–447, 1995.

[32] Yoonsik Cheon. *A Runtime Assertion Checker for the Java Modeling Language*. PhD thesis, Iowa State University, Ames, Iowa, USA, 2003. Available as ISU/CS TR 03-09, from archives.cs.iastate.edu.

[33] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[34] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An open AspectJ-like language. In AOSD '04 [7], pages 101–111.

[35] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In OOPSLA '02 [4], pages 292–310.

[36] Siobhán Clarke and Robert J. Walker. Composition patterns: an approach to designing reusable aspects. In *Proceedings of the 23rd international conference on Software engineering*, pages 5–14. IEEE Computer Society, 2001. ISBN 0-7695-1050-7.

[37] Siobhán Clarke and Robert J. Walker. Towards a standard design language for AOSD. In Kiczales [79], pages 113–119.

[38] Curtis Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Master's thesis, Iowa State University, Ames, Iowa, USA, 2001.

[39] Curtis Clifton and Gary T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, pages 33–44, Enschede, the Netherlands, 2002. Iowa State University, Dept. of Computer Science.

[40] Curtis Clifton and Gary T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, 2002.

[41] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report 03-15, Iowa State University, Department of Computer Science, 2003.

[42] 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.

[43] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 35(10), pages 130–145, New York, 2000.

[44] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Formal definition of the parameterized aspect calculus. Technical Report 03-12b, Iowa State University, Department of Computer Science, 2003.

[45] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Iowa State University, Department of Computer Science, 2003.

[46] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *Trans. on Prog. Lang. and Sys.*, 2005. To appear, preliminary version available from ftp://ftp.cs.iastate.edu/pub/techreports/TR04-01/TR.pdf on July 17, 2005.

[47] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–98, Vienna, Austria, 2001. ACM.

[48] Daniel S. Dantas and David Walker. Harmless advice. In FOOL-12 [2].

[49] Á. Darvas and P. Müller. Reasoning About Method Calls in JML Specifications. Submitted to FTfJP 2005, available from http://sct.inf.ethz.ch/publications/getpdf.php?bibname=Own&id=DarvasMueller.pdf on July 17, 2005.

[50] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[51] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[52] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 2005.

[53] 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.

[54] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behavior of aspectj programs. In *Proc. of the 19th annual ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, pages 150–169, Vancouver, BC, Canada, 2004. ACM.

[55] Matthew Dwyer, editor. *Proc. of the 12th ACM SIGSOFT symposium on the Foundations of software engineering (FSE-12)*, Newport Beach, California, USA, 2004. ACM Press.

[56] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, Lecture Notes in Comp. Sci., pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag.

[57] Erik Ernst and David H. Lorenz. Aspects and polymorphism in aspectj. In AOSD '03 [6], pages 150–157.

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

[59] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[60] Robert E. Filman, Stuart Barrett, Diana D. Lee, and Ted Linden. Inserting ilities by controlling communications. *Comm. of the ACM*, 45(1):116–122, 2002.

[61] 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.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[63] David S. Gibson, Bruce W. Weide, Scott M. Pike, and Stephen H. Edwards. Toward a normative theory for component-based system design. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 10, pages 211–230. Cambridge University Press, 2000. ISBN 0-521-77164-1.

[64] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[65] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[66] Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. An approach for supporting aspect-oriented domain modeling. In *Proceedings of the second international conference on Generative programming and component engineering*, pages 151–168, Erfurt, Germany, 2003. Springer-Verlag.

[67] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In AOSD '03 [6], pages 60–69.

[68] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428. ACM Press, Washington, D.C., USA, 1993.

[69] Erik Hatcher and Steve Loughran. *Java Development with Ant*. Manning, Greenwich, CT, 2003.

[70] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–583, Oct. 1969.

[71] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[72] IBM. Concern manipulation environment. Available from http://www.research.ibm.com/cme/papers/handout.html on July 17, 2005, March 2003.

[73] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Berman [18], pages 132–146.

[74] 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.

[75] Radha Jagadeesan, Alan Jeffrey, and James Riely. A calculus of untyped aspect-oriented programs. In Cardelli [30], pages 54–73.

[76] Shmuel Katz and Yossi Gil. Aspects and superimpositions. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, Lisbon, June 1999. Available from http://trese.cs.utwente.nl/aop-ecoop99/papers/katz.pdf on July 17, 2005.

[77] Mik A. Kersten and Gale C. Murphy. Atlas: A case-study in building a web-based learning environment using aspect-oriented programming. In Berman [18], pages 340–352.

[78] Gregor Kiczales. The fun has just begun. AOSD'03 Keynote Address, available from http://www.cs.ubc.ca/~gregor on July 17, 2005, March 2003.

[79] Gregor Kiczales, editor. *Proc. of the 1st international conference on Aspect-oriented software development*, Enschede, the Netherlands, 2002. ACM.

[80] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In ICSE '05 [3], pages 49–58.

[81] 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.

[82] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Comm. of the ACM*, 44(10):59–65, October 2001.

[83] 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.

[84] Ivan Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.

[85] Shriram Krishnamurthi, Kathleen Fisler, and Micahel Greenberg. Verifying aspect advice modularly. In Dwyer [55], pages 137–146.

[86] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[87] Donal Lafferty and Vinny Cahill. Language-independent aspect-oriented programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, Anaheim, California, USA, 2003. ACM Press.

[88] Patrick Lam and Martin Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In Cardelli [30], pages 273–302.

[89] Ralf Lämmel. A semantical approach to method-call interception. In Kiczales [79], pages 41–55.

[90] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Comp. Sci.*, pages 1087–1106. Springer-Verlag, 1999.

[91] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[92] Gary T. Leavens and William E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 212–223, Ottawa, Canada, 1990. ACM Press.

[93] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06y, Iowa State University, Department of Computer Science, 2004.

[94] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, and Joseph Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from www.jmlspecs.org on July 17, 2005, December 2004.

[95] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, 2005.

[96] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[97] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proc. of the 13th annual ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, pages 144–153, Vancouver, BC, Canada, 1998. ACM.

[98] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Odersky [124], pages 491–516.

[99] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Trans. on Prog. Lang. and Systems*, 24(5):491–553, September 2002.

[100] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 246–257, Berlin, Germany, 2002. ACM Press.

[101] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Comm. of the ACM*, 44(10):39–41, 2001.

[102] Karl Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, 2003.

[103] Barbara Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23(5):17–34, May 1988. Revised version of the keynote address given at OOPSLA '87.

[104] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, Cambridge, Mass., 1986.

[105] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *Trans. on Prog. Lang. and Sys.*, 16 (6):1811–1841, 1994.

[106] Vincent Massol and Timothy M. O'Brien. *Maven: A Developer's Notebook*. O'Reilly, Sebastopol, CA, 2005.

[107] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Programming Languages and Systems: First Asian Symposium, APLAS 2003*, volume 2895 of *Lecture Notes in Comp. Sci.*, pages 105–121, Taipei, Taiwan, November 2003. Springer-Verlag.

[108] Hidehiko Masuhara and Gregar Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In Cardelli [30], pages 2–28.

[109] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[110] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[111] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.

[112] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In OOPSLA '02 [4], pages 52–67.

[113] Mira Mezini and Klaus Ostermann. Variability management with function oriented programming and aspects. In Dwyer [55], pages 137–146.

[114] Todd Millstein and Craig Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, 2002.

[115] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[116] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262. Springer-Verlag, 2002. The author's Ph.D. Thesis.

[117] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[118] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency, Computation Practice and Experience.*, 15:117–154, 2003.

[119] Andrew Myers and Barbara Liskov. JFlow: Practical mostly-static information flow control. In *Proc. of the 26th ACM symposium on Principles of programming languages*, pages 226–241, San Diego, California, USA, 1998. ACM Press.

[120] Nanning Project. Nanning Java aspects AOP framework. http://nanning.codehaus.org/project-info.html, valid July 17, 2005.

[121] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP '98 — Object-Oriented Programming 12th European Conference*, volume 1445 of *Lecture Notes in Comp. Sci.*, pages 158–185, Brussels, Belgium, 1998. Springer-Verlag.

[122] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proceedings of CC 2003: 12'th International Conference on Compiler Construction*. Springer-Verlag, 2003.

[123] ObjectWeb Consortium. Java aspect components. http://jac.objectweb.org/, valid July 17, 2005.

[124] Martin Odersky, editor. *ECOOP '04 - Object-Oriented Programming European Conference*, volume 3086 of *Lecture Notes in Comp. Sci.*, Oslo, Norway, 2004. Springer-Verlag.

[125] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Cardelli [30], pages 201–224.

[126] Peter O'Hearn and David Pym. The logic of bunched implication. *Bulletin of Symbolic Logic*, 5(2): 215–244, 1999.

[127] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[128] Doug Orleans. Incremental programming with extensible decisions. In Kiczales [79], pages 56–64.

[129] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Comm. of the ACM*, 44(10):43–50, 2001.

[130] Johan Ovlinger. *Modular Programming with Aspectual Collaborations.* PhD thesis, Northeastern University, 2003.

[131] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In POPL '05 [5].

[132] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM*, 15 (12):1053–1058, 1972.

[133] D. L. Parnas. Software engineering or methods for the multi-person construction of multi-version programs. In Clemens E. Hackl, editor, *Programming Methodology, 4th Informatik Symposium, IBM Germany, Wildbad, 25-27 September, 1974*, volume 23, pages 225–235. Springer-Verlag, 1975.

[134] Frank Pfenning and Carsten Schürmann. System description: Twelf—a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proc. of the 16th Intl. Conf. on Automated Deduction (CADE-16)*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, Berlin, Heidelberg, New York, July 1999. Springer-Verlag.

[135] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Softw. Pract. Exper.*, 33(10):957–974, 2003.

[136] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, April 1994. A preliminary version appeared in POPL 1993.

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

[138] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-in-time aspects: efficient dynamic weaving for Java. In AOSD '03 [6], pages 100–109.

[139] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. on Prog. Lang. and Systems*, 25(1):117–158, Jan 2003.

[140] Arun D. Raghavan and Gary T. Leavens. Desugaring JML method specifications. Technical Report 00-03e, Iowa State University, Department of Computer Science, May 2005.

[141] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In ICSE '05 [3], pages 59–68.

[142] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In AOSD '03 [6], pages 120–129.

[143] John Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford-Microsoft Symposium in Honor of Sir Tony Hoare*. Palgrave, 2000.

[144] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2002.

[145] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Proc. IFIP Congress '83, Paris*, September 1983.

[146] Martin Rinard, Alexandur Sălcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In Dwyer [55], pages 147–158.

[147] Daniel Sabbah. Aspects—from promise to reality. Keynote address at AOSD 2004, available from http://aosd.net/2004/archive/AOSD-FromPromiseToReality.ppt on July 17, 2005, March 2004.

[148] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5), 2003.

[149] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60. Australian Computer Society, Inc., 2002. ISBN 0-909925-88-7.

[150] Stanley M. Sutton, Jr and Isabelle Rouvellou. Modeling of software concerns in Cosmos. In Kiczales [79].

[151] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In AOSD '03 [6], pages 21–29.

[152] Clemens Szyperski, Domiinik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, second edition, 2002.

[153] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. of the 21th international conference on Software engineering*, pages 107–119, Los Angeles, California, USA, 1999. ACM Press.

[154] David B. Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In AOSD '03 [6], pages 158–167.

[155] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, London, 2000. Springer-Verlag.

[156] 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.

[157] 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.

[158] Alan Wills. Capsules and types in Fresco: Program validation in Smalltalk. In P. America, editor, *ECOOP '91 — Object-Oriented Programming 5th European Conference*, volume 512 of *Lecture Notes in Comp. Sci.*, pages 59–76, Geneva, Switzerland, 1991. Springer-Verlag.

[159] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Trans. on Prog. Lang. and Systems*, 9(1):1–24, January 1987.

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

[161] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In FOOL-12 [2].

[162] Jianjun Zhao and Martin C. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Proc. Fundamental Approaches to Software Engineering*, volume 2621 of *Lecture Notes in Comp. Sci.*, Warsaw, Poland, 2003. Springer-Verlag.