MAO: Ownership and Effects for More Effective Reasoning About Aspects

Curtis Clifton[1], Gary T. Leavens[2], and James Noble[3,4]

[1] Rose-Hulman Institute of Technology, Terre Haute, Indiana, USA
[2] Iowa State University, Ames, Iowa, USA
[3] Imperial College, London, UK
[4] Permanent: Victoria University of Wellington, New Zealand

Abstract. Aspect-oriented advice increases the number of places one must consider during reasoning, since advice may affect all method calls and field accesses. MAO, a new variant of AspectJ, demonstrates how to simplify reasoning by allowing programmers, if they choose, to declare limits on the control and heap effects of advice. Heap effects, such as assignment to object fields, are specified using *concern domains*—declared partitions of the heap. By declaring the concern domains affected by methods and advice, programmers can separate objects owned by the base program and by various aspects. When desired, programmers can also use such concern domain annotations to check that advice cannot interfere with the base program or with other aspects. Besides allowing programmers to declare how concerns interact in a program, concern domains also support a simple kind of semantic pointcut. These features make reasoning about control and heap effects easier.

1 Introduction

Serve the People![1]

Aspect-oriented software development [13] (and its conjugates such as subjectivity, generative programming, Model-Driven Architecture and so on) are changing the way programs are structured. Rather than a program being a hierarchy, with each module or class defined in one place, a program becomes a heterarchy, where multiple crosscutting aspects contribute to the definition of multiple components. Aspect-oriented designs can help increase cohesion by reducing code scattering and tangling. This can positively affect a system's maintainability; each crosscutting concern can be dealt with in a single module, making it much easier to change the policies that govern that concern.

In this paper we describe Modular Aspects with Ownership, MAO, a variant of AspectJ 5 that helps programmers state and enforce restrictions on control and heap effects. Control effects are caused by advice that perturbs the program's control flow. Heap effects are modifications to object fields. Giving programmers the ability to state and enforce restrictions on these effects allows more effective reasoning in MAO than

[1] Mao Tse-tung's quotes are from `http://art-bin.com/art/omaotoc.html`. Use of these quotations in no way indicates our approval of Mao or his actions.

E. Ernst (Ed.): ECOOP 2007, LNCS 4609, pp. 451–475, 2007.

is generally possible in aspect-oriented languages such as ApsectJ. By "reasoning" we mean both informal checks, including desk-checking of code, and formal proofs.

MAO makes the following contributions:

1. **Surround Advice.** We introduce **`surround`** and **`curbing`** advice annotations that allow programmers to declare that their advice makes no (or limited) changes to the advised control flow. Surround advice can be used in spectator aspects to ensure they do not perturb the control flow of the base program [7, 9].
2. **Concern Domains.** We use a shallow ownership type and effect system [2, 5] to identify explicitly the concerns that own each object or aspect in the program. Programmers and tools can inspect the domain declarations and so statically determine how an aspect will interact with objects, or if two aspects may potentially interfere.
3. **Writes Pointcut Designator.** We introduce a new semantic pointcut designator, **`writes`**, which uses the ownership and effect system to provide a robust declaration for advice that matches all join points that may modify a particular concern domain.
4. **Spectator Aspects.** We state precise conditions on **`spectator`** aspects [7, 9]. Spectator aspects write only their own concern domains and use only surround advice, ensuring that they cannot affect the observable behavior of any other aspect or class in the program.

MAO's design is supported by MiniMAO$_3$, a formal model of MAO. The full details of MiniMAO$_3$ are described within Clifton's dissertation [7], including details we omit, such as a proof of type soundness and an ownership invariant for concern domains.

The paper proceeds as follows: The next section briefly presents the problem. Then, we informally introduce our solution with three sections describing the design of MAO. We give a high-level overview of our formal results and discuss a practical evaluation of our work. Finally, we conclude with a comparison to related work.

2 A Tale of Two Aspects

New things always have to experience difficulties and setbacks as they grow.

The key problem this paper addresses is reasoning about whether one module (class, method, aspect, advice) may potentially affect the behavior of another module. This is especially interesting for aspect-oriented programs, since interference among aspects and between aspects and other code can be quite subtle. Consider the venerable asteroids game [3]. The positions and vectors of a spaceship and some asteroids are managed by an N-body simulation — the spaceship can be influenced by player input. A `Model` class runs the simulation and stores spaceships, asteroids, missiles and so on.

Adding a user interface to this game is done with the `OutputWindow` aspect in Fig. 1. This aspect's advice runs after the simulation updates, when it reads data from the model, and then updates its output window. Reasoning about this aspect requires some assumptions that are not explicit in its code.

First, suppose we want to find control effects of the `OutputWindow` aspect. A *control effect* is a perturbation of the program's flow of control, such as throwing an exception, or stopping the call of a method. Since this aspect uses **`after`** advice, it does not seem to have any control effects. But that reasoning is not sufficient — we also have to

```
aspect OutputWindow {
  private SpacewarWindow w = new SpacewarWindow();

  after(Model m): target(m) && (call(void Model.set*())
                        || call(void Model.moveShip())
                        || call(void Model.updateAsteroids())) {
    w.reset();
    Spaceship s = m.getSpaceship();
    w.drawSpaceship(s.getX(), s.getY(), s.getHeading());
    for (Asteroid a : m.getAsteroids()) {
      w.drawAsteroid(a.x, a.y, a.size); }
    w.update();
}}
```

Fig. 1. The `OutputWindow` aspect

determine that the advice will not throw an exception that may affect the continuation of the program after the advice returns. This requires determining what exceptions can be thrown by the methods called in the advice, which are not explicit if the code calls methods that can throw unchecked exceptions.

Second, suppose we want to find the heap effects of the `OutputWindow` aspect. A *heap effect* is an assignment to some object fields.[2] The advice has no direct assignments to object fields, but we must also determine the potential side effects of the methods it calls. Methods like `reset` presumably have heap effects, and methods like `getSpaceship` presumably do not, though in practice we would need to verify that. Once we determine what method calls may have side effects the question becomes, what objects are affected? It matters if the object affected is owned by the advice, such as the window `w`, or not. In this case only `w` seems to be affected, but determining the heap effects of methods is not obvious from the code.

Finally, to determine when the advice will execute, we must understand its pointcut. The pointcut specifies when the display is to be updated. It does this by matching methods that may change the state of the model. This is quite a large design-level dependency on the program — we assume that the execution of any setter methods, plus a couple of specific methods on the `Model` class (such as `moveShip` or `updateAsteroids`) capture all effects on the model that need to be reflected. The problems with explicit naming and syntactic patterns are well known [21, 30]. The core issue here is that the pointcut specification is at the wrong level of abstraction. This advice should not match "*all calls where the first three characters of the method name are '*`set`*', or where the method is named* `moveShip` *or* `updateAsteroids`", instead what we need to express is: "*all calls to methods that may change the* `Model`." Such a *heap effect dependency* cannot be expressed directly in AspectJ. While it could be expressed with XPIs [15, 31], the XPI mechanism for expressing this is again an AspectJ pointcut, and does not provide a way of checking that the methods in the pointcut accurately express the dependency.

We can compare the benign `OutputWindow` aspect with the `Cheat` aspect in Fig. 2 on the following page. This aspect aims to override the collision detection function in the

[2] Heap effects implicitly include I/O, since object fields are used to represent I/O devices.

program, so that when the player's Spaceship hits something the collision is ignored and the ship's shields are activated, rather than the ship being destroyed and the player losing! Compared with the `OutputWindow` aspect there are three main differences. First, the aspect certainly has control effects: the around advice may return "false" rather than calling proceed. Second, by looking at the code of `raiseShields` we could determine that the advice also has heap effects on objects in the base program. On the other hand, the pointcut in this aspect — which matches calls to the collision method — is not expressing a heap effect dependency: it simply picks out a single method's execution.

```
aspect Cheat {
  boolean around(Model m, Thing one, Thing two) :
     call(boolean Model.collision(Thing,Thing)) && target(m) && args(one,two) {
  if ((one == m.getSpaceship()) || (two == m.getSpaceship())) {
       m.getSpaceship().raiseShields();
       return false;
  } else { return proceed(m, one, two); }
}}
```

Fig. 2. The `Cheat` aspect

These two aspects illustrate the three problems we address in this paper:

1. How can programmers find the control effects of advice?
2. How can programmers find the heap effects of advice?
3. How can programmers select join points according to their effects on the heap?

MAO provides solutions to each of these problems: control-limited advice mitigates control effects, concern domains describe heap effects, and effect pointcut designators select join points according to their effects on the heap. Compared with other work, MAO is designed as an extension to AspectJ, rather than as a more idealized AO language [11], and relies on types and annotations that can be checked locally, rather than global control and dataflow analyses [19, 29].

3 Control-Limited Advice

We cannot do without freedom, nor can we do without discipline.

The first problem we address is how to make finding the control effects of advice easier. All kinds of advice in AspectJ can cause control effects directly by throwing exceptions. (However, we do not consider errors, which inherit from `Error`, to be exceptions. Since errors indicate failures of the virtual machine, they are outside the scope of our analysis.) Around advice can also perturb control flow by not calling **`proceed`**, or by calling it several times. It is also convenient to consider changing the result returned by a computation (in around advice) to be a control effect. In AspectJ one can also change the target (receiver) object in a method call with around advice, which causes a control effect. The `Cheat` aspect in section 2 has two kinds of control effects, since it does not call **`proceed`** in some cases, and in those cases it supplies a new return value.

MAO allows programmers to declare that a piece of advice (or a whole aspect) has no control effects, or that those effects are limited to exceptional cases. We call such advice *control-limited advice*. MAO has two annotations for declaring that a piece of advice is control-limited: `@surround` and `@curbing`.

Advice marked with `@surround` has no control effects. When invoked in a particular state, `@surround` advice will proceed to the same join point, with the same arguments, and return the same value or throw the same exception, as it would in absence of the advice. (Note that this allows extra join points to be introduced, both within the advice and within the advised code.) For example, the advice in the `OutputWindow` (Fig. 1 on page 453) could have been declared using `@surround`, but not the `Cheat` aspect (of Fig. 2).

Advice whose only control effects are potentially to throw one or more exceptions that would not have been thrown otherwise can be marked with the `@curbing` annotation. Curbing advice can stop control flowing through a join point, but cannot augment it or change it in any other way. Curbing advice can be used, for example, to check authorizations or preconditions. The `@surround` and `@curbing` annotations can also be applied to entire aspects: thus requiring all their advice to be curbing or surrounding.

MAO uses simple desugarings and conservative criteria to modularly check that advice declared as control-limited actually is control-limited. These checks work differently for different kinds of advice.

For **`before`** and **`after`** advice annotated with `@surround`, MAO translates the advice body in such a way that all exceptions that might potentially be thrown out of their bodies are caught and discarded. The user does not have to write code to catch these exceptions, though she certainly may. But MAO automatically places the body inside a statement of the form "**`try`** */*body*/* **`catch`** `(Exception e) { ; }`", which discards all exceptions that might otherwise perturb the control flow.

Since **`around`** advice is inherently more powerful, it requires stronger checks. MAO checks that the advice has a body that is the sequential composition of a before part, a top-level call to **`proceed`**, and an after part that returns the result of the call to proceed (if any). No call to **`proceed`** may occur in either the before or the after part. MAO statically checks that surround advice always proceeds exactly once to the advised join point, unless the before part fails to terminate (e.g., loops forever). MAO automatically translates the before and after parts, as above, to automatically discard exceptions that occur in the code before and after the mandatory **`proceed`** call. Because the call to proceed is at the top-level, exceptions from the call to proceed cannot be caught; they must be propagated up the call stack as they would be in the absence of the advice. Fig. 3 on the following page gives an example satisfying the restrictions.

MAO checks that any arguments passed to the join point are always the original arguments and that the original arguments are declared to be **`final`** and `@readonly` (see Sec. 4.4), so surround advice cannot mutate the arguments and cannot pass along new arguments. The result returned from executing a piece of surround advice must be (if the return type is not **`void`**) saved in a final, `@readonly` variable named **`reply`**, and must be returned at the end of the after part. The after part expression has read-only access to **`reply`**. From these restrictions it follows that the before and after parts of surround advice are evaluated solely for their heap effects. Another way to think of such around

```
@surround Object around() : call( Object *(..) ) {
  // before part
  int event_no = Logger.nextEventNumber();
  this.log.append("before" + event_no);

  // mandatory proceed to advised code
  @readonly final Object reply = proceed();

  // after part
  this.log.append("after"  + event_no + "reply:" + reply);
  return reply;
}
```

Fig. 3. Example of `@surround` for **`around`** advice

advice is as paired before and after advice, where the before part can declare variables that the after part can access.

Checks on `@curbing` advice also differ depending on the advice type. Before or after advice declared to be `@curbing` is unchanged from AspectJ, since control effects that cause exceptions are permitted. Around advice that is declared to be `@curbing` must satisfy all of MAO's checks for `@surround` advice, but does not have exceptions that arise in the before and after parts automatically caught and discarded. In particular it must have the form illustrated in Fig. 3, so that it proceeds exactly once (unless the before part throws an exception).

Control-limited advice makes reasoning about aspect-oriented programs easier in three ways. First, by declaring a piece of advice as `@surround` or `@curbing`, programmers can express guaranteed limits on the control effects of their advice. MAO can ensure an advice's implementation matches the annotations on its declaration with an efficient, local analysis. The straightforward tests required — and the error messages MAO will produce if advice does not meet the conditions — should be easily comprehensible by programmers. Because the syntactic conditions can be checked locally, requiring only the code of the advice, changes to other parts of the program will not affect whether a particular piece of advice is surrounding or curbing.[3]

Second, because `@surround` and `@curbing` annotations are part of the interface of advice, programmers can immediately tell, by examining that interface, whether the advice does perturb the existing control flow. Thus when reasoning about control effects, one can simply ignore `@surround` advice. Furthermore one only has to look at `@curbing` advice in reasoning about exceptions; when reasoning about other kinds of control flow perturbation, one can also ignore `@curbing` advice.

Third, on a larger scale the search for control effects can be limited to non-spectator and non-surround aspects, since only such aspects may contain non-`@surround` advice. All of these represent some modest gains in effectiveness of reasoning.

[3] Of course, other advice that is not control-limited can cause control effects that occur at join points within control-limited advice. However, those control effects can be blamed on the advice that is not control-limited; that is, if all advice in a program is control-limited advice, then no control effects will affect the base program code.

4 Concern Domains

Qualitatively different contradictions
can only be resolved by qualitatively different methods.

To identify the heap effects of advice and ease reasoning about interference among and between aspect and base program code, MAO uses an ownership type-and-effect system we call *concern domains*. As with other ownership and confined type systems [2, 4, 12, 24, 27] concern domains require programmers to identify objects with a particular owner — in this case, a particular concern domain. For this reason, concern domains partition the program's heap, and so help answer the question of what object fields may be read and written by a piece of advice.

4.1 Declaring Concern Domains

Concern domains themselves are declared by classes or aspects. Unlike many ownership systems (but more like confined types [32]), concern domains are static: a particular system configuration will have a fixed set of concern domains. Following Generic Confinement and Generic Ownership, we reify domains using inert marker classes [27, 28].

Programmers explicitly declare concern domains by declaring an empty, **`final`** class that implements the interface `Domain`. Explicitly declared domains may be either publicly available, or may be private to a class or aspect [2]. To keep them static, however, they cannot be inner classes, although they can be static, nested classes.

Each concrete aspect implicitly defines a concern domain; that is the name of a concrete aspect can be used as a concern domain. This is appropriate because each such aspect is often associated with a concern in a well-designed program. Note that all instances of a particular non-singleton aspect, such as instances created per-cflow, all share a common concern domain. This sharing does not cause problems for MAO's effect analysis, though it does make it coarser.

MAO's concern domain `World` owns all objects not owned by other domains.

4.2 Using Concern Domains

Every object creation expression names the new object's owner, which may be a public concern domain, or a private one visible in the class (or aspect) instantiating the object. The ownership domain of every expression is statically tracked by the ownership type system, and objects owned by private domains are inaccessible outside the scope of those private domains (i.e. the class or package declaring the private domain).

The types of objects in the program can be annotated to describe the concern domains to which they belong. Classes and abstract aspects can be made ownership-parametric — in class and abstract aspect declarations a list of concern domain variables are given following the class or aspect name. The first concern domain variable listed, typically called `Owner`, represents the owner domain for instances of the class or aspect, that is, the domains to which they belong. Other concern domain variables allow referencing objects in other domains.

For example, the following code shows how the Model class could be defined with a domain parameter named `Owner`. The Spaceship and Vectors of Asteroids within Model are all stored in the same domain (`Owner`) as the Model object containing them.

```
class Model<Owner extends Domain> {  /* ... */
  private Spaceship<Owner> s = new Spaceship<Owner>();
  private Vector<Owner,Asteroids<Owner>> v =
      new Vector<Owner,Asteroids<Owner>>();
}
```

Then we declare a new concern domain MODEL and in it allocate a new Model instance.

```
final class MODEL implements Domain {}
static Model<MODEL> myModel = new Model<MODEL>();
```

Thanks to the field declarations in the Model class, a new Spaceship and Vector are also instantiated in the MODEL domain.

Instances of classes declared without domain annotations are owned by World.

4.3 Concern Domains and Aspects

As with classes, MAO aspects require ownership parameters to give them access to concern domains. Because MAO extends AspectJ 5, generic (and hence abstract) aspects cannot be instantiated directly, rather a concrete aspect extends the generic aspect while instantiating the generic aspect's type parameters. Thus, a concrete aspect cannot have ownership parameters (or any other type parameters), rather, the concrete aspect binds parameters of a generic aspect from which it inherits. Finally, so that aspects can have their own private data, each concrete aspect has its own domain, and the name of the concrete aspect is the name of that domain.

We rewrite the OutputWindow example (compare Fig. 1) in Fig. 4 on the next page, using the ownership type parameter Owner for the aspect's own concern domain, and parameter Other for the type of the exposed context m.[4] Note how the ownership types describe the concern domains to which each variable or argument must belong. For example the private field w belongs to the same concern domain as the aspect (its Owner) while the model m is in the Other domain. To use this aspect, we instantiate it by making a concrete aspect, binding the domain variables:

```
aspect ConcreteOutputWindow extends OutputWindow<ConcreteOutputWindow,MODEL>{}
```

the Owner variable is bound to the aspect's domain, and the Other variable to the MODEL domain. The Other domain could alternatively have been bound to any (public) domain in the program, including the default World domain, or a public domain belonging to another aspect (to support mutually-crosscutting aspects).

4.4 Effect Declarations

Concern domains and domain parameters separate expressions owned by different domains via their types: within the annotated OutputWindow aspect, we know which expressions belong to the aspect (concern domain Owner) and which to the base program (Other). To track aspect interference, we also need to determine when a method or advice execution may have a potential heap effect on the fields in a particular domain. For

[4] AspectJ 1.5.3 does not allow Other to be used as a type parameter in advice formals and PCDs. We leave compilation techniques that overcome this limitation as future work.

```
@readonlyDomains({"Other"})  @depends({ @varies({"Owner", "Other"}) })
abstract aspect OutputWindow<Owner extends Domain,
                             Other extends Domain> {
  private SpacewarWindow<Owner> w = new SpacewarWindow<Owner>();

  @curbing @writes({"Owner"})
  after(@readonly Model<Other> m):
          target(m) && call(void Model<Other>.*()) && writes(Other) {
    w.reset();
    Spaceship<Other> s = m.getSpaceship();
    w.drawSpaceship(s.getX(), s.getY(), s.getHeading());
    for (Asteroid<Other> a : m.getAsteroids()) {
      w.drawAsteroid(a.x, a.y, a.size); }
    w.update();
}}
```

Fig. 4. The `OutputWindow` aspect with annotations

this we augment the ownership type system with effects [5]. The basic effect annotations are `@writes`, which is attached to method and advice declarations, and `@readonly`, which is a type modifier.

The `@writes` annotation declares the concern domains that a particular method or piece of advice can potentially mutate. Due to limitations of Java 5 annotations, this annotation contains an array literal, with a comma-separated list of strings naming concern domains or parameters. For example, the advice in the above example is allowed to write into the `Owner` domain (which will be the `ConcreteOutputWindow` domain in that concrete subaspect). Since `Other` is not named by this `@writes` annotation, however, the advice is not allowed to write that domain—or `MODEL` in the concrete subaspect. (Due to the inherent differences in how one reasons about methods, which are explicitly called, and advice, which is triggered implicitly, we do not consider the effects of **proceed** as belonging to the effects of the advice. These effects do play a role in domain dependencies discussed below).

The `@readonly` annotation applies to types. Read-only fields and parameters cannot be written into, although they can be read from. For example, the model `m` in the above example is read-only, and hence the advice cannot mutate any field of an object reachable through that reference — so `@readonly` is transitive.

As a shorthand, a method (or piece of advice) can be annotated with `@pure`, meaning that all of its parameters (or variables in an advice's exposed context) are read-only. It is an error for a `@pure` method (or advice) to have a `@writes` annotation.

The programmer declares an ownership parameter of a class or aspect to be read-only using the `@readonlyDomains` annotation; this annotation is then effective wherever that parameter is used.[5] Such a read-only concern domain cannot be mentioned in a `@writes` annotation, making the whole concern domain read-only within the scope of that ownership parameter's declaration. If a method or advice has no `@writes` annotation, then by default it can write to any non-read-only concern domain in scope.

[5] Planned Java enhancements (JSR 308) will allow `@readonly` annotations on type parameters.

In Fig. 4, for example, the advice is annotated `@writes("Owner")` because it writes to the output window, which is allocated within the aspect's own concern domain. The important point is that the `Other` domain — representing the base program holding the model — is a read-only domain, to which the aspect cannot write. In this way, the effect annotations let programmers make their intentions clear and checkable. (The fact that the `Other` domain is read-only means that the `@readonly` annotation on `m`'s type is actually redundant in this example).

Programmers using MAO can state their intention in another way also. In Fig. 4, the `@depends` annotation on the aspect says that the `Owner` domain is allowed to vary whenever `Other` may. This *dependency declaration* allows the after advice in the example to mutate the `Owner` domain while advising methods that mutate the `Other` domain. This dependency is used both to check the `@writes` annotation on the advice and to reason about potential effects without considering the internal details of the advice.

The effect annotations illustrate a key benefit of MAO's ownership types: by inspecting only the aspects and their annotations, we can be sure that `OutputWindow` does not change any object owned by the base program. MAO's type system enforces a non-interference property so that a static, signature-level search can identify all the code that might mutate a particular concern domain. By "static" we mean that the search can be confined to areas of the program where the concern domain in question is visible, either because it is directly visible or because it was passed as a domain parameter. By "signature-level", we mean that only method and advice headers, and not their bodies, must be considered. In fact, if a programmer is just concerned about the effects of aspects on a method call, she can consider just the headers of aspects apart from their advice. Thus MAO statically identifies code tangling, based on a separation of concerns defined by the programmer.

Finally, we can combine concern domains with the control-limited advice from Sec. 3 to define *spectator aspects* [10]. A `@spectator` aspect contains advice that is (implicitly) `@surround`, all ownership parameters other than `Owner` are (implicitly) `@readonly`, and the concrete concern domain used to instantiate a spectator aspect cannot be shared. Thus, a spectator aspect concisely specifies advice that has no control effects, and whose heap effects are confined within the aspect's own concern domain. In Sec. 6, we formally show that spectator aspects do not cause heap interference. Since lack of control effects is direct from the definition of `@surround` advice, spectators do not affect the execution of the base program in any way.

4.5 Annotating Base Code

We consider that annotating *aspects* with ownership types and effects is a necessary price to pay for the tighter granularity of reasoning.

An important advantage of AOP, however, is that aspects can be attached to base code that is oblivious to the aspects, that is the base code should *not* need to be changed to have aspects applied to it. Thus it will often be infeasible to annotate preexisting base code. MAO's `World` domain, and the default that instances of unannotated classes are owned by this domain, offers a broad brush solution to this problem. Because all objects created in the base program are owned by `World`, the base program still type checks. Despite the coarseness of the `World` domain, MAO still has more than enough

information to separate base program objects from objects belonging to aspects, as the `OutputWindow` example above illustrates.

5 Effect Pointcut Designators

If you want knowledge, you must take part in the practice of changing reality.

To select join points according to their effects on the heap, MAO introduces a new kind of pointcut designator (PCD), **`writes`**. It allows programmers to use concern domain declarations to refine their aspect's pointcut definitions. We call this PCD an *effects* PCD, as it matches join points with heap effects on a given concern domain. Unlike AspectJ **`set`** and **`get`** PCDs, which describe heap effects and accesses syntactically (in terms of concrete field names or patterns), effects PCDs describe effects semantically (in terms of concern domains). Another difference is that they can work at the level of methods and advice, instead of just at the lower level of individual operations on fields.

A MAO PCD of the form **`writes`**(D) expresses heap effect dependencies by matching all join points that may write to concern domain D. MAO statically calculates the heap effect of field sets based on the owner domain of the field type. MAO also statically calculates the heap effect of a method or advice, either from an explicit `@writes` or `@pure` annotation, or from its default (see Sec. 4.4). Note that these PCDs do not describe method or advice call or execution join points that actually *do* write to a specific concern domain, but to those that may possibly write to that concern domain.

For example, MAO's **`writes`** PCD lets programmers express the heap effect dependency implicit in Fig. 1 on page 453. Instead of saying that the call should be to methods whose name matches a method pattern (`set*`) or one of the two named methods (`moveShip` or `updateAsteriods`), we can rewrite this PCD as in Fig. 4 on page 459, using the `Other` concern domain:

```
target(m) && call(void Model<Other>.*()) && writes(Other)
```

Given that the `set*`, `moveShip`, and `updateAsteriods` methods and so on are annotated with effects annotations, and that the `OutputWindow` aspect is instantiated with an appropriate domain binding (see Sec. 4.3), then this pointcut will match exactly the same methods as the previous explicit pointcut from Fig. 1. More importantly, if there are other methods that are declared as writing the `Other` concern domain, this pointcut will match those methods too. As the program evolves, if more methods are added that write that domain, this pointcut will stay valid. Because the effect PCDs are tied to the appropriate concern domain (`MODEL` which is bound to `Other`), these kind of pointcut designators are more closely tied to the program's semantics and can be automatically checked. Automatic checking ensures they are maintained when the program changes, unlike other, programmer-constructed pointcut abstractions, such as explicit advice points from open modules [1] or design rules [15].

The main disadvantage of effects pointcuts is that currently they can only apply to classes (and aspects) that have been annotated with concern domains and effect clauses. Defaulting base programs to a single `World` domain — while very effective for separating aspects from unmodified based programs — is almost completely ineffective here: we can assign all base program objects to a single domain only because that assignment

is so indiscriminate. We expect that using a confined-types-style analysis [16] to assign e.g., individual Java packages into their own domain, should make enough distinctions to be useful in many cases.

6 Formal Model: MiniMAO$_3$

When we look at a thing, we must examine its essence ...

MiniMAO$_3$ provides a formal model of MAO's design, building on Clifton's and Leaven's earlier formal model (MiniMAO$_1$) that described around advice [8]. Space constraints keep us from fully detailing MiniMAO$_3$ here, but we aim to illuminate the important issues. Clifton's dissertation [7] provides full details of MiniMAO$_3$.

6.1 The MiniMAO$_3$ Language

The object-oriented core of MiniMAO$_3$ is based on Featherweight Java (FJ) [18]. As such, a MiniMAO$_3$ program includes of a list of class declarations followed by an expression, which represents the `main` method in a Java program. Like FJ, class declarations in MiniMAO$_3$ contain a list of field declarations and a list of method declarations.

MiniMAO$_3$ departs from FJ in several ways to support our study of heap effects in aspect-oriented programs. As described in our earlier work on MiniMAO$_1$ [8], we use an imperative formal language with features from Classic Java [14]. Among other things, this choice admits **`null`** values for fields, so we omit constructors from the language. MiniMAO$_3$ includes concern domain annotations on types and class declarations. It also includes declaration forms that model MAO's aspects, spectator aspects, around and surround advice, domain dependencies, and ground concern domains.

Figure 5 gives the surface syntax of MiniMAO$_3$. A program consists of a series of declarations—of classes, regular aspects, and spectator aspects. These type declarations are followed by a list of public concern domain declarations, like **`domain`** `MODEL`. The public concern domains form the set of ground domains for the program and correspond to MAO's inert classes that implement the `Domain` interface. After the public concern domain declarations, a program gives a list of aspect instantiation statements, like **`use`** `OutputWindow`$\langle$**`self`**`,MODEL`$\rangle$, that model MAO's generic aspect instantiation. In our formalism, all classes and aspects are polymorphic with respect to concern domains. Class and aspect declarations give a list of concern domain variables, denoted by the metavariable G in Figure 5. These variables are instantiated with ground domains when the program is evaluated. The usual **`new`** expression instantiates a class; the new **`use`** statement instantiates an aspect.

As mentioned, class and aspect declarations include a list of concern domain variables, $\langle G^* \rangle$, following the class or aspect name. The first concern domain variable listed represents the home domain for instances of the class or aspect. The remaining variables are used to endow instances with permission to access objects in other domains, like concern domain parameters in MAO. For spectator aspects, we always write **`self`** as the first concern domain variable. This is a special variable that represents the private concern domain of the spectator. Each spectator instance has its own unique concern domain as in MAO. Only the spectator instance, and any objects it creates in **`self`**, may

$$
\begin{aligned}
P &::= \mathit{decl}^*\ \{\mathit{domain}^*\ \mathit{asp}^*\ e\}\\
\mathit{decl} &::= \texttt{class}\ c\langle G^*\rangle\ \texttt{extends}\ c\langle G^*\rangle\ \{\mathit{field}^*\ \mathit{meth}^*\}\\
&\mid \texttt{aspect}\ a\langle G^*\rangle\ \{\mathit{dep}^*\ \mathit{field}^*\ \mathit{adv}^*\}\\
&\mid \texttt{spectator}\ a\langle \texttt{self}, G^*\rangle\ \{\mathit{field}^*\ \mathit{surr}^*\}\\
\mathit{field} &::= t\ f\texttt{;}\\
\mathit{meth} &::= t\ m\texttt{(}\mathit{form}^*\texttt{)}\ \mathit{eff}\ \{e\}\\
\mathit{dep} &::= \gamma\ \texttt{varies with}\ \gamma\texttt{;}\\
\mathit{adv} &::= t\ \texttt{around(}\mathit{form}^*\texttt{)}\ \mathit{eff}\ \texttt{:}\ \mathit{pcd}\ \{e\}\\
\mathit{surr} &::= \texttt{surround (}\mathit{form}^*\texttt{)}\ \texttt{:}\ \mathit{pcd}\ \{e\texttt{; proceed; }e\}\\
\mathit{eff} &::= \texttt{writes}\ \langle\gamma^*\rangle\\
\mathit{pcd} &::= \texttt{call(}\mathit{pat}\texttt{)} \mid \texttt{execution(}\mathit{pat}\texttt{)} \mid \texttt{writes(}\gamma^*\texttt{)} \mid \texttt{args(}\mathit{form}^*\texttt{)}\\
&\mid \texttt{this(}\mathit{form}\texttt{)} \mid \texttt{target(}\mathit{form}\texttt{)} \mid \mathit{pcd}\ \texttt{\&\&}\ \mathit{pcd} \mid \texttt{!}\ \mathit{pcd} \mid \mathit{pcd}\ \texttt{||}\ \mathit{pcd}\\
\mathit{pat} &::= t\ \mathit{idPat}\texttt{(..)}\\
\mathit{form} &::= t\ \mathit{var}\text{, where } \mathit{var} \notin \{\texttt{this}, \texttt{reply}\}\\
e &::= \texttt{new}\ c\langle\gamma^*\rangle\texttt{()} \mid \mathit{var} \mid \texttt{null} \mid e\texttt{.}m\texttt{(}e^*\texttt{)} \mid e\texttt{.}f \mid e\texttt{.}f\ \texttt{=}\ e\\
&\mid \texttt{cast}\ t\ e \mid e\texttt{;}\ e \mid e\texttt{.proceed(}e^*\texttt{)}\\
t, s, u &::= \delta^*\ T\langle\gamma^*\rangle\\
\delta &::= \varepsilon \mid \texttt{readonly}\text{, where } \varepsilon \text{ represents the empty string}\\
T &::= c \mid a\\
\gamma &::= g \mid G \mid \texttt{self}\\
\mathit{domains} &::= \texttt{domain}\ g\texttt{;}\text{, where } g \notin \mathcal{G}_{\texttt{self}}\\
\mathit{asp} &::= \texttt{use}\ a\langle g^*\rangle\texttt{;} \mid \texttt{use}\ a\langle \texttt{self}, g^*\rangle\texttt{;}\text{, where } g \notin \mathcal{G}_{\texttt{self}}
\end{aligned}
$$

$$
\begin{aligned}
G &\in \mathcal{G}_{var}\text{, the set of concern domain variable names}\\
\mathcal{G}_{\texttt{self}} &= \{\texttt{self}_{loc} \cdot \mathit{loc} \in \mathcal{L}\}\text{, the set of private concern domain names}\\
g &\in \mathcal{G} \cup \mathcal{G}_{\texttt{self}}\text{, where } \mathcal{G} \text{ is the set of public concern domain names}\\
c, d, a, f, m &\in \mathcal{I}\text{, the set of identifiers}\\
\mathit{var} &\in \{\texttt{this}, \texttt{reply}\} \cup \mathcal{V}\text{, where } \mathcal{V} \text{ is the set of variable names}\\
\mathit{idPat} &\in \mathcal{IP}\text{, the set of identifier patterns}
\end{aligned}
$$

Fig. 5. Surface Syntax of MiniMAO$_3$

write to this private domain. Furthermore, the spectator and its progeny may *only* write to this domain. These restrictions are enforced by MiniMAO$_3$'s static type system.

Like MAO, regular aspects in MiniMAO$_3$ include dependency declarations. These declarations allow an aspect to declare that one concern domain may be modified when code is executed that might modify some other domain. This allows an aspect to modify its own accessible concern domains, like a concrete output window, when advising code that modifies another concern domain, like the model in our game example. We would indicate this like `ConcreteOutputWindow` **`varies with`** `MODEL`. Dependency declarations allow—thanks to the aspect instantiation instructions—a static analysis of what

$$
\begin{array}{rll}
e ::= \ldots \mid v \mid \texttt{(}l\ \texttt{(}e^*\texttt{))} \mid \langle e \rangle_{\delta,\hat{\gamma}} \mid e \curvearrowright e & & \\
\quad \mid \texttt{joinpt}\ j\texttt{(}e^*\texttt{)} \mid \texttt{chain}\ \bar{B}, j\texttt{(}e^*\texttt{)} \mid \texttt{under}\ e & & \\
v ::= loc_{\delta*} \mid \texttt{null}_{\delta*} & & \hat{\gamma} \in \mathcal{P}(\mathcal{G} \cup \mathcal{G}_{var} \cup \mathcal{G}_{\texttt{self}}) \\
l ::= \texttt{fun}\ m\langle var^* \rangle . e : \tau\ \texttt{.}\ \hat{\gamma} & & \hat{g} \in \mathcal{P}(\mathcal{G} \cup \mathcal{G}_{\texttt{self}}) \\
\tau ::= t \times \ldots \times t \rightarrow t & & \\
t, s, u ::= \ldots \mid \top & &
\end{array}
$$

Fig. 6. Syntax Extensions for the Operational Semantics of MiniMAO$_3$

domains might be modified by any operation. Spectator aspects do not include dependency declarations, because we assume a spectator's private concern domain can always vary. But since objects not owned by the spectator cannot observe the private concern domain, this mutability does not matter for reasoning.

The **writes** clause specifies all the concern domains that a method or advice declaration may modify. The type system ensures that only these domains, and those transitively reachable through dependency declarations, can be modified when the method or advice executes. These features ensure that the modifiable domains for any operation can be determined from a global "signature-level" analysis of the code, the bodies of methods and advice need not be considered. The bodies are checked through local rules.

Spectator aspects may only include surround advice. Surround advice differs from around advice by syntactically enforcing the restrictions described above for MAO's `@surround` advice. Note that in MiniMAO$_3$, the proceed that separates the before and after parts of surround advice is not an expression. It merely serves as a mnemonic for the semantics of surround advice, which is to evaluate the before part, proceed to the advised join point with the original arguments, evaluate the after part for its side-effects, then return the value from the advised join point. The after part may use the reserved variable reference **reply** to refer to the result of the advised code. Because of this semantics no return type is declared for surround advice. Furthermore, surround advice does not include a **writes** clause; every piece of surround advice implicitly writes **self** and no other concern domains.

Types in MiniMAO$_3$ have the form $\delta^*\ T\langle\gamma_1, \ldots, \gamma_q\rangle$, where δ is either **readonly** or the empty string, T is a valid class or aspect name, and γ ranges over concern domain variables and ground domains. Since **readonly** is idempotent, using δ^* for multiple such annotations lets us write **readonly** t to confer read-only status on any type t.

Other than including our new **writes** pointcut descriptor, the join point model for MiniMAO$_3$ is standard. Similarly, its expressions need no additional explanation.

6.2 Operational Semantics

Like most small-step operational semantics, that of MiniMAO$_3$ relies on some additional syntax to represent intermediate states of computation. Figure 6 presents these syntax extensions. We give the intuition behind these expressions here.

Two new expressions, loc_{δ^*} and $\texttt{null}_{\delta^*}$, represent values. The meta-variable v ranges over values. The store in MiniMAO$_3$, denoted by S, maps locations, $loc \in \mathcal{L}$, to objects. Values carry a subscript δ^* that denotes whether the reference is read-only.

The other new expressions represent intermediate computation states. To model method execution independently from method calls [8], we use a *function application expression*, $(l(e^*))$, that represents a method and its operands. The meta-variable l ranges over method representations. A context-sensitive translation converts method declarations to more convenient method representations. For example, the declaration

```
boolean collision(Thing one, Thing two) writes ⟨cache⟩ { false }
```

inside a class `Model` is represented by

```
fun collision ⟨one,two⟩.false :
    Model × Thing × Thing → boolean . {cache}.
```

Tagged expressions, written $\langle e \rangle_{\delta,\hat{\gamma}}$, propagate effect constraints through the semantics (necessary for the soundness proof). In a tagged expression, the set $\hat{\gamma}$ says which concern domains may be mutated during the evaluation of e, and the subscript δ gives the read-only status of any value that results from any (non-divergent) evaluation.

Leap expressions, written $e_1 \curvearrowright e_2$, represent intermediate evaluation of surround advice, with e_1 representing the advised code and e_2 the after part of the advice. 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.

As in MiniMAO$_1$ [8], **joinpt**, **chain**, and **under** expressions are used to represent the intermediate stages of advice matching, execution, and proceeding to advised code. A **joinpt** expression reifies a join point for advice binding. The meta-variable j ranges over join points and records a join point kind and optional data including things like the current **this** object, the method executing, and the writable concern domains in the current context. The operational semantics also maintains a *join point stack*, a list of join points that records dynamic context information needed for advice matching. The join point stack is a formal analogue of the call stack information that can be matched by AspectJ advice. A **chain** expression records the bodies of all advice matched at a join point. The meta-variable B ranges over advice body representations. We elide the details here, but it suffices to think of the advice body representations as like the method representations in that they record all necessary context-sensitive information about advice needed during evaluation. Finally, the operational semantics uses **under** expressions to pop join points from the join-point stack when evaluation of the code under the join point is complete.

The evaluation relation for MiniMAO$_3$ has the form: $\langle e, J, S \rangle \hookrightarrow \langle e', J', S' \rangle$. It takes an expression, a join point stack, and a store and produces a new expression or an exception, plus a new stack and a new store. The exceptional results, `NullPointerException` and `ClassCastException`, handle dereferencing null pointers and bad casts.

6.3 Static Semantics of MiniMAO$_3$

The static semantics of MiniMAO$_3$ checks the restrictions of the concern domains type system, read-only annotations, and effects clauses.

Like Featherweight Java [18], a global class table, denoted CT, records all the class declarations in a MiniMAO$_3$ program. MiniMAO$_3$ extends the class table to

record aspect declarations as well. Additionally, an evaluation dependency table, DT, records the information embedded in the "varies with" dependency declarations, reified according to the aspect instantiation instructions. A dependency table is a reflexive, transitive relation on concern domain names and variables. It has the type $(\mathcal{G} \cup \mathcal{G}_{var} \cup \mathcal{G}_{\texttt{self}}) \rightarrow (\mathcal{G} \cup \mathcal{G}_{var} \cup \mathcal{G}_{\texttt{self}})$. Intuitively, for any pair of concern domain names $(g, g') \in DT$, code that is allowed to mutate g may also trigger mutation of g'.

We use the notation $t \preccurlyeq s$ to denote that the type t is a subtype of the type s. The subtyping relationship starts with the reflexive and transitive closure induced by the `extends` declarations of classes, with every type a subtype of $\top$. To this we add a few additional tweaks. A couple of these handle read-only objects: $t \preccurlyeq \texttt{readonly}\ t$, allowing writable objects to be passed where read-only ones are expected, but not the converse; and $t \preccurlyeq s$ implies that $\texttt{readonly}\ t \preccurlyeq \texttt{readonly}\ s$, allowing a read-only object of a subtype to be passed where a read-only object of a supertype is expected, which is necessary for subsumption. The other tweak handles concern domains: following Aldrich and Chambers [2], a subtype must have at least as many concern domains as its supertype and the concern domains must be positionally invariant. For example, $\texttt{IterImpl}\langle\texttt{H,E,D}\rangle \preccurlyeq \texttt{Iterator}\langle\texttt{H,E}\rangle$.

The typing judgment for expressions in MiniMAO$_3$ has the form $\Gamma \centerdot \hat{\gamma} \vdash_{DT} e : t$. This says that, given the type environment Γ, the set of writable concern domain domains $\hat{\gamma}$, and the concern dependency table DT, we can derive that the expression e has type t.

For example, the typing rule for set expressions is:

T-SET

$$\frac{\begin{array}{c}\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_1 : T\langle\gamma_1, \ldots, \gamma_n\rangle \\ \gamma_1 \in \hat{\gamma} \qquad \mathit{fieldsOf}\,(T\langle\gamma_1, \ldots, \gamma_n\rangle)\,(f) = t \qquad \Gamma \centerdot \hat{\gamma} \vdash_{DT} e_2 : s \qquad s \preccurlyeq t\end{array}}{\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_1\texttt{.}f\ \texttt{=}\ e_2 : s}$$

This is mostly standard except for the hypothesis $\gamma_1 \in \hat{\gamma}$ that ensures that the domain containing the object to be mutated is in the set of writable concern domains.

As another example, the typing rule for method calls is:

T-CALL

$$\frac{\begin{array}{c}\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_0 : \delta\ T_0\langle\gamma_1, \ldots, \gamma_p\rangle \qquad \forall i \in \{1..n\} \cdot \Gamma \centerdot \hat{\gamma} \vdash_{DT} e_i : u_i \\ \mathit{methodType}(\delta\ T_0\langle\gamma_1, \ldots, \gamma_p\rangle, m) = t_1 \times \ldots \times t_n \rightarrow t \\ \mathit{writable}(\delta\ T_0\langle\gamma_1, \ldots, \gamma_p\rangle, m) = \hat{\gamma}' \\ \mathit{depClose}_{DT}\,(\hat{\gamma}') \subseteq \hat{\gamma} \qquad (\delta = \texttt{readonly}) \implies (\hat{\gamma}' = \emptyset) \qquad \forall i \in \{1..n\} \cdot u_i \preccurlyeq t_i\end{array}}{\Gamma \centerdot \hat{\gamma} \vdash_{DT} e_0\texttt{.}m\texttt{(}e_1, \ldots, e_n\texttt{)} : t}$$

Again, much of this is standard. Interestingly, if the read-only status δ of the receiver expression is in fact **`readonly`**, the second to last hypothesis ensures that no domains are writable in the body of the called method. The hypothesis $\mathit{depClose}_{DT}\,(\hat{\gamma}') \subseteq \hat{\gamma}$ ensures that the potentially writable domains in the body of the method form a subset of those writable in the context of the method call. The dependency closure function, $\mathit{depClose}_{DT}$, operates on a dependency table and a set of concern domains. It places a bound on the concern domains that might be modified by a given method call in the presence of a given set of advice.

$$depClose_{DT}(\hat{\gamma}) = \{\gamma' \cdot \exists \gamma \in \hat{\gamma} \cdot (\gamma, \gamma') \in DT\} \cup \{\texttt{self}_{loc} \cdot (\exists loc \in \mathcal{L} \cdot (\texttt{self}_{loc}, \texttt{self}_{loc}) \in DT)\}.$$

6.4 Meta-theory of MiniMAO$_3$

This section highlights the key theorems in the meta-theory of MiniMAO$_3$. These include static type safety and two theorems related to effects and the (un-)observability of mutations made by spectators.

Type safety is proved using the standard subject reduction and progress theorems.

In the meta-theory, the type environment maps variables and store locations to types. Additionally, the type environment records the ground concern domains, so that for a ground concern domain g, $\Gamma(g) = \texttt{domain}$. A type environment Γ is *concern complete* for a program P if every ground concern domain in P is in the domain of Γ.

A type environment Γ is *consistent with a store* S, written $\Gamma \approx S$, if all objects in the store conform to their types, both as declared and as given by Γ, and if the sets of locations in the domains of both Γ and S are the same. A *valid store* for a program P contains objects (with the appropriate concrete concern domains) representing every aspect instantiated in P. Additionally, for a store to be valid there must exist some type environment consistent with it. Similarly, a join point stack J is *consistent with a store* S, written $J \approx S$, if all locations named in J appear in S's domain.

The Subject Reduction theorem says that, given a configuration that meets appropriate initial conditions including having a well-typed expression, single-step evaluation results in a new configuration that satisfies the same conditions and that has an expression that is a subtype of the original expression.

Theorem 1 (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 $\Gamma' \centerdot \hat{\gamma} \vdash_{DT} e' : t'$.*

The Progress theorem says that, given a configuration that meets these same conditions, the expression is either a value or can be evaluated in a single step to a configuration giving a new expression or an exception.

Theorem 2 (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 $\hat{g}' \subseteq \hat{\gamma} \subseteq (\hat{g} \cup \hat{g}')$, 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 $loc \in dom(S)$,*
- *$e = \texttt{null}_\delta$ for some δ, or*
- *one of the following hold:*

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

The Type Safety theorem says that a well-typed program either diverges or evaluates to a value or exception.

Theorem 3 (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* $loc \in dom(S)$,
- $x = \texttt{null}_\delta$ *for some* δ,
- $x = \texttt{NullPointerException}$, *or*
- $x = \texttt{ClassCastException}$

Besides static type safety, MiniMAO$_3$ provably enforces the constraints given by effects clauses and concern domain dependency declarations. The central theorem here is called Tag Frame Soundness. It states that for any concern domain g that is not directly or transitively declared to be mutable for a given expression e, the portion of the store corresponding to g will be unchanged when e is evaluated to a value.

The formal statement of this theorem demands just a bit more terminology. Like Classic Java, we use evaluation context rules, denoted by $\mathbb{E}$ to implicitly define the congruence rules and give a non-constructive definition of evaluation order [14]. The rules are completely standard and are omitted here. To refer to the portion of the store S corresponding to a particular ground concern domain g, we write $S|g$, which is the set of all mappings in the store where the owner domain of the mapped object is g.

Theorem 4 (Tag Frame Soundness). *Let P be a well-typed program with concern domains $\hat{g}$ and evaluation dependency table DT. If the configuration $\langle \mathbb{E}[\langle e\rangle_{\delta,\hat{\gamma}}], J, S\rangle$ 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 $\forall g \in (\hat{g} \setminus depClose_{DT}(\hat{\gamma})) \cdot S|g = S'|g$.*

The last theorem we discuss here applies to programs that meet a particular restriction, discussed below. In such programs, no mutation is possible by dereferencing a read-only location. This is different than Tag Frame Soundness in that it says a read-only reference may not be used for mutation *even if* it points to a writable domain.

The formal statement of this theorem uses several auxiliary functions. Intuitively, $domains_S(loc)$ gives the set of ground concern domains for the object pointed to by loc in the store S; $locations(e)$ gives every location appearing syntactically in e; $\mathbb{G}_S(loc)$ is the "object graph" of a location, whose nodes are locations and whose edges are field references; $rep_S(loc)$ gives the nodes in $\mathbb{G}_S(loc)$; and $writeReach(S)$ is the reflexive, transitive closure of all the write-enabled field references in the store. That is, $(loc, loc') \in writeReach(S)$ implies that a program with a reference to loc can obtain a write-enabled reference to loc' by a series of field references.

Given all that, the theorem assumes an intermediate state with expression e in the evaluation of a program and some location loc that only names public, not private, concern domains. Then—supposing that any references from e to loc are read-only (assumption 1), that e does not have any aliases into the object graph of loc (assumption 2), and certain restrictions on the program hold (assumption 3)—we can conclude that the evaluation of e to a value will not mutate the object graph of loc.

Theorem 5 (Read-only Soundness). *Suppose the configuration* $\langle \mathbb{E}[e], J, S\rangle$ *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:*

1. $\forall \delta \cdot (loc_\delta \in locations\,(e)) \implies (\delta = \texttt{readonly})$.
2. $\forall loc'_\delta \in locations\,(e) \cdot$
   $(\delta = \varepsilon) \implies \big(\forall loc'' \in rep_S(loc) \cdot \big(loc', loc''\big) \notin writeReach(S)\big)$.
3. $\forall loc' \in dom\,(S) \cdot S\left(loc'\right) = [t \,.\, F] \implies isClass(t) \vee isSpectator(t)$.

If $\langle \mathbb{E}[e], J, S\rangle \overset{*}{\hookrightarrow} \langle \mathbb{E}[v], J', S'\rangle$*, then* $\mathbb{G}_S(loc) = \mathbb{G}_{S'}(loc)$.

So what is this restricted class of programs? Just those that do not contain regular aspects! These non-spectator aspects can "leak" pointers into the computation without being explicitly referenced. Thus, the restrictions on aliasing in assumption 2, which are sufficient without regular aspects, are not sufficient in their presence.

By Read-only Soundness, spectators can be used in a program without breaking the read-only references 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. Said another way, we need whole-program knowledge at the level of effects clauses, aspect instantiation, and dependency declarations to reason about the effects of regular aspects. But with just spectator aspects, we can reason about the effects of a method call solely based on its effects clause—aspect instantiation and dependency declarations are not necessary.

The Tag Frame Soundness theorem allows unseen, private concern domains to be modified during method or advice execution, since the dependency closure of the evaluation dependency table includes all private concern domains. However, because of the (elided) Respect for Privacy theorem—which states that only a spectator and objects directly or transitively created by the spectator may appear in or reference the spectator's private concern domain—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 non-spectator aspects in the program, as represented by the aspect instantiation instructions and dependency declarations. By 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.

7 Evaluation

Many things . . . may become encumbrances if we cling to them blindly and uncritically.

The theoretical analysis above demonstrates the soundness of MAO's effect specifications. However, this says nothing about MAO's *usefulness* — that is, the extent to which MAO's annotations benefit programmers. In this section, we present a small case study that attempts to give a preliminary answer to this question.

Our case study is based on packages in version 1.5.3 of the *AspectJ Programmer's Guide* [3]. We omitted `introduction` and `ltw`, since these very small packages are just for demonstrating AspectJ tools. For the other 7 packages, we specified each aspect using MAO annotations, and then examined the result to determine how much these specifications aided reasoning. The case study's files are available at `http://www.cs.iastate.edu/~leavens/modular-aop/ajpg-153-examples/`.

7.1 Case Study Data

This subsection presents the raw data from our case study in a pair of tables. The subsequent subsection analyzes the data.

Basic statistics about the packages we studied are presented in Table 1. We counted: (1) the number of `.java` files, but in the `tracing` package we only counted files for version 3; (2) the number of lines in these files, with the first 12 lines for each file (a copyright notice) omitted; (3) the number of aspects and (4) abstract aspects; (5) the number of lines (determined by inspection) in the original AspectJ code that would need to be searched to determine control and heap effects of the aspect's advice — this equals the number of lines in advice and method bodies in all aspects[6]; (6) the number of lines (by inspection) in the MAO code that would need to be searched for control effects — this is 0 for `@surround` advice, and 1 for `@curbing` advice, and otherwise includes all lines in advice and method bodies called, if those are part of the aspect; and (7) the number of lines (by inspection) in the MAO code that would need to be searched for heap effects — this is 1 for advice or methods with `@writes` or `@pure` annotations, otherwise it includes all lines in other advice and method bodies.

Table 2 presents some statistics on the use of various features in MAO. We counted: (1) the number of times the `@surround` annotation was used — not counting implicit uses in spectator aspects (2) the number of times `@curbing` was used, (3) the number of times `@writes` was used as a method or advice annotation — we only used this within aspects and did not count implicit uses in spectator aspects, (4) the number of times `@spectator` was used, (5) the number of lines that would have to be searched in AspectJ, within the relevant classes, to determine all the methods that would correspond to the **`writes`** PCDs that the MAO code used. The MAO code used **`writes`** once in each of the two relevant packages.

7.2 Lessons Learned from the Case Study

Table 2 contains lessons about MAO annotation usage. We found many instances of `@surround` advice, especially if one counts the implicit uses of `@surround` in aspects

[6] This assumes, pessimistically, that all code in an aspect can have control and heap effects.

Table 1. Basic statistics about the packages studied

Package	Original Files	Original Lines	Aspects	Abstract aspects	AspectJ lines to search for effects	MAO search for control effects	MAO search for heap effects
tjp	2	62	1		17	0	1
tracing (v3)	6	352	2	1	18	0	0
bean	3	203	1		9	2	2
observer	8	164	2	1	9	0	1
telecom	13	593	3		20	0	4
spacewar	19	2049	8		177	19	23
coordination	8	673	1	1	118	0	0

Table 2. Usage statistics on the packages studied

Package	MAO count use of `@surround`	MAO count use of `@curbing`	MAO count use of `@writes`	MAO count use of `@spectator`	AspectJ lines to search for equivalent of `@writes` PCDs
tjp	1		1		
tracing (v3)	0			2	
bean	0		3	1	
observer	1		2		18
telecom	4		4	1	
spacewar	5	2	27		27
coordination				1	

that are annotated with `@spectator` or `@surround`. By contrast, `@curbing` was only used twice. There were also many uses of `@writes`, especially in `spacewar`'s `Debug` aspect. Moreover, each of the uses of `@spectator` on an aspect suppresses several uses of the `@writes` annotation. While we found several spectators, we found three aspects, like `spacewar`'s `Debug` aspect, do not qualify, principally because they perform I/O and have heap effects on the GUI. For these three aspects it was convenient to use `@surround` at the aspect-level, which is a shorthand for listing `@surround` on each piece of advice.

Lessons about reasoning can be drawn from Table 1. First, the use of MAO's features significantly cuts down the number of lines that need to be inspected to determine control and heap effects. This is important, as we noticed that for examples as large as the `telecom` package or larger it becomes quite difficult to determine the control and heap effects of advice by hand. Thus we believe that a person trying to understand a program, even of such a modest size as `telecom`, would benefit from the use of annotations on advice. The real problem here is not the efficiency of the analysis: it is that without MAO's modular aspect interfaces, any analysis to determine control and heap effects must depend upon fine implementation details of advice body code. Relatively sophisticated static techniques [11, 29] can certainly compute these dependencies, and IDEs present it to programmers [6, 22] but these dependencies will be *fragile*: whenever the configuration of the system, the implementation of the base program, and (especially)

the internals of an aspect changes, then this analysis must be repeated and the results may change in nonlocal, unpredictable ways. The advantage of MAO's effect specifications are, ultimately, that they provide *aspects* with specifications that act both as a unit of analysis and as a boundary to changes. By writing such specifications in MAO annotations, a programmer can make their intentions clear: a tracing aspect can be declared be a spectator upon the program, with no control nor heap effects. If subsequent evolution of the aspect invalidates this intention, the change can be detected statically.

We also found that using the default ownership domain of `World` for everything outside an aspect worked well. This gives some hope that the annotation burden may mostly fall on aspects, and not on the base program, which is usually much larger. It also gives some hope that annotations are not necessary for Java libraries.

In summary, the case study gives some preliminary indications that the features of MAO help in reasoning about control and heap effects in aspect-oriented code. The case study does not contain enough places where the **`writes`** pointcut designator is used to make even much of a preliminary estimate as to its utility.

8 Related Work

All reactionaries are paper tigers. In appearance, the reactionaries are terrifying, but in reality they are not so powerful.

Mulet, Malenfant, and Cointe [25] identified a similar problem: composing metaobjects. They offer language mechanisms that make composition possible, but offer no language mechanisms to help programmers control interference.

Dantas and Walker's Harmless Advice [11] is probably the closest formal system to MAO. Harmless advice is similar to our notion of spectators but allows advice to have "curbing" control effects and to write to what we would call a system I/O domain. So, while technically spectators are more restrictive than harmless aspects, both restrict aspects to make reasoning about heap effects easier. Harmless Advice is formalized using an information flow analysis that establishes that the computation in advice cannot affect the base program's computation. As the name implies, all the advice in this system is harmless, and the user-level calculus offers only one protection domain for the base program. MAO does not restrict advice to be harmless, as it can document different kinds of advice. MAO also has a more precise set of annotations, separately documenting control and heap effects, and allowing more fine-grained specification of heap effects. Thus MAO allows specification of control and heap effects that are outside the range allowed by Harmless Advice, but are useful in AspectJ programs (for example, in the spacewar example). MAO's explicit domain declarations and aspect domain parameters allow concern domains to cross-cut the program's modularity structure, whereas the protection domain structure of Harmless Advice is tied to the program's structure.

Kiczales and Mezini [22] take a different approach to the problem of reasoning about aspect-oriented software, by introducing "aspect-aware interfaces." These interfaces are computed from a whole program's configuration and provide a bi-directional mapping from methods to associated advice, and from advice to advised methods. This certainly is helpful in reasoning about control effects of advice that may apply at a given join

point or program point. As such it could be used in conjunction with MAO's annotations to determine whether the advice being applied has potential control effects. Aspect-aware interfaces give no help in reasoning about potential heap effects of advice or in reasoning, apart from helping one find what advice might have to be considered. By contrast MAO's annotations can provide more help with such questions.

Another route towards helping people reason about aspect-oriented software is provided by research that establishes interfaces for aspect-oriented program modules. Generally, these systems attach interfaces to base code elements that either permit or prohibit advice from being applied, or describe what advice has been applied. So, Pointcut Interfaces [17], Open Modules [1, 26], Aspectual Collaborations [23], and XPIs [15, 31] all require code to declare or specify the join points to which aspects may be attached. In contrast our focus is on specifying *aspects* and their effects. That is, MAO allows programmers to specify aspects to make reasoning about their effects easier, instead of restricting what they can do. Thus, in MAO, rather than describing potential pointcuts in the base code, programmers describe the important properties of their designs, such as what concerns exist, what advice writes what concerns, etc. These annotations are contained within existing interfaces in their code. MAO allows the use of some of these annotations in writing semantic PCDs, with its **`writes`** PCD. Furthermore, MAO's statically checked annotations help make reasoning about control and heap effects easier.

MAO is also related to a range of work on categorizing and classifying aspects [19, 20, 29]. Generally, this work identifies a number of (relatively) fine-grained aspect categories, either via manual or automatic analysis. MAO, however, does not address categorization per se: rather, our aim is to provide practical language constructs programmers can use to express properties of their aspects.

In terms of language mechanisms, MAO and especially its concern domains are closely related to other ownership and confined type systems [2, 4, 5, 12, 16, 24, 27, 28, 32]. MAO's novelty here is in demonstrating how ownership types can be used to capture the concerns in an aspect-oriented system: the techniques providing concern domains (a statically fixed set of ownership domains; objects tied to domains by type parameterization and defaults; domains for effect disjointness) are now well known. MAO shows how even a such a simple ownership type and effect system can aid reasoning about the subtle heap effects that may occur in aspect-oriented systems.

9 Conclusion

Conclusions invariably come after investigation, and not before.

In this paper, we have presented Modular Aspects with Ownership, MAO. MAO makes four contributions to the design of Aspect-Oriented languages, to make it easier for programmers to determine how aspects will affect the base code of the program, and how they interfere with each other.

First, surround advice uses simple syntactic restrictions so that programmers can ensure that an aspect will not perturb the control flow of the program to which it is bound. Second, concern domains provide an ownership type and effect system to make clear whether aspects modify data structures in the base program, and if so, what parts of the

data they modify. Third, the **`writes`** PCD leverages concern domains to provide succinct, precise designations for pointcuts that modify data. Finally, MAO is underpinned with a formal model and demonstrates that spectator aspects (defining only surround advice, and writing only their own concern domain) cannot materially affect the execution of other classes or aspects in the program.

Using MAO, programmers can specify the full range of their aspects' interactions with the base program and their interference with one another, making aspect oriented programs more precisely documented and easier to reason about.

Acknowledgments

The work of Clifton and Leavens was supported in part by NSF grant CCF-0428078. The work of Leavens was also supported in part by NSF grant CCF-0429567. The work of Noble was supported in part by a gift from Microsoft Research, by an IBM Eclipse Innovation Grant, by the EPSRC grant Practical Ownership Types for Objects and Aspect Programs, EP/D061644/1, and by the Royal Society of New Zealand Marsden Fund. Thanks to many anonymous reviewers for comments on prior drafts of this work.

References

[1] Aldrich, J.: Open modules: Modular reasoning about advice. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, Springer, Heidelberg (2005)
[2] Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, Springer, Heidelberg (2004)
[3] AspectJ Team: The AspectJ programming guide, Version 1.5.3. (2006), Available from `http://eclipse.org/aspecti`
[4] Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL, pp. 213–223 (2003)
[5] Clarke, D., Drossopoulou, S.: Ownership, Encapsulation, and the Disjointness of Type and Effect. In: OOPSLA (2002)
[6] Clement, A., Colyer, A., Kersten, M.: Aspect-oriented programming with AJDT. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
[7] Clifton, C.: A design discipline and language features for modular reasoning in aspect-oriented programs. PhD thesis, Iowa State (2005)
[8] Clifton, C., Leavens, G.T.: MiniMAO$_1$: Investigating the semantics of proceed. Sci. Comput. Programming 63(3), 321–374 (2006)
[9] Clifton, C., Leavens, G.T.: Observers and assistants: A proposal for modular aspect-oriented reasoning. In: FOAL (2002)
[10] Clifton, C., Leavens, G.T.: Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report TR #02-10, Dept. of Computer Science, Iowa State University (October 2002)
[11] Dantas, D.S., Walker, D.: Harmless advice. In: POPL (2006)
[12] Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology 4(8), 5–32 (2005)
[13] Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.): Aspect-Oriented Software Development. Addison-Wesley, Reading (2005)

[14] Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. In: Formal Syntax and Semantics of Java, ch. 7, pp. 241–269. Springer, Heidelberg (1999)
[15] Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular software design with crosscutting interfaces. IEEE Software, 51–60 (January/February 2006)
[16] Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating Objects with Confined Types. In: OOPSLA, pp. 241–255 (2001)
[17] Gudmundson, S., Kiczales, G.: Addressing practical software development issues in AspectJ with a pointcut interface. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
[18] Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java. A minimal core calculus for Java and GJ. ACM Trans. Prog. Lang. Syst. 23(3), 396–459 (2001)
[19] Katz, S.: Aspect categories and classes of temporal properties. In: Rashid, A., Aksit, M. (eds.) Transactions on Aspect-Oriented Software Development I. LNCS, vol. 3880, Springer, Heidelberg (2006)
[20] Katz, S., Gil, Y.: Aspects and superimpositions. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, Springer, Heidelberg (1999)
[21] Kiczales, G.: The fun has just begun. AOSD'03 Keynote Address (2003), available from `http://www.cs.ubc.ca/~gregor`
[22] Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: ICSE, pp. 49–58. ACM Press, New York (2005)
[23] Lieberherr, K., Lorenz, D.H., Ovlinger, J.: Aspectual collaborations: Combining modules and aspects. The Computer Journal 6(5), 542–565 (2003)
[24] Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL, pp. 359–371 (2006)
[25] Mulet, P., Malenfant, J., Cointe, P.: Towards a methodology for explicit composition of metaobjects. In: OOPSLA, pp. 316–330. ACM, New York (1995)
[26] Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O., Sittampalam, G.: Adding open modules to AspectJ. In: AOSD (2006)
[27] Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic Java. In: OOPSLA, pp. 311–324 (2006)
[28] Potanin, A., Noble, J., Clarke, D., Biddle, R.: Featherweight Generic Confinement. Journal of Functional Programming 16(6), 793–811 (2006)
[29] Rinard, M., Salcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, Springer, Heidelberg (2004)
[30] Steimann, F.: The paradoxical success of aspect-oriented programming. In: OOPSLA, pp. 481–497 (2006)
[31] Sullivan, K., Griswold, W., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: FSE, pp. 166–175 (May 2005)
[32] Vitek, J., Bokowski, B.: Confined types in Java. S—P&E 31(6), 507–532 (2001)