Execution Levels for Aspect-Oriented Programming: Design, Semantics, Implementations and Applications

Éric Tanter[a], Ismael Figueroa[a,b], Nicolas Tabareau[b]

[a]*PLEIAD Laboratory*
Computer Science Department (DCC)
University of Chile – Santiago, Chile
[b]*INRIA – Nantes, France*

Abstract

In aspect-oriented programming (AOP) languages, advice evaluation is usually considered as part of the base program evaluation. This is also the case for certain pointcuts, such as `if` pointcuts in AspectJ, or simply all pointcuts in higher-order aspect languages like AspectScheme. While viewing aspects as part of base level computation clearly distinguishes AOP from reflection, it also comes at a price: because aspects observe base level computation, evaluating pointcuts and advice at the base level can trigger infinite regression. To avoid these pitfalls, aspect languages propose ad-hoc mechanisms, which increase the complexity for programmers while being insufficient in many cases. After shedding light on the many facets of the issue, this paper proposes to clarify the situation by introducing levels of execution in the programming language, thereby allowing aspects to observe and run at specific, possibly different, levels. We adopt a defensive default that avoids infinite regression, and gives advanced programmers the means to override this default using level-shifting operators. We then study execution levels both in practice and in theory. First, we study the relevance of the issues addressed by execution levels in existing aspect-oriented programs. We then formalize the semantics of execution levels and prove that the default semantics is indeed free of aspect loops. Finally, we report on existing implementations of execution levels for aspect-oriented extensions of Scheme, JavaScript and Java, discussing their implementation techniques and current applications.

Key words: Aspect-oriented programming, meta-programming, infinite

☆Earlier versions of the main matter of this article appeared in the informal proceedings of the Scheme and Functional Programming workshop 2009 [42], and in the proceedings of the 9th International Conference on Aspect-Oriented Software Development [43]. Additional content on exception handling appears in the proceedings of the Foundations of Aspect Languages workshop 2011 [22]. Accompanying material and notes are available online:
`http://pleiad.cl/research/scope/levels`
Éric Tanter is partially funded by FONDECYT Project 1110051. Ismael Figueroa is funded by a CONICYT-Chile Doctoral Scholarship.

Email addresses: `etanter@dcc.uchile.cl` (Éric Tanter), `ifiguero@dcc.uchile.cl` (Ismael Figueroa), `nicolas.tabareau@inria.fr` (Nicolas Tabareau)

regression, execution levels, AspectJ, LAScheme, AspectScript.

1. Introduction

In the pointcut-advice model of aspect-oriented programming (AOP) [31, 53], as embodied in *e.g.* AspectJ [29] and AspectScheme [19], crosscutting behavior is defined by means of pointcuts and advices. A pointcut is a predicate that matches program execution points, called join points; and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices.

A major challenge in aspect language design is to cleanly and concisely express where and when aspects should apply. To this end, expressive pointcut languages have been devised. While pointcuts were initially conceived as purely "meta" predicates that cannot have any interaction with base level code [53], the needs of practitioners have led aspect languages to include more expressive pointcut mechanisms. This is the case of the `if` pointcut in AspectJ, which takes an arbitrary Java expression and matches at a given join point only if the expression evaluates to true. Going a step further, higher-order aspect languages like AspectScheme and AspectScript [47] consider a pointcut as a first-class function like any other, thus giving the full computational power of the base language to express both pointcuts and advices. In these cases pointcut evaluation is performed at the base level, emitting its own join points.

On the other hand, advices were initially seen as a piece of base-level functionality [53]. In other words, an advice is just like an ordinary function or method, that happens to be triggered implicitly whenever the associated pointcut matches. Indeed, considering advice as base-level code clearly distinguishes AOP from runtime meta-object protocols (MOPs), considered by many as the ancestors of this form of AOP.

Because aspects observe evaluation of base computation, evaluating advices and pointcuts at the base level can trigger infinite regression. This is a widely-recognized[1] problem that can happen easily: it is sufficient for the evaluation of an advice or a pointcut to trigger a join point that is potentially matched by the same aspect, either directly or indirectly. Although the potential for infinite regression is a direct consequence of considering advice and pointcut execution as base computation, existing solutions are not based on this insight. Instead, most of them rely on control flow checks, which are eventually unable to properly discriminate aspect computation from base computation.

To address this issue we choose to question the basic assumption that pointcut and advice are *intrinsically* either base computation or meta computation. Looking at how programmers use advices, it turns out that while some advices are clearly base code, some are used to implement concerns like synchro-

[1] `http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html`

nization and monitoring, which were previously considered as forms of metaprogramming. Inspired by a solution to infinite regression in MOPs [11], we propose a reconciliating approach in which the metaness concern is decoupled from the pointcut-advice mechanism, by introducing a notion of *level of execution* to structure computation in the core execution model.

In execution levels computation is stratified in a tower in which the flow of control navigates. Given an initial level, join points are always emitted one level above the *current level*—which is a dynamically scoped value. Aspects are deployed at a specific level and can only affect join points emitted one level below. This way, computation performed by aspects is not observable to themselves nor to any other aspects that are deployed at the same level or below.

To alleviate the task for non-expert programmers, we adopt a defensive default that avoids regression by making aspect computation happen at a higher level than base computation. For the advanced programmer, level-shifting operators provide complete control over what aspects see and where they run (*i.e.* who sees them), at the expense of reintroducing the potential for infinite loops. Execution levels seamlessly address all the issues of current proposals for avoiding infinite loops, while maintaining extreme simplicity in the most common cases for which programmers do not even need to be aware of them.

This paper is structured as follows: Section 2 describes several issues with the current state of affairs regarding aspect weaving. Section 3 discusses related work in both AOP and MOPs, and focuses on the central issue of conflation. Section 4 develops our proposal of execution levels, including its safe default, explores the flexibility offered by explicit level shifting, and shows how all the issues raised previously are addressed. Section 5 reports on an evaluation of the relevance of the problem we address and the benefits of our proposal in practice through an analysis of a large number of existing AspectJ programs. Section 6 formalizes the operational semantics of our proposal, by modeling a core higher-order aspect language with execution levels, and prove that *programs that do not make use of explicit level shifting are free of aspect loops* (The full proof is given in Appendix A). Section 7 reports on three practical implementations of execution levels in the context of Scheme, JavaScript and Java, highlighting both applications and implementation techniques. Finally, Section 8 concludes.

2. A Plethora of Issues

This section briefly presents several issues associated to the current state of affairs of aspect languages. The first one, advice loops, is widely known, so much so that its "solution", which relies on control flow checks, has almost acquired the status of a pattern. The second issue we discuss, pointcut loops, is only partially addressed. The following four issues, regarding different kinds of advice, visibility of aspects, concurrency, and exceptions reveal fundamental flaws of currently-acknowledged patterns.

We illustrate the issues in pedagogical variants of the geometrical shapes example—basically, points that can be moved around—using AspectJ as an implementation language. To the best of our knowledge, most (if not all) of these issues are present in mainstream languages like AspectJ, research languages like AspectScheme, and in general in aspect languages based on the pointcut-advice model, at least as described in [53]. We will come back to other languages and proposals in Section 3.

2.1. Advice Loops

Consider an `Activity` aspect that traces whenever a point is active, that is, when one of its methods is executing:

```
aspect Activity {
 before(Point p) : execution(* Point.*(..)) && this(p) {
  System.out.println("point active: " + p);
 }
}
```

This AspectJ definition introduces an aspect named `Activity` with a single pointcut and advice. It states that before execution of any method of a `Point` object, the advice prints a message. The `this(p)` part of the pointcut is used to bind the currently-executing object to the identifier `p`, to be used in the advice.

While straightforward, this definition fails: tracing a point object is done by (implicitly) calling its `toString` method, whose execution is going to be matched by the same aspect, and so on infinitely. Folk wisdom is that the solution consists in excluding join points that occur in the control flow of the advice execution. To identify the advice execution, AspectJ includes a specific pointcut designator, which can be used as follows:

```
execution(* Point.*(..)) && this(p)
  && !cflow(adviceexecution() && within(Activity));
```

The added conjunction excludes join points that are in the control flow of an advice execution join point triggered by the `Activity` aspect (the `adviceexecution` join point itself is not parametrized in AspectJ). Note that there exists variants of this pattern. Some are too "strict": omitting the `within` part implies excluding join points in the control flow of *any* advice of *any* aspect, while using only `cflow(within(Activity))` rules out join points that can occur in the control flow of a standard, non-advice, method of `Activity` (an aspect, like an object, may have instance variables and methods). Finally, not using `cflow`, but just checking for `!within(Activity)` is too "loose", since it only rejects join points that occur *lexically* in the aspect; this would clearly be insufficient in our example, because only the *call* to `toString` happens lexically in the advice, not its actual *execution*[2].

[2]The join point model of AspectJ differentiates *call* join points happening on the caller side, prior to method lookup, and *execution* join points, on the callee side, after method lookup.

2.2. Pointcut Loops

Let us refine the `Activity` aspect such that only point objects located within a given area are subject to monitoring. We can use the `if` pointcut designator for this purpose:

```
aspect Activity {
 Area area = ...;
 before(Point p) : execution (* Point.*(..)) && this(p)
   && if(p.isInside(area)) ... {
   ...
 }
}
```

As before, `this(p)` is used to get a hold on the currently-executing point object; we can use it in the `if` condition to check that the point is within the area. This definition is however incorrect, for a similar reason as above. Calling `isInside` eventually results in an execution join point against which that *very same* pointcut is evaluated again, provoking an infinite loop. As a solution we could revert to an imperfect (too strict) variant, by ruling out join points in the control flow of any join point that occurs in the aspect: `!cflow(within(Activity))`.

However this is totally impossible with current AspectJ compilers (in the absence of a complete formal semantics of the language, compilers dictate). Surprisingly, both the oficial AspectJ compiler `ajc`, and the alternative `abc` compiler [6] *hide join points occurring* lexically *in an* `if` *pointcut*. Therefore, the roots of the guilty flows of execution cannot be identified, because they are hidden! The only solution is to refactor the aspect and move out the `if` condition from the pointcut either to the advice(s), or to an external method whose *execution* (but not its call) will be visible. The latter solution is arguably the best, but it involves creating new methods for each conditional that appear in pointcuts. In addition it is inconsistent, and probably unexpected, that the semantics of the program should change as much depending on whether a method call or a conditional expression is used in an `if` pointcut.

It is important to remark that pointcut loops are not an issue specific to AspectJ or static aspect languages in general. In fact, the problem is evident from the beginning in the design of languages like AspectScheme due to the first-class nature of pointcuts (see [19], Section 2), and it is addressed with special primitives that disable emission of join points on a particular function application, as we will see in Section 3.

2.3. Confusion all Around

To add to the already-large confusion and complexity, control flow checks (if at all possible) interfere in unpleasing ways with the *kind* of advice (before, after, around) bound to a pointcut, leading to aspects not observing base computation as expected. Up to now, we have only used *before* advice in the examples but aspect languages generally support *around* advice as well. Consider the following tracing aspect with *around* advice:

```
aspect Activity {
 Object around(Point p) : execution(* Point.*(..)) && this(p) {
  System.out.println("execution on point: " + p);
  return proceed(p);
 }
}
```

Since the advice now has to call `proceed` in order to trigger the original computation, an advice execution control flow check, whose purpose is to avoid the advice loop, will discard *all subsequent join points of the nested base program execution*, preventing the tracing of *all* method executions on `p` as intended.

In the particular case of AspectJ, a before advice will actually print all method executions on `p`, including executions caused by self calls (such as `move` calling `setX`, or recursive methods). Of course, because the advice prints the point object, it is subject to an advice loop that can only be avoided using a control flow check (Section 2.1).

The fact that using before advice works might seem surprising because, intuitively, one could expect that

```
before() : pc() { ...before action... }
```

is equivalent to[3]:

```
Object around() : pc() {
  ...before action...
  return proceed();
}
```

Nevertheless, the core of the issue is that control flow checks are unable to discriminate advice execution from the original base program computation triggered by `proceed`. Relying on AspectJ-like semantics of `before` advice is still insufficient, since it does not cover the cases where the advice needs to change the arguments or return value of `proceed`. We consider the unfortunate interaction between control flow checks and `proceed` a major issue of current languages.

2.4. Visibility (of) Aspects

Previous issues mostly deal with the visibility of aspect computation to itself. It is also important to consider the fact that several aspects coexist in a program, and may or may not need to observe each other's computation.

[3]The degree to which this equivalence is explicitly recognized and accepted differs according to the language. For instance, in AspectScheme, before advice is only syntactic sugar for around advice following the given pattern. The formal semantics of AspectScheme therefore contemplates only around advice. On the contrary, AspectJ implementations do not consider before advice as syntactic sugar, but as an opportunity for optimization.

Suppose we add a FrequencyDisplay aspect that measures the number of times a point object is used per time unit in order to update its displayed size accordingly. The sheer fact of having the Activity aspect calling isInside and toString means that the measurements of FrequencyDisplay are silently affected.

Conversely, one may want the computation of an aspect to be (at least partially) visible to others. Suppose that the Activity aspect calls the refresh() method of a global Display. In addition, a Coalescing aspect is in charge of gathering all refresh actions that occur within a certain time interval into a single refresh. Both base objects and the Activity aspect call refresh, and Coalescing ought to be aware of all of them. For that, at least part of Activity's computation must be visible to the coalescing aspect.

Control-flow checks cannot fulfill the need for visibility control between aspects in a satisfying manner, mostly for the same reason we described in Section 2.3; neither can aspect precedence, which only deals with the issue of shared join points.

2.5. Concurrency

Moreover, control-flow checks completely break in the presence of concurrency. Concurrent advice execution is gaining attention to take advantage of multicores, as for instance with the technique of buffered advice [1]. To illustrate the issue of control-flow checks in presence of concurrency, suppose the Activity aspect logs its output to a file. In order to be more efficient, writing to the file is delegated to a timer thread that buffers pending log actions, and flushes them to the file at certain time intervals[4]. In AspectJ, a simplified version of this behavior could be implemented using the Timer/TimerTask framework of Java, *e.g.*:

```
class LogTask extends TimerTask {
  Point p;
  LogTask(Point p){ this.p = p; }
  void run(){
    log.write("execution on point: " + p);
  }
}

aspect Activity {
  Timer t = new Timer();
  before(Point p) : execution (* Point.*(..)) && this(p) {
    t.schedule(new LogTask(p), 1000);
  }
 }
```

[4]Note that the timer thread runs independently of the aspect and therefore there is no parent-child relation between the flow of the aspect and the flow of the timer thread.

Writing to the file in the `LogTask` implies calling the `toString` method of point objects, resulting in an infinite loop. This loop, however, cannot be avoided through control-flow checks related to the advice execution, simply because the execution of `toString` does *not* happen in the control flow of the advice, but in a separate thread of execution. Note that if the `LogTask` class were defined lexically within `Activity`, either as a named or anonymous inner class, then the `!cflow(within(..))` pattern would work. However, this is not the general case, and in addition, as argued in Section 2.1, the pattern is too strict.

2.6. Exception Handling

An additional issue comes from the fact that aspect-oriented languages usually extend existing languages without paying special attention to the underlying exception handling mechanism. Accordingly, there is no difference between aspect and base handlers or exceptions, and as a consequence exceptions may inadvertently trigger execution of unintended handlers.

This happens when aspect exceptions are handled by base handlers, or dually, when base exceptions are caught by aspect handlers. Although these situations may be desired in some cases, *e.g.* to implement global exception-handling aspects, current aspect languages lack a mechanism to control this interaction. This makes it difficult to reason about complex control flow interactions between aspects and base code in a system.

A base handler catching an aspect exception. To illustrate the first problematic situation consider the single-threaded `Activity` aspect with logging (using the standard Java `Logger` class):[5]

```
aspect Activity {
 String logName = ...
 Object around(Shape s):
   call(* Shape+.*(..)) && this(s) {
     Logger.getLogger(logName).log("execution on shape" + s);
     return proceed(s);
   }
}
```

The aspect intercepts calls to any method of `Shape` objects, including objects from subclasses. Note that `getLogger` throws a `NullPointerException` if its argument is null.

Consider now the `pipeline` method that applies a composed transformation to a shape, yielding a new shape[6]. The method receives a list of `Transforms`, it concatenates them, and finally apply the resulting transformation to the shape. Applying the composed transformation fails with a `NullPointerException` if a transformation in the composition is null.

[5]Assume a control-flow check (Section 2.1) to avoid looping on the implicit `toString` calls.

[6]We base our example on the Java 6 `java.awt.geom` API for 2-D geometry.

```
Shape pipeline(Shape s, List<Transform> trans) {
   try {
     Transform t = new Transform(); // identity trans.
     for (Transform tran : trans) {
         t.concatenate(tran);
     }
     return s.transform(t);
   }
   catch (NullPointerException npe) {
     return s;
   }
}
```

Note that instead of eagerly performing a linear scan of the list of transformations just to check that no transformation is null, or to check for null at every step of the iteration, the whole operation is performed inside a `try-catch` block. If applying the composed transformation fails, the method simply returns the unmodified shape.

Observe that the call to `transform` is advised by the `Activity` aspect. Because the exception propagation semantics do not take into account the difference between aspect and base exceptions or handlers, a `NullPointerException` raised from evaluation of the advice will be caught by the `catch` block of `pipeline`. As a result the method will return the original shape regardless of the transformations it receives as argument.

An aspect handler catching a base exception. Conversely, the problem can manifest as a base exception caught by an aspect handler. Consider the following variant of the `Activity` aspect, which enforces a strict logging policy in which the program is halted if the aspect cannot properly write to the log:

```
aspect Activity {
 String logName = ...
 Object around(Shape s): call(* Shape+.*(..)) && this(s) {
   try {
     Logger.getLogger(logName).log("execution on shape: " + s);
     return proceed(s);
   }
   catch (NullPointerException npe) {
     System.exit(ERROR);
     return null;
   }
 }
}
```

In this case, exceptions thrown by `transform` above will be caught by the advice handler because this handler is closer in the call stack to the point in execution where the exception is thrown. Consequently, passing a list of transformations with null elements to `pipeline` will halt the program instead of just returning the unmodified shape.

3. From Conflation To Stratification

Some of the issues presented above are known by the AOP community, and there are several proposals to address them. This section describes these proposals and then shows that all the issues come as a consequence of the *conflation* of aspect and base computation. To show this we then step back to reflect about conflation in meta-object protocols, considered as ancestors of AOP, to finally describe how conflation manifests in aspect-oriented programming. Based on these insights Section 4 presents our proposal of execution levels for aspect-oriented programming.

3.1. Lessons Learned from Aspect-Oriented Programming

Existing proposals address the issues presented in Sections 2.1 to 2.5 either by selectively and explicitly disabling aspect weaving, or by implicitly detecting looping situations and avoiding them. In addition there are specific proposals to deal only with the exception handling issues of Section 2.6.

Explicit disabling of weaving. AspectScheme [19] supports a *primitive function application*, `app/prim`, which does not generate a join point. This is required in AspectScheme to address looping issues that arise from the fact that pointcuts and advices are standard first-class functions. Evaluating a pointcut therefore immediately results in a loop if no mechanism is provided. The same holds for applying `proceed` (which is not a keyword, but a function too). Unfortunately, `app/prim` does not help in most of the issues we presented because it only hides the *application* join point, not the subsequent function execution and nested computation (in that sense, it shares the same limitations as the lexical `within` pointcut of AspectJ).

AspectML [14] suggests a `disable` primitive that hides the computation of a whole expression. While this is certainly more effective than a pure lexical primitive like `app/prim`, it shares the same flaws as the control flow patterns in AspectJ. A similar mechanism is proposed by Bockisch *et al.* in the form of a `declare ignore` statement [7]. This statement allows to ignore join points raised during the control flow of specific aspects or individual pointcuts. Although more expressive, it can be considered as a variant of the `disable` primitive, thus sharing its limitations.

Controlling aspect reentrancy. In previous work, we draw an analysis of the two first issues of the previous section, under the umbrella term of *aspect reentrancy* [39]. We distinguish between base-triggered reentrancy (caused when an aspect matches join points that are produced by *e.g.* a recursive base program, not discussed here), advice-triggered reentrancy (Section 2.1), and pointcut-triggered reentrancy (Section 2.2). We show that base- and advice-triggered reentrancy can be avoided using well-known patterns like control-flow checks, at the expense of complex definitions. We also pinpoint the fact that current AspectJ compilers make it impossible to get rid of pointcut-triggered reentrancy without having to refactor the aspect definition.

In particular, we propose a revised semantics for `if` pointcuts, such that their execution is fully visible to all aspects, except themselves. To be able to determine reentrant join points at a pointcut, we introduce a *pointcut execution* join point, similarly to the already-existing advice execution join point found in several aspect languages. Such a join point is produced internally upon pointcut evaluation, and is necessary to be able to get rid of pointcut-based reentrancy. The approach is however also based on control-flow checks.

Stratified Aspects. To the best of our knowledge, the first piece of work directly related to the issue of infinite recursion with the pointcut/advice mechanism is due to Bodden and colleagues [8]. With stratified aspects, aspects are associated with levels, and the scope of pointcuts is restricted to join points of lower levels. The work focuses on advice-triggered reentrancy only, and does not mention the issue related to *e.g.* `if` pointcuts.

The fundamental limit of stratified aspects is that levels are *statically* determined. That is, classes live at level 0, aspects at level 1, meta-aspects at level 2, and so forth. As a consequence, as recognized by the authors, it is impossible to properly handle shift downs, *i.e.* when an aspect calls a method of a level 0 object.

Stratified aspects only avoid advice looping provided that aspects (and meta-aspects, and so on) never call code at lower levels. This is a serious limitation because entities at each static level have to be implemented as completely closed worlds: they cannot use any shared library that is subject to weaving. In addition, a static analysis would be required to enforce the absence of loops. This could be good enough for particular applications of aspects, but would still be subject to the other issues of Section 2.

Originated from the work on stratified aspects, the recent join point types for implicit invocation with implicit annoucement (IIIA) by Steimann *et al.* [38] advocates for representing join points as instances of class-like structures called join point types. Although their work is focused on modularity issues, they relate their proposal with the issue of infinite regression. They argue that stratified aspects can be emulated simply by defining disjoint sets of join point types. In addition, it is said that aspects cannot exhibit join points in order to avoid infinite loops, following the stratified aspects principle. Being a static mechanism, this approach suffers from the same closed-world limitation as stratified aspects, yielding infinite loops as soon as shared entities are used[7].

Solving Confusion in Exception Handling. One of the *bug patterns* described by Coelho *et al.* [13] is that of a base handler inadvertently catching an aspect exception. To avoid it, they propose to define custom exception hierarchies for each aspect, as also suggested by Robillard and Murphy [35] for Java programs.

[7]See loop examples in IIIA online: `http://pleiad.cl/research/scope/levels`

However, the use of custom exception hierarchies does not solve the issues presented in Section 2.6. First, because a custom aspect exception can still be caught by a general base handler, like `catch(Exception e)` in AspectJ. And also because regular exceptions raised in the control flow of an advice can still propagate to an unintended base handler. This problems could be alleviated using static exception flow analyses, but with a considerable increase in the cost and complexity of aspect development.

Cacho *et al.* [10] propose the notion of exception channels and pluggable handlers. They extend AspectJ's pointcut language to allow the creation of global exception paths, associating a particular handler to a set of code locations where exceptions are raised. Although this approach can solve the presented issues, it must be done manually and case-by-case, which can be error-prone and hard to maintain.

We conclude this section remarking that although the first three approaches presented seem to address the looping issues, all of them are (at best) based on control flow checks and therefore fail when considering the issues presented in Section 2.3, 2.4 and 2.5. None of them considers the issue of confusing base and advice execution through `proceed` (our previous reentrancy control proposal is formulated only in terms of before advice). Mutual visibility among aspects as well as the possibility of delayed advice computation are also not considered. The most promising solution seems to be that of stratified aspects, but the limitation they face is too restrictive in practice.

3.2. Lessons Learned from Meta-Object Protocols

It turns out that the issues we have been exposing up to now are, to a large extent, reminiscent of the issue of *meta-circularity*, which has long been identified in reflective architectures.[8]

Reflective towers. Seminal work on reflection focused on the notion of a *reflective tower*. This tower is a stack of interpreters, each one executing the one below. Reification and reflection are level-shifting mechanisms, by which one can navigate in the tower. This idea was first introduced by Brian Smith [37] with 2-Lisp and 3-Lisp, and different flavors of it were subsequently explored, with languages like Brown [52] and Blond [15].

2-Lisp focuses on structural reflection, by which values can be moved up and down. An `up` operation reduces its argument to a value and returns (a representation of) the internal structure of that value (*i.e.* its "upper" identity). Conversely, `down` returns the base-level value that corresponds to a given internal structure. 3-Lisp introduces procedural reflection by which *computation* can actually be moved in the tower. This is done by introducing a special kind of abstraction, a *reflective procedure*, which is a procedure of fixed arity that,

[8]We could go much further back in time, and relate these issues to Russel's paradox; but that would take us too far.

when applied, runs at the level above. It receives as parameters some internal structures of the interpreter (typically the current expression, environment, and continuation). Control can return back to the level below by applying the evaluation function. Blond makes the distinction between reflective procedures that run at the level above the level at which they are *applied*, and procedures that run at the level above that at which they were *defined*.

Infinite regression and conflation. The issue of infinite regression in metalevel architectures has long been identified [17, 28]. The typical solutions to address meta-circularity issues were: *a)* to add base checks that stop regression, and *b)* to introduce a more primitive mechanism that is *not* subject to redefinition.

A major step forward was proposed by Chiba *et al.*, who recognized the ad-hoc nature of regression checks, and identified the more general issue of metalevel *conflation* [11]. In the proposed meta-helix architecture, extensions to objects (*e.g.* new fields) are layered on top of each other. Levels are reified, at runtime if necessary, and an object has a representative at each level. An "implemented-by" relation based on delegation keeps levels clearly separated. Denker *et al.* develop a similar mechanism in which metaobjects receive an implicit "meta-context" argument such that they can explicitly determine at which level they run [16]. Metaobjects always execute at their original level, and execution only shift downs when a metaobject triggers the execution on the reification of an execution event (*i.e.* a join point in AOP terminology).

The key lesson from the work in metaobject protocols is that the fundamental issue comes from conflating levels that ought to be kept separate at runtime.

3.3. Towards Execution Levels for AOP

Let us reflect on the intriguing and visionary sentence from a seminal paper of Kiczales that gave birth to what is now known as aspect-oriented programming:

> *"very often, the concepts that are most natural to use at the meta-level cross-cut those provided at the base level."* [26]

The sentence clearly places what we now call an aspect at the "meta-level", something of a different kind. Arising from work in meta-object protocols, designed to address what was mostly a *locality* issue, aspects have since then lost their "metaness", at least to some extent. While a pointcut is originally seen as a pure metalevel entity (a method applicability predicate expressed in its own language [53]), advice is just another—probably misnamed—piece of code that has the same ontological status as a method [27].

Clearly, the pure view of pointcuts does not hold in practice: pointcuts *do* generate join points. This is the case with the `if` pointcut of AspectJ, and simply with all pointcuts in higher-order aspect languages like AspectScheme, AspectML, and AspectScript. Whether advice is meta or not is debatable, and we believe, depends on what advice we are considering. Our stance on this issue is that while we recognize that some aspects can be part of the base

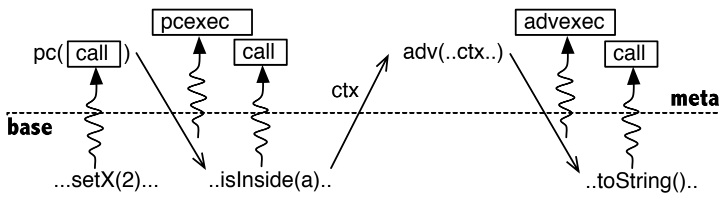

Figure 1: Join points and aspect execution in aspect languages with `if` pointcuts or higher-order pointcuts, and base-level advice.

application logic, some aspects do correspond to metalevel concerns as stated above by Kiczales. In other words, we acknowledge the fact that AOP *can* be (and is) used for metaprogramming. Many applications that used to be considered as illustrative of MOPs [54], like synchronization and monitoring, are now programmed using aspects, mostly due to the practical benefits of pointcut languages.

In the context of AOP, meta-circularity stems from the fact that we are using all the power of the base language (*e.g.* Java) to redefine, via (`if`) pointcuts and advices, the meaning of some specific base computation (join points). Moreover, the two initial approaches used in reflective architectures, ad-hoc checks and special primitive mechanisms, are also found in AOP. Explicit control-flow checks in AspectJ, or the default reentrancy control we proposed previously [39], are examples of the former. AspectScheme's `app/prim` and AspectML's `disable` are examples of the latter.

To see the connection with conflation of levels, let us consider Figure 1. When a call occurs at the base level, a call join point is created (snaky arrow). The join point (call box) is passed to the pointcut. As already discussed, the evaluation of the pointcut does not occur entirely at the metalevel, due to the presence of *e.g.* `if` pointcuts. The pointcut returns either false (if there is no match), or a list of bindings (`ctx`) if there is a match. The bindings are used to expose context information to the advice. The advice is then called, and runs at the base level. This means that calls occurring in the dynamic extent of the pointcut or advice execution are reified as call join points, just as visible as the first one. The fact that all join points (boxes) are present at the same "level" depicts conflation. Figure 1 also shows pointcut and advice execution join points, used in control-flow checks.

Following the meta-helix architecture proposed by Chiba *et al.* would mean placing pointcut and advice execution at a higher-level of execution than "base" code. On the one hand, this allows for a stable semantics, where issues of conflation can be avoided [11, 8, 16]. On the other hand, this boils down to reconsidering AOP as just a form of metaprogramming, which defeats the original idea of AOP [27].

4. Execution Levels for Aspect-Oriented Programming

We propose to resolve the "base vs meta" ontological conflict of aspect-oriented programming by decoupling the "metaness" concern from the pointcut and advice mechanism. To do this we introduce *execution levels* in the language in order to structure computation. Each evaluation step happens at a *current level of execution.* Introducing the level as a dynamic property of execution means that the same piece of code can be evaluated at different levels during execution of a program. This is the key fundamental difference between execution levels and stratified aspects (Section 3.1). The essence of our proposal is to use the execution level to establish and to distinguish the ontological status of pointcuts and advices.

In this section we introduce execution levels and discuss how they can be used in conjunction with aspects. Instead of using AspectJ as in the previous sections we use our research language LAScheme, a simple higher-order aspect language with execution levels. Its formal semantics is described in Section 6.

We start with an overview of LAScheme in Section 4.1. Then Section 4.2 exposes the default way in which pointcuts and advices are evaluated, showing how this default behavior directly addresses the issues described in Sections 2.1, 2.2, and 2.3. Next, Section 4.3 gives more control to programmers by introducing level-shifting operators, making it possible to address visibility concerns (Section 2.4). Section 4.4 shows how to *capture* and later *reinstate* an execution level, allowing to address the concurrency issue described in Section 2.5. Section 4.5 defines a notion of control flow that is sensitive to execution levels, and shows how to override the default semantics given in Section 4.2. Finally Section 4.6 discusses how to address the issues related to exception handling, described in Section 2.6.

4.1. LAScheme Basics

LAScheme is an aspect-oriented extension to the higher-order functional programming language Scheme, following the design of AspectScheme [19], but considering only one kind of join points: function application.

A join point is composed of the function being applied, its arguments and the level of execution it is bound to. A join point is also bound to a *context*, which we explain below in Section 4.5. An aspect is defined by two functions: a pointcut function and an advice function. An aspect is deployed globally using the `deploy` primitive that takes a pointcut and an advice as arguments. A pointcut is a function that takes a join point as input and returns either false (`#f`) if it does not match, or a (possibly empty) list of context values exposed to the advice. A predefined `call` pointcut is provided as well as the logical pointcut combinators `&&`, `||` and `!`. An advice takes as parameters a `proceed` function, a list of context values (coming from the pointcut), and the arguments at the join point. Figure 2 illustrates programming in LAScheme; the code combines advice and pointcut loops, based on the examples of Section 2.1 and 2.2.

```
(define point-operation
   (λ (jp)
     (let ((p (car (args jp))))
      (if (and (Point? p) ((|| (call move) (call setX)
                               (call setY) (call to-string)
                               (call is-inside)) jp))
          '()
          #f))))

(define point-in-area
  (let ((area ...))
    (λ (jp)
      (let ((p (car (args jp))))
       (if (and (Point? p) (is-inside p area)) '() #f)))))

(define activity
  (λ (proceed ctx . args)
    (write "point active ")
    (write (to-string (car args)))
    (proceed args)))

(deploy (&& point-operation point-in-area) activity)
(define p (make-point 0 0))
(setX p 2)
```

Figure 2: Programming in LAScheme. The code combines the examples of Section 2.1 and 2.2.

4.2. Aspects and Levels: Default

Join Points. Upon creation, a join point is bound to a specific–and fixed–level of execution. When a function application is performed at level n, the corresponding join point is bound at level $n + 1$.

Aspects. Similar to join points, whenever evaluation of `deploy` happens at level n, the deployed aspect is bound at level $n + 1$. A deployed aspect is a triple composed of its level, pointcut and advice.

Weaving. When a join point is emitted, the weaver examines the deployed aspects to construct a woven function. If several aspects match the join point, the corresponding advices are chained such that calling `proceed` in an advice triggers evaluation of the next advice in the chain. The original computation is performed only when the *last* advice proceeds. Join points bound at level n can only be matched *by pointcuts whose aspect is also bound at that same level.* Both pointcut and advice evaluation is performed at the level at which the aspect is bound.

Avoiding Advice and Pointcut Loops. Figure 3 depicts the default evaluation of pointcuts and advice described so far, and how both advice and pointcut loops

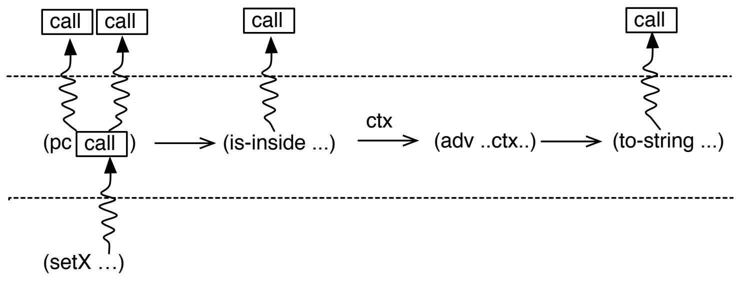

Figure 3: Running pointcut and advice at a higher level of execution. The diagram above depicts the evaluation of the code of Figure 2, using the default semantics for execution levels.

are avoided. The diagram corresponds to the code of Figure 2, evaluated in the default semantics of execution levels. Evaluation at base code (at level 0) generates join points at level 1 (the $\boxed{\text{call}}$ box), where aspects can potentially match and trigger advice.

Similarly the whole evaluation of pointcuts[9] and advices is done at level 1, thus the join points produced in the dynamic extent of these evaluations are generated at level 2. This ensures that the call of `is-inside` done during pointcut evaluation of `activity` is not seen at the same level as the call to `setX`. The same holds for the call to `to-string` in the advice. The default semantics therefore addresses both issues raised in Sections 2.1 and 2.2.

Proceed. As briefly explained in Section 2.3, an advice can `proceed` to the computation originally described by the join point. Naturally, the original computation clearly belongs to the *same level* as the original expression. This is fundamental, and is precisely why using control flow checks to discriminate advice execution fails. Base computation should remain base computation, no matter if some aspect applies to it or not, and no matter the advice kind. Using around advice (with `proceed`) rather than before advice should *not* change the status of the underlying computation.

Therefore, our default semantics ensures that the *last call* to `proceed` in a chain of advices triggers the original computation at the original level. Subsequently, join points generated by the evaluation of the original computation (level 0 in that case) are seen at the same level as before (level 1). This is shown on Figure 4, and addresses the issue raised in Section 2.3.

Aspects of aspects. The default semantics of computing pointcut and advice at a higher-level ensures that other aspects do not see these computations. As

[9]Observe that pointcut evaluation emits two join points. They correspond to the `Point?` predicate. In AspectJ, the `Point?` predicate would be performed by an `instanceof` check, which happens to not pertain to the join point model, so there is no risk of loops. In contrast, here it is just a function application, like the application of `is-inside`, and the `proceed` function. All these would lead to loops in AspectScheme, if `app/prim` were not used. With levels, `app/prim` is not needed in user programs.

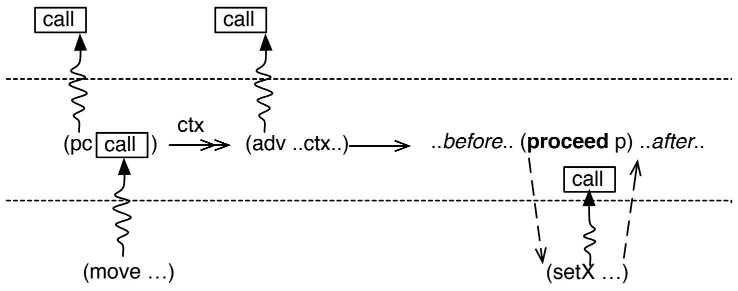

Figure 4: Proceeding to the original computation is done at the original level.

discussed in Section 2.4, this is the desired semantics to avoid interferences between aspects. For instance, using the `Activity` aspect should not affect the measurements performed by `FrequencyDisplay`.

However, this layering also implies that `Coalescing` cannot see the computation of `Activity`; therefore it cannot optimize the refreshing of the `Display`. In order to allow aspects to observe the activity of other aspects, while keeping the same default semantics, it is necessary to define aspects at higher levels. However this is not possible without language support to specify the level to which an aspect pertains.

4.3. Shifting Execution Levels

While installing aspects at higher levels is correct, it stays within the perspective of "aspects are meta". From a software engineering viewpoint, it also implies that at the time `Coalescing` is deployed, it is known that this aspect may be required at higher levels.

As we already mentioned before, AOP is not solely metaprogramming with syntactic sugar: the original idea is that advice is a piece of base-level code [53, 27]. In some cases, advice execution should be visible to aspects that observe base level execution. This approach is more compatible with the traditional view according to which "advices are base". From an engineering viewpoint, supporting advice as base code ensures that other aspects do not need to be explicitly deployed at a higher level, since they perceive that computation just like base computation.

Up and down. In order to reconcile both approaches, we introduce explicit level shifting operators in the language, such that a programmer can decide at which level an expression is evaluated. Level shifting is orthogonal to the pointcut and advice mechanism, and can be used to move any computation.

Figure 5 shows that shifting up an expression moves the computation of that expression a level above the current level. This implies that join points generated during the evaluation of that expression are visible one level above. Conversely, shifting an expression down moves the computation of that expression a level below the current level, as depicted on Figure 6.

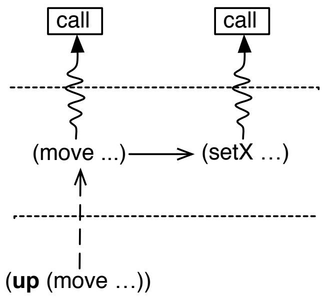

Figure 5: Shifting up.

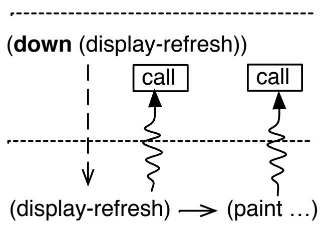

Figure 6: Shifting down.

Using `up` and `down`, it is possible to control where aspectual computation is performed, relative to the default semantics described in Section 4.2. One can also use these level-shifting operators to actually *deploy* aspects at a particular level.

Deploying aspects of aspects. In order to deploy an aspect that observes aspect computation at level 1, we can simply deploy it using the `up` level-shifting operator:

```
(up (deploy pc adv))
```

Assuming the expression is evaluated at level 0, `up` shifts evaluation to level 1, where the aspect deployment expression is then evaluated. This results in the aspect being deployed at level 2, thereby observing the computation of aspects standing at level 1. This addresses one part of the visibility issue discussed in Section 2.4. Note that it is possible to deploy the same pointcut and advice pair at different levels, in order to observe computation at multiple levels.

Shifting some aspect computation. One can use level-shifting operators directly within the definitions of pointcut and advice. In line with the example of Section 2.4, the following `activity` advice writes out the point object, refreshes the display, and `proceed`s.

```
(define activity
  (λ (proceed ctx . args)
    (write "point active")
    (write (car args))
    (down (display-refresh))
    (proceed args)))

(deploy point-in-area activity)
```

The advice uses the level-shifting operator `down` to move the computation of `display-refresh` down to the base level. This is depicted in Figure 6.

Exposing `display-refresh` as base computation using `down` allows another aspect, like `Coalescing`, to take effect and optimize that computation. This example illustrates how execution levels can be used to fully address the visibility issues of Section 2.4.

Note however that moving down a part of an aspect computation may potentially lead back to pointcut or advice loops; *e.g.* consider what would happen if we were to move down the computation of `is-inside` in the `point-in-area` pointcut.

4.4. Capturing Execution Levels

We now turn our attention to the issue of delayed aspect computation, such as in the concurrency example presented in Section 2.5. Consider that the `activity` advice defined previously schedules a logging task to be run by a separate timer thread. How can we recognize that the computation of that task relates to the advice execution?

In the model we have presented so far, functions run at the level at which they are *applied*. However, to track delayed advice execution we also need to define functions that execute at the level they were *defined*[10]. Intuitively, these approaches can be related to dynamic and static scoping respectively. This interpretation fits the notion that the execution level is a property of a flow of execution, which nevertheless can be captured and reinstated.

We therefore introduce a new kind of lambda abstraction, denoted $\lambda^\bullet$, called a level-capturing function. A $\lambda^\bullet$-abstraction is executed at the level of execution during which it was *evaluated to a closure value*[11].

```
(define activity
 (λ (proceed ctx . args)
   (schedule-task (λ•() (log-to-file (car args))))
   (proceed args)))
```

By defining the `activity` advice as above using a level-capturing function, we ensure that the call to the point structure performed by the timer thread when running the task is actually performed at *the same level as the advice* that originated it. This addresses the issue described in Section 2.5. Level-capturing functions are a generic construct that embodies the static scoping discipline for execution levels; beyond the concurrency scenario presented here, a static scoping discipline is helpful when designing a library that provides a set of functions that should be guaranteed to run at the same, predefined level.

4.5. Exploiting Execution Levels

Execution levels provide a certain amount of *structure* to computation, a structure that can be used to reason about the computation that is taking

[10]The idea of level-capturing functions is directly inspired by the reflective language Blond [15] (Section 3.2), which supports two different kinds of reflective procedures.

[11]Because the level of execution is a property of execution flow, a lexical definition using $\lambda^\bullet$ in a higher-order setting can yield closures bound to different levels.

place. We now extend the traditional notion of control flow to take levels into account.

Level-sensitive control flow. Certain pointcuts perform join point selection not only by looking at the current join point, but by looking at its *context*, which may include other join points. This is the case of `cflow` pointcuts, which inspect the current stack of execution[12]. It is important for these pointcuts to be able to distinguish between levels, in order to avoid conflation. Section 2 has illustrated the many downsides of a *conflating* control flow pointcut. As another example, consider an aspect that watches for a particular sequence of nested calls in the base computation. When observing the stack, it would be unfortunate for the aspect to consider join points that do not belong to base computation at all.

The stack of execution is reified as a chain of join points, each referencing its parent join point, denoting the surrounding pending application. Given a join point `jp`, `(parent jp)` returns its parent, and `has-parent?` tests whether a join point has a parent (only the root join point does not). Also, `(level jp)` returns the level at which join point `jp` occurs. It is straightforward to define a non-conflating control flow pointcut descriptor:

```
(define cflow
 (λ (pc)
  (λ (jp)
    (or (pc jp)
        ((cflowbelow pc) jp)))))

(define cflowbelow
 (λ (pc)
  (λ (jp)
    (and (has-parent/l? jp)
         ((cflow pc) (parent/l jp))))))
```

This mutually-recursive definition of `cflow` and `cflowbelow` is standard [19, 44]. The only modification needed to make these pointcut designators non-conflating is to use `has-parent/l?` and `parent/l`. These functions only find a parent join point if it occurs at the same level as the given join point.

Overriding the default semantics. As a final exercise with the practice of execution levels, let us see how to override the default semantics according to which *both* pointcuts and advices execute at the meta level. We want to easily deploy an aspect such that the original view holds: pointcuts at the meta level, and advice at the base level.

We can take advantage of advice as first-class functions, and define a `shift-down` higher-order function that takes a function `f` and returns a new function that applies `f` one level below:

[12]We do not consider state-based (as opposed to stack-based) implementation of control flow checks here [31]. It is straightforward to extend our argument to state-based `cflow`.

```
(define (shift-down f)
  (λ args (down (apply f args))))
```

However, simply deploying an aspect with `shift-down` as follows:

```
(deploy pc (shift-down adv))
```

would be incorrect. Indeed, multiple advices are chained together by means of `proceed`. As we have seen, in a higher-order aspect language, an advice is a function that receives, amongst other arguments, a `proceed` function used to either call the next advice, or to run the original computation, if it is the last advice in the chain. Therefore, simply shifting the execution level of one advice implies that subsequent advices also run at the modified level, and that the base computation runs potentially at a different level than where it originated.

Therefore, care must be taken to preserve levels appropriately. The following higher-order function `adv-shift-down` ensures that the execution levels are properly maintained by shifting the `proceed` function in the reverse direction: *i.e.* the advice body is shifted down, while the `proceed` function is shifted up with `shift-up` (defined similarly to the `shift-down` function above).

```
(define (adv-shift-down adv)
  (λ (proceed ctx . args)
    (let ((new-proc (shift-up proceed)))
      (down (apply adv (append (list new-proc ctx)
                               args))))))
```

It is now possible to depart from the chosen default semantics, *for a given aspect*, in order to support the original view according to which pointcuts are metalevel predicates and advice is base code. We can define a `deploy-aj` function as follows:

```
(define (deploy-aj pc adv)
  (deploy pc (adv-shift-down adv)))
```

4.6. Exception Conflation

The issues about exception handling presented in Section 2.6 are also caused by conflation, and as such we refer to them as the *exception conflation problem.* We show how to exploit the structure of execution levels to address exception conflation by *level-aware* exceptions and handlers. The idea is to make both handlers and exceptions sensitive to the level of execution, such that an exception bound at level n can only be caught by a handler bound at the same level. In LAScheme we reuse the standard exception handling mechanism of Scheme to implement this behavior. This mechanism features the `raise` and `with-handlers` forms to raise and handle exceptions, respectively.

Raising Exceptions. To raise level-aware exceptions we first define `raise/l`, which raises a compound exception comprising the actual exception value and the level of execution present when raising the exception:

```
(define (raise/l v)
  (raise (make-exn/l (current-level) v)))
```

We obtain the current level of execution applying `current-level`. Note that an exception carries a value `v`, which can be used for debugging purposes or, in general, for non-local transfer of information.

Handling Exceptions. To handle level-aware exceptions we use Scheme's macro system to define the `with-handlers/l` expression:

```
(with-handlers/l ((pred1 handler1) (pred2 handler2) ...) body)
```

This expression takes several pairs of predicates and handlers and a `body` expression. A predicate determines whether the exception is handled by the corresponding handler. In turn, a handler is a function that takes an exception as argument. To make handlers aware of the level of execution, each of them is bound to a *handler level*, which is defined as follows. If the handler is a level-capturing function bound at level n we use that value as the handler level. Otherwise we use the current level of execution, obtained applying `current-level`.

An exception raised during evaluation of `body` and bound to level m will be compared in ascending order against the handler levels of $\texttt{handler}_{i=1\dots}$. When the levels of the exception and the handler match, the corresponding predicate $\texttt{pred}_i$ is evaluated. If the predicate matches the exception, then $\texttt{handler}_i$ is evaluated with the exception as an argument.

In the default semantics of execution levels, an exception-handling advice can only capture exceptions raised by other advices applied to the same join point, but not exceptions from the original computation. This behavior follows the defensive design of execution levels, where programmers need not to be aware of potential interferences unless they explicitly want to.

Overriding the default semantics. The default semantics of level-aware exception handling can also be overridden using the `up` and `down` level-shifting operators, and level-capturing functions. To illustrate a situation where this is desirable, consider a simple exception-handling aspect that provides a default value in case the base computation that it advises fails with an exception:

```
(define (default-value value)
 (λ (proceed ctx . args)
   (with-handlers/l ((λ (x) #t) (down (λ• (x) value))
     (proceed args)))))
```

	avoid adv loops	avoid pc loops	discriminate aspect/base	visibility wrt other aspects	delayed aspect computation	avoid exn conflation
plain	no	no	-	partial [cannot hide]	-	no
`cflow` checks	yes	(no)	no [`proceed` conflation]	partial [cannot hide]	no	no
levels	yes	yes	yes	yes [higher or down]	yes[λ·]	yes

Figure 7: Benefits of execution levels to address the issues of Section 2.

To capture base-level exceptions, the level of the handler must match the level of exceptions potentially raised by the original computation, which is one level below. Therefore it suffices to use `down` to create a level-capturing handler function that will be bound to that level.

Observe that in contrast to the default semantics the handler will match only exceptions raised from base computation. Of course if an advice uses `down` to evaluate an expression, that evaluation will also be considered as base computation and its exceptions can be caught by the advice handler.

Conversely, a situation where it is desirable to raise exceptions one level above is to enhance a contract system such that contract-related exceptions cannot be caught by base handlers. We describe this application in Section 7.1.

4.7. Summary

To conclude this section, Figure 7 summarizes the benefits of execution levels compared to traditional/conflating control flow checks. The columns refer to the different issues described in Sections 2, in order. The first row describes the situation of "plain" AOP, that is, without using any defense against loops; while the second row describes the approach of using control flow checks.

As shown in the third row, introducing execution levels in an aspect-oriented language seamlessly addresses all the issues described. The possibility to shift up/down the `proceed` computation is fundamental to avoid the confusion raised by conflation. Similarly, and in a uniform way, the level-shifting operators allow to control the interactions between aspect and base exceptions and handlers.

Moreover, programmers by default are not exposed to any additional cognitive burden; average programmers are oblivious to execution levels. They only need to know and learn about them when facing a scenario requiring level shifting; that is, a scenario that requires specific visibility control between aspects, or delayed aspect computation.

5. Evaluation of Execution Levels in AspectJ Programs

To get a more practical grasp of the issues addressed by execution levels, as well as of the suitability of the default semantics chosen in this work, we have performed two studies of AspectJ code. We have manually inspected the aspect definitions in the famous textbook AspectJ in Action [30], by Laddad; as well as from the corpus of AspectJ projects used by Khatchadourian et al. [25].

In the book Laddad recognizes infinite recursion of advice as a common problem in AspectJ programs. He explains that the solution is "easy to implement" [30], and explains how to use the lexical condition `!within(Aspect)` in order to avoid infinite loops. This is the exact same solution that is explained in the AspectJ Programming Guide [5].

As explained in Section 2, the proposed solution of using `within` checks is not correct. It happens to work in many cases, but is not a reliable solution. Even when it works, it is at times not modular, creating unnecessary dependencies between aspect definitions. An example of this modularity issue can be observed in Listing 5.19 of AspectJ in Action[13], where a `within` check in a logging aspect has to be modified after using another aspect to make logging be indented[14].

It is surprising that the limitations of this pattern are not even discussed in these programming guides, and that pointcut loops are never mentioned. As we have seen in Section 2, the only "solution" to pointcut loops in AspectJ is to refactor the aspect moving the condition either to each associated advice (duplicating code), or to a separate method whose execution can be filtered. The other issues discussed in Section 2 are also not mentioned in these guides.

5.1. Study of a Textbook

We inspected the 61 aspect definitions provided in Parts 2 and 3 of AspectJ in Action [30], which are dedicated to applications; some definitions are progressively refined versions of the same aspect[15]. We found that almost a third of these aspects (19) use a `within` check to avoid infinite regression caused by advices. These aspects cover applications of logging, thread safety, and transaction management. The rest of the aspects do not need to protect themselves against infinite regression because their pointcuts are "narrow-enough".

A closer look at these 19 aspects, and at why the `within` check is sufficient for them, reveals an important implicit assumption in current programming guides: that advices do not perform any computation that is both advisable and *not* lexically within the advice definition. Indeed, the 19 aspects only invoke core Java libraries (*e.g.* string manipulation, streams, etc.), in which the standard AspectJ weaver is not capable of weaving aspects. This means that `within` checks would break if any of these libraries could be advised. For instance, in the area of dynamic analysis, it is fundamental to be able to advise the complete code base in order to collect meaningful metrics, so more powerful AspectJ weavers have been implemented, which can weave aspects into core libraries [51, 32]. Note also that the implicit assumption that aspects do not normally call advisable code contradicts the AOP premise according to which "advice is just another method" (recall Section 3.2). This assumption is not compatible with more symmetric models of AOP, as adopted for instance in

[13]We are working with the first edition of the book [30].

[14]To be exact, the indentation extension is done through a super aspect from which the logging aspect inherits, rather than a separate aspect; the issue is the same either way.

[15]Details of the inspection are available online: `http://pleiad.cl/research/scope/levels`

CaesarJ [4], Classpects [33] or EScala [24], where every object in the system can declare pointcuts to observe and react to the activity of other objects.

Finally, in our inspection of the 61 aspect definitions of AspectJ in Action, we found that only 2 aspect definitions would not work "out of the box" with the default semantics of execution levels. These two cases correspond to aspects of aspects (for instance, Listing 11.7 shows a logging aspect that is used to trace the behavior of both the base program and another aspect). They would require deploying aspects at a higher level.

This means that the extension of AspectJ with execution levels that we describe in Section 7.3 can directly support the entire set of examples in the book. Interestingly, almost all these aspects would work directly, without any modification; among these, a third could even be made simpler, by removing the `within` check (and would be more robust); only 2 examples would require actually knowing about levels, in order to deploy aspects appropriately.

Of course, we cannot but notice that AspectJ in Action has very little examples where multiple aspects are applied at the same time. The two examples that rely on aspects of aspects imply the need for deploying at specific levels. We intuit that in complex composition scenarios, knowledge about levels becomes more necessary, and the need for explicit level shifting may arise.

5.2. Study of a Corpus of Applications

In order to further study existing aspect-oriented software, we analyzed the corpus of 23 AspectJ projects used by Khatchadourian *et al.* in their pointcut rejuvenation study [25]. To the best of our knowledge, this is the largest AspectJ corpus, consisting of 496 aspects, in which we identify 1296 pointcut descriptors (PCDs)[16].

Like the study of AspectJ in Action, we manually inspected the aspect definitions and the PCDs in order to classify them according the whether or not they rely on aspect loops checks, and in this case, whether they only rely on static checks (`within`), dynamic checks (`cflow`), or both. The results over the 23 projects, in terms of aspects and pointcut designators, are presented in Figure 8.

They confirm that the issue of aspect loops is relevant, since nearly half of the projects (10 out 23) have to deal with them, amounting to 14% of all aspects, and 18% of all pointcut designators. Nearly all aspects that perform loops checks are "system wide" aspects (tracing, fault tolerance, contracts, error handling, persistence, distribution). The only exception we found is Quicksort*, where the check is done to avoid loops on recursive calls. Other aspects are narrow enough in their quantification that loops do not occur. Also, static checks are sufficient in many cases thanks to the assumption that advice code does not call into woven code, as discussed previously. Finally, we found that all 496 aspects would work well "out of the box", by removing all checks and relying on the default semantics of execution levels. Generally, these findings are con-

[16]To count pointcuts, we inlined named pointcuts used in other pointcut definitions.

Pointcut designators	#	%
no check	1011	78
abstract	52	4
	1063	82
static check	132	10
dynamic check	24	2
both	77	6
	233	18
TOTAL	1296	

Aspects	#	%
no check	428	86
	428	86
static check	34	7
dynamic check	14	3
both	20	4
	68	14
TOTAL	496	

Projects with checks (10/23): RacerAJ, Recovery Cache, Quicksort*, N-version, LawOfDemeter, HealthWatcher v6, Jakarta Cactus, mysql-connector-java, DCM, Contract4J.

Figure 8: Analysis of aspect loops checks in the AspectJ corpus.

sistent with those of the textbook study.

The conclusions of our inspection of AspectJ programs are, first that the problems addressed by execution levels are indeed present in existing aspect-oriented software. And second, that most of the affected code would work well "out of the box" in the default semantics of execution levels, even in many cases the code could be made simpler by removing control flow checks. Level shifting is useful in few cases, especially in presence of several aspects, as expected.

We also observe that system-wide dynamic analysis aspects, like Racer [9], confirm the need to treat the execution level as a dynamic property. These kind of aspects monitor all computation on a system and eventually need to perform base code operations to update their internal state, which can again be observed by them; more generally, the computation done by any aspect ends up being observable, even if it only calls some standard libraries. The default semantics of execution levels address this situation directly, in contrast to stratified aspects (Section 3.1) where aspects must be implemented as closed worlds. Interestingly, the first AspectJ weaver for comprehensive dynamic analysis [51] uses an ad hoc bootstrap phase flag, which is a specific case of execution level [45, 32].

The following section dives into a formal study of execution levels and their properties. Section 7 then describes several existing implementations of execution levels and their applications.

6. Semantics and formal properties

We now turn to the formal semantics of higher-order aspects with level shifting. We introduce a core language extended with execution levels and aspect weaving. In this section we only present the essential elements, and skip the obvious. The complete formal description of the language is provided online (see Section 6.9). In Section 6.10 we summarize the most important formal properties of execution levels, along with intuitive sketches of their proofs. The complete proofs are described in Appendix A. By convention, user-visible syntax is written in **`bold typewriter`** font, while internal forms added only for the sake of the semantics are written in `plain typewriter` font.

6.1. Core Language

Figure 9 presents the user-visible syntax of the core language, *i.e.* without aspects nor execution levels. The language is a Scheme-like language with booleans, numbers and lists, and a number of primitive functions to operate on these. The only expressions considered are multi-arity function application, and **`if`** expressions. The full language includes also sequencing (**`begin`**) and binding (**`let`**) expressions for convenience. The notation $X \cdots$ denotes zero or more occurrences of the pattern X.

$$\begin{array}{rlcl}
\mathit{Value} & v & ::= & (\lambda(x\cdots)\ e) \ \mid\ n \ \mid\ \#t \ \mid\ \#f \ \mid\ (\mathbf{\texttt{list}}\ v\cdots) \\
 & & \mid & \mathit{prim} \ \mid\ \mathit{unspecified} \\
 & \mathit{prim} & ::= & \mathbf{\texttt{list}} \mid \mathbf{\texttt{cons}} \mid \mathbf{\texttt{car}} \mid \mathbf{\texttt{cdr}} \mid \mathbf{\texttt{empty?}} \mid \mathbf{\texttt{eq?}} \\
 & & \mid & \mathbf{\texttt{pred}} \mid \mathbf{\texttt{succ}} \mid \ldots \\
\mathit{Expr} & e & ::= & v \ \mid\ x \ \mid\ (e\ e\cdots) \ \mid\ (\mathbf{\texttt{if}}\ e\ e\ e) \\
 & v & \in & \mathscr{V}, \text{ the set of values} \\
 & n & \in & \mathscr{N}, \text{ the set of numbers} \\
 & \mathit{list} & \in & \mathscr{L}, \text{ the set of lists} \\
 & x & \in & \mathscr{X}, \text{ the set of variable names} \\
 & e & \in & \mathscr{E}, \text{ the set of expressions} \\
\mathit{EvalCtx} & E & ::= & [\,] \ \mid\ (v\cdots\ E\ e\cdots) \ \mid\ (\mathbf{\texttt{if}}\ E\ e\ e)
\end{array}$$

Figure 9: Syntax of the core language.

We describe the operational semantics of our language via a reduction relation $\hookrightarrow$, which describes evaluation steps:

$$\hookrightarrow: \mathscr{L} \times \mathscr{J} \times \mathscr{E} \to \mathscr{L} \times \mathscr{J} \times \mathscr{E}$$

An evaluation step consists of an execution level $l \in \mathscr{L}$ (initially 0), a join point stack $J \in \mathscr{J}$ and an expression $e \in \mathscr{E}$. The reduction relation takes a level, a stack, and an expression and maps this to a new evaluation step. The reduction rules for the core language are standard and not presented here. In the following we describe the semantics of execution levels, join points, aspects and their deployment, as well as the weaving semantics. Then we describe the semantics of level-aware exception handling.

$$\begin{array}{llcl}
\textit{Expr} & e & ::= & \ldots \mid (\textbf{up}\ e) \mid (\textbf{down}\ e) \mid (\texttt{in-up}\ e) \mid (\texttt{in-down}\ e) \\
\textit{EvalCtx} & E & ::= & \ldots \mid (\texttt{in-up}\ E) \mid (\texttt{in-down}\ E)
\end{array}$$

$$\begin{array}{rcll}
\langle l, J, E[(\textbf{up}\ e)]\rangle & \hookrightarrow & \langle(\textbf{succ}\ l), J, E[(\texttt{in-up}\ e)]\rangle & \textsc{InUp} \\
\langle l, J, E[(\texttt{in-up}\ v)]\rangle & \hookrightarrow & \langle(\textbf{pred}\ l), J, E[v]\rangle & \textsc{OutUp} \\
\langle l, J, E[(\textbf{down}\ e)]\rangle & \hookrightarrow & \langle(\textbf{pred}\ l), J, E[(\texttt{in-down}\ e)]\rangle & \textsc{InDown} \\
\langle l, J, E[(\texttt{in-down}\ v)]\rangle & \hookrightarrow & \langle(\textbf{succ}\ l), J, E[v]\rangle & \textsc{OutDown}
\end{array}$$

Figure 10: Shifting execution levels.

6.2. Execution Levels

The language supports explicit execution level shifting forms, **up** and **down** (Figure 10). Correspondingly, there are two internal (*i.e.* not user-visible) marker expressions, `in-up` and `in-down` used to keep track of the level counter. When encountering an **up** expression, the level counter is increased, and an `in-up` marker is placed in the execution context (InUp). When the nested expression has been reduced to a value, the `in-up` mark is disposed, and the level counter is decreased (OutUp). Evaluation of a **down** expression is done similarly (see rules InDown and OutDown).

6.3. Join Points

We follow Clifton and Leavens [12] in the modeling of the join point stack (Figure 11). The join point stack J is a list of *join point abstractions* j, which are tuples $\lceil l, k, v, v\cdots\rceil$: the execution level of occurrence l, the join point kind k, the applied function v, and the arguments $v\cdots$. An interesting benefit of using execution levels is that it is not necessary anymore to introduce advice execution join points to avoid advice loops, or pointcut execution join points to avoid pointcut loops. Pointcut and advice evaluation are normal function applications, that just happen to occur at a higher level. For simplicity and conciseness, we only consider call join points.

In order to keep track of the join point stack in the semantics we introduce two internal expression forms: `jp` j introduces a join point, and (`in-jp` e) keeps track of the fact that execution is proceeding under a given dynamic join point. The definition of the evaluation context is updated accordingly (Figure 11).

A join point abstraction captures all the information required to match it against pointcuts, as well as to trigger its corresponding computation when necessary. For instance, consider the reduction rule for `call` join points (Figure 11, App). The rule specifies that when a function is applied to a list of arguments, the expression is reduced to a `jp` expression with the definition of the corresponding join point, which embeds the execution level at which it is visible, (**succ** l), its kind `call`, the applied function, and the values passed to it. A later rule (Weave, explained below) pushes the thus created join point to the

$$
\begin{array}{rrcl}
 & J & ::= & j + J \ \mid \ \epsilon \\
 & j & ::= & \lceil l, k, v, v \cdots \rceil \\
 & k & ::= & \texttt{call} \ \mid \ \ldots \\
 & l & \in & \mathscr{N} \\
 & J & \in & \mathscr{J}\text{, the set of join point stacks} \\
\mathit{Expr} & e & ::= & \ldots \ \mid \ \texttt{jp}\ j \ \mid \ (\texttt{in-jp}\ e) \\
\mathit{EvalCtx} & E & ::= & \ldots \ \mid \ (\texttt{in-jp}\ E)
\end{array}
$$

$$\langle l, J, E[((\lambda(x \cdots)\ e)\ v \cdots)]\rangle \ \hookrightarrow \langle l, J, E[\texttt{jp}\ j]\rangle \qquad \textsc{App}$$
$$\text{where } j = \lceil(\textbf{succ}\ l), \texttt{call}, (\lambda(x \cdots)\ e), v \cdots\rceil$$

$$\langle l, j + J, E[\texttt{in-jp}\ v]\rangle \hookrightarrow \langle l, J, E[v]\rangle \qquad \textsc{OutJp}$$

Figure 11: Join points: stack, creation and disposal.

stack J, marking the expression with in-jp, and then triggers weaving. Popping a join point from the stack is done by the OutJp rule, when the expression under a dynamic join point has been reduced to a value.

6.4. Aspects and Deployment

As described in Figure 12, an aspect is a tuple $\langle l, pc, adv\rangle$ where l denotes the execution level at which it stands, *pc* is the pointcut and *adv* the advice (both first-class functions). More precisely, a pointcut is a function that takes a join point stack as input and produces either $\#f$ if it does not match, or a (possibly empty) list of context values exposed to the advice. Higher-order advice is modeled as a function receiving first a function to apply whenever the advice wants to proceed, a list of values exposed by the pointcut, and the arguments passed at the original join point [18, 19].

An aspect environment $\mathscr{A}$ is a set of such aspects. An aspect is deployed with a **deploy** expression (added as a primitive to the language, see Figure 12). To simplify our reduction semantics, in this section we have not included the aspect environment as part of the description of an evaluation step. Rather, we simply "modify" the global aspect environment $\mathscr{A}$ upon aspect deployment[17] (see rule Deploy). Also note that we do not model the different scoping strategies of AspectScheme here—we restrain ourselves to deployment in a global aspect environment. For more advanced management of aspect scoping and aspect

[17]The complete semantics we provide explicitly includes the aspect environment in the program configuration (Section 6.9).

$$\begin{array}{rlcl} \textit{Aspects} & \mathscr{A} & = & \{\langle l_i, pc_i, adv_i\rangle \mid i = 1, \ldots, |\mathscr{A}|\} \\ \textit{Pointcut} & pc & \in & \mathscr{J} \rightarrow \{\#f\} \cup \mathscr{L} \\ \textit{Advice} & adv & \in & (\mathscr{V}^* \rightarrow \mathscr{V}) \times \mathscr{L} \times \mathscr{V}^* \rightarrow \mathscr{V} \end{array}$$

$$prim ::= \ldots \mid \texttt{deploy}$$

$$\langle l, J, E[(\texttt{deploy}\ v_{pc}\ v_{adv})]\rangle \hookrightarrow \langle l, J, E[\textit{unspecified}]\rangle \qquad \textsc{Deploy}$$
$$\text{and } \mathscr{A} = \{\langle(\texttt{succ}\ l), v_{pc}, v_{adv}\rangle\} \cup \mathscr{A}$$

Figure 12: Aspects and deployment (global environment $\mathscr{A}$).

$$\begin{array}{rlcl} \textit{Expr} & e & ::= & \ldots \mid (\texttt{app/prim}\ e\ e \cdots) \\ \textit{EvalCtx} & E & ::= & \ldots \mid (\texttt{app/prim}\ v \cdots\ E\ e \cdots) \end{array}$$

$$\langle l, J, E[(\texttt{app/prim}\ (\lambda(x \cdots)\ e)\ v \cdots)]\rangle \hookrightarrow \langle l, J, E[e\{v \cdots / x \cdots\}]\rangle \quad \textsc{AppPrim}$$

Figure 13: Primitive application.

environments, see [40]. When an aspect is deployed, it is annotated with the execution level at which it stands. This means that, when executing at level l, (**deploy** p a) deploys the aspect such that it sees join points at level (**succ** l) (which in turn denote computation of level l).

6.5. Primitive applications

The primitive application form, `app/prim`, described in Figure 13 denotes an application that does not trigger a join point: rule AppPrim simply performs the classical β_v reduction. `app/prim` is used to hide "administrative" applications, *i.e.* the initial application of the composed advice chain, and its recursive applications. Note that contrary to AspectScheme, `app/prim` is *not* in user-visible syntax, thanks to execution levels.

6.6. Level-capturing functions

Figure 14 extends the semantics of the language with level-capturing functions. There is a new syntactic form to define a level-capturing function, $\lambda^\bullet$, and a new value form, called an *instantiated* level-capturing function and noted λ^l, which represents a function that is always executed at level l. The capture of the level is described by the rule Capture. Similar to regular function application, applying a level-capturing function emits a join point one level above current computation (rule App$^\bullet$). It is primitive function application using `app/prim` which distinguishes between regular and level-capturing functions,

$$\begin{array}{rcll}
\textit{Expr} & e & ::= & \ldots \mid (\lambda^{\bullet}(x\cdots)\ e) \mid (\texttt{in-shift}(l)\ e) \\
\textit{Value} & v & ::= & \ldots \mid (\lambda^{l}(x\cdots)\ e) \\
\textit{EvalCtx} & E & ::= & \ldots \mid (\texttt{in-shift}(l)\ E)
\end{array}$$

$$\langle l, J, E[(\lambda^{\bullet}(x\cdots)\ e)]\rangle \hookrightarrow \langle l, J, E[(\lambda^{l}(x\cdots)\ e)]\rangle \qquad \textsc{Capture}$$

$$\langle l, J, E[((\lambda^{l_1}(x\cdots)\ e)\ v\cdots)]\rangle \hookrightarrow \langle l, J, E[\texttt{jp}\ j]\rangle \qquad \textsc{App}^{\bullet}$$
$$\text{where } j = \lceil(\textbf{succ}\ l), \texttt{call}, (\lambda^{l_1}(x\cdots)\ e), v\cdots\rceil$$

$$\langle l, J, E[(\texttt{app/prim}\ (\lambda^{l_1}(x\cdots)\ e)\ v\cdots)]\rangle \qquad \textsc{AppShift}$$
$$\hookrightarrow \langle l_1, J, E[(\texttt{in-shift}(l)\ e\{v\cdots/x\cdots\})]\rangle$$

$$\langle l_1, J, E[(\texttt{in-shift}(l)\ v)]\rangle \hookrightarrow \langle l, J, E[v]\rangle \qquad \textsc{Shift}$$

Figure 14: Level-capturing functions.

using rule AppPrim for the former, and rule AppShift for the latter. In order to keep track of the level shifting incurred by applying (using `app/prim`) an instantiated level-capturing function, there is an extra expression `in-shift` that keeps track of the level at which such a function is originally applied (Rule AppShift). This is necessary in order to be able to restore the original level once the execution of the level-capturing function has finished (Rule Shift). This level-capturing mechanism will be used in the weaving process (described in the next section) to proceed with the original computation at the right execution level.

6.7. Weaving

We now turn to the semantics of aspect weaving. The Weave rule describes the process (Figure 15). A `jp` expression reduces to an `in-jp` expression, and the join point is pushed onto the stack. The inner expression of `in-jp` is the application, one execution level up, of the list of advice functions that match the given join point, properly chained together, to the original arguments.

The weaving process is closely based on that described by Dutchyn; it differs only in that we deal with execution levels. The W metafunction recurs on the global aspect environment $\mathscr{A}$ and returns a composed procedure whose structure reflects the way advice is going to be executed.

For each aspect $\langle l_i, pc_i, adv_i\rangle$ in the environment, W first checks whether the aspect is at the same execution level as the join point, *i.e.* if the aspect can actually "see" the join point. If so, it applies its pointcut pc_i to the current join point stack. If the pointcut matches, it returns a list of context values, c. W then returns a function that, given the actual join point arguments, applies the advice adv_i. All this process is parametrized by the function to proceed with, p. This function is passed to the advice, and if an aspect does not apply,

$$\langle l, J', E[\texttt{jp}\ \lceil(\textbf{succ}\ l), k, v_\lambda, v\cdots\rceil]\rangle \qquad\qquad \textsc{Weave}$$

$$\hookrightarrow \langle l, J, E[(\texttt{in-jp}\ (\textbf{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v\cdots)))]\rangle$$

$$\text{where } J = \lceil(\textbf{succ}\ l), k, v_\lambda, v\cdots\rceil + J'$$

$$\begin{array}{rcl}
W[\![0]\!]_J & = & P[\![\lceil(\textbf{succ}\ l), k, v_\lambda, v\cdots\rceil]\!] \\
W[\![i]\!]_J & = & (\texttt{app/prim}\ (\lambda(p) \\
 & & \quad (\textbf{if}\ (\textbf{eq?}\ l_i\ (\textbf{succ}\ l)) \\
 & & \quad\quad (\textbf{let}\ ((c\ (pc_i\ J))) \\
 & & \quad\quad\quad (\textbf{if}\ c \\
 & & \quad\quad\quad\quad (\lambda(x\cdots)\ (adv_i\ p\ c\ x\cdots)) \\
 & & \quad\quad\quad\quad p)) \\
 & & \quad\quad p)) \\
 & & \quad W[\![i-1]\!]_J)
\end{array}$$

$$P[\![\lceil(\textbf{succ}\ l), \texttt{call}, v_\lambda, v\cdots\rceil]\!] \quad = \quad (\lambda^l(x\cdots)\ (\texttt{app/prim}\ v_\lambda\ x\cdots)))$$

Figure 15: Aspect weaving.

then W simply returns this function. The base case, $W[\![0]\!]_J$ corresponds to the execution of the original computation, $P[\![\lceil(\textbf{succ}\ l), \texttt{call}, v_\lambda, v\cdots\rceil]\!]$ (here we only define P for join points of kind `call`). Intuitively, P corresponds to the last application of **`proceed`** in a chain of advice; in the following we use *last-proceed* to refer to $P[\![jp]\!]$, for some jp. Note that the original computation is performed by using a level-capturing function instantiated at the original execution level (λ^l). This reflects the fact that, while pointcuts and advices run at an upper level, the original function runs at its original level of application.

The Weave rule uses the primitive application form, `app/prim` (Section 6.5). `app/prim` is also necessary to eventually perform the original function application, when all aspects (if any) have proceeded. Also note that in W, the pointcut and advice functions are applied using a standard function application.

6.8. Level-Aware Exceptions

To address the issue of exception conflation we now introduce the semantic rules for a level-aware exception handling mechanism. Without loss of generality, we introduce the (**`try`** e **`with`** h) expression consisting of a single exception handler h that is unconditionally applied to exceptions raised from evaluation of the protected expression e (in contrast to the multiple predicates approach described in Section 4.6, or multiple type predicates as in Java).

Figure 16 describes the additional syntax and reduction rules. We extend the user-visible syntax with the **`raise`** and **`try-with`** expressions and their

$$
\begin{array}{rcll}
\textit{Expr} & e & ::= & \cdots \mid (\textbf{raise}\ e) \mid (\textbf{try}\ e\ \textbf{with}\ e) \\
\textit{Exception} & ex & ::= & (\textit{exn}\ l\ v) \\
\textit{EvalCtx} & E & ::= & \cdots \mid (\textbf{raise}\ E) \mid (\textbf{try}\ E\ \textbf{with}\ e) \\
 & & & \mid (\textbf{try}\ ex\ \textbf{with}\ E)
\end{array}
$$

$$
\begin{array}{ll}
\langle l, J, E[(\textbf{raise}\ v)]\rangle \hookrightarrow \langle l, J, E[(\textit{exn}\ l\ v)]\rangle & \textsc{RaiseCapture} \\
\langle l, J, E[(\textbf{raise}\ (\textit{exn}\ l_1\ v))]\rangle \hookrightarrow \langle l, J, E[(\textit{exn}\ l_1\ v)]\rangle & \textsc{Reraise} \\
\langle l, J, E[(\texttt{in-up}\ (\textit{exn}\ l_1\ v))]\rangle & \textsc{OutUpEx} \\
\qquad \hookrightarrow \langle (\textbf{pred}\ l), J, E[(\textit{exn}\ l_1\ v)]\rangle & \\
\langle l, J, E[(\texttt{in-down}\ (\textit{exn}\ l_1\ v))]\rangle & \textsc{OutDownEx} \\
\qquad \hookrightarrow \langle (\textbf{succ}\ l), J, E[(\textit{exn}\ l_1\ v)]\rangle & \\
\langle l_1, J, E[(\texttt{in-shift}(l)\ (\textit{exn}\ l_2\ v))]\rangle & \textsc{ShiftEx} \\
\qquad \hookrightarrow \langle l, J, E[(\textit{exn}\ l_2\ v)]\rangle & \\
\langle l, J, E[(\textbf{try}\ v\ \textbf{with}\ e)]\rangle \hookrightarrow \langle l_1, J, E[v]\rangle & \textsc{TryV} \\
\langle l, J, E[(\textbf{try}\ (\textit{exn}\ l\ v)\ \textbf{with}\ (\lambda(x\cdots)\ e)]\rangle & \textsc{HndEx1} \\
\qquad \hookrightarrow \langle l, J, E[((\lambda(x\cdots)\ e)\ v)]\rangle & \\
\langle l, J, E[(\textbf{try}\ (\textit{exn}\ l_1\ v)\ \textbf{with}\ (\lambda(x\cdots)\ e))]\rangle & \textsc{HndProp1} \\
\qquad \hookrightarrow \langle l, J, E[(\textit{exn}\ l_1\ v)]\rangle \text{ where } l_1 \neq l & \\
\langle l, J, E[(\textbf{try}\ (\textit{exn}\ l_1\ v)\ \textbf{with}\ (\lambda^{l_1}(x\cdots)\ e)]\rangle & \textsc{HndEx2} \\
\qquad \hookrightarrow \langle l, J, E[((\lambda^{l_1}(x\cdots)\ e)\ v))]\rangle & \\
\langle l, J, E[(\textbf{try}\ (\textit{exn}\ l_1\ v)\ \textbf{with}\ (\lambda^{l_2}(x\cdots)\ e))]\rangle & \textsc{HndProp2} \\
\qquad \hookrightarrow \langle l, J, E[(\textit{exn}\ l_1\ v)]\rangle \text{ where } l_1 \neq l_2 &
\end{array}
$$

Figure 16: Exception handling extensions.

respective evaluation contexts. We also define the *exn* normal form that contains a value and is annotated with the level at which the exception was raised. Evaluating (**raise** v) at level l creates an ($exn\ l\ v$) value bound to that level (rule RAISE).

Most of the rules for exception propagation through the language expressions are standard, and we omit them. We focus on propagation through a **raise** expression (rule RERAISE), in which case no new exception is created; instead the exception keeps propagating. In addition we focus on propagation through the level-shifting expressions `in-up`, `in-down` and `in-shift` (rules OUTUPEX, OUTDOWNEX and SHIFTEX). These rules ensure that when an exception is propagating through the call stack, the resulting level of execution after propagation is the same as if the original expression would have returned a value. In Section 6.10 we show that this is crucial to preserve the guarantees of execution levels in presence of exceptions.

If the protected expression yields a value, the handler is not evaluated and program reduction continues (rule TryV). Otherwise it means an exception is raised from evaluation of the protected expression. Essentially, the exception propagates if the handler is not bound at the same level of the exception; and the handler is evaluated with the exception as argument if the levels match. We distinguish the case where the handler is a regular function (rules HndEx1 and HndProp1), and the case where the handler is a level-capturing function (rules HndEx2 and HndProp2). Application of both kind of handlers correspond to function applications that will emit join points.

6.9. Executable Model

We have mechanized the complete semantics of our language using PLT Redex, a domain-specific language for specifying executable reduction semantics [20]. The full definition along with a test suite is available at the execution levels website: `http://pleiad.cl/research/scope/levels`.

The Redex model explicitly manages the aspect environment (ignored in the reduction rules of this paper) and adopts a representation of function values that includes a unique identifier, in order to be able to do reference equality of functions (used in pointcuts).

6.10. Avoiding Loops with Execution Levels

We now analyze what kind of loops can be avoided with the default semantics of execution levels. We present our results relying on an intuitive understanding of the semantics described above, and defer all formal definitions and proofs to Appendix A.

First observe that certain loops that occur due to aspects are not avoidable with execution levels. These cases arise from interactions between advice and base code, for instance due to using **`proceed`** with modified arguments, shared state, etc. These interactions can be hard to track down, especially in a higher-order setting. For instance, the following program is subject to a loop because of a subtle interaction between the base program and the advice:

```
(define (f b)
  (if b
      (f false)
      false))

(deploy (call f)
        (λ (proceed ctx . args)
          (proceed true)))
```

When `f` is applied, the advice intercepts the application and proceeds with `true`, which makes `f` call itself recursively. The recursive invocation is again advised, with the same consequence. Even though `f` is a function that terminates and always returns `false`, in the presence of the aspect it diverges.

Aspect Loops We characterize the kinds of loops that are avoided by execution levels as *aspect loops.* Intuitively, from the examples of Sections 2.1 and 2.2, an aspect loop can be defined as: "a loop that happens when an aspect matches a join point that is coming from its own activity (either pointcut or advice)".

Aspect loops arise in AspectJ because advice and base computation are conceptually at the same level. The same situation can happen in execution levels if advice uses **`down`** or applies a level-capturing function bound to a level lower than that of the advice. However, because our objective is to validate the defensive design of the default semantics of execution levels, we consider only programs that do not perform explicit level-shifting operations, which we call *level-agnostic* programs.

Level-agnostic expression An expression e is *level-agnostic* if it is a user-visible term that does not contain **`down`**, **`up`**, or level-capturing functions. This definition only considers programs that can be expressed with user-visible syntax to rule out degenerated cases not available to the programmer.

Theorem 6.1 (No Aspect Loops). *Let e be a level-agnostic term. If all aspects in the environment are also level-agnostic, the reduction of e is free of aspect loops.*

In other words, programs written in the default semantics of execution levels are free of aspect loops. The intuitive structure of the proof is as follows: if the level of execution is tracked during evaluation and it is only incremented and decremented implicitly by the weaver at appropriate points of execution, and if evaluation of advice depends on that level and the fixed level of advice, then no loop can occur via advice invocation because an advice cannot trigger computation (other than **`proceed`**) that will emit join points below the level of that advice.

Observe that a less strict version of the theorem, which allows to use **`up`**, also holds because by definition an advice cannot see computation at an upper level. We prefer the stricter version because it holds on programs written in the default semantics, where programmers do not need to be aware of levels.

Exceptions and Loops. If execution levels are combined with an exception-handling mechanism that does not take the level into account to propagate exceptions, then Theorem 6.1 does not hold. Consider as a counter-example the following program:

```
(define (log name value)
  (if (logger-exists? name)
      (write-to-logger (get-logger name) value)
      (raise "Logger not found")))

(define (log-exn-adv (λ (proceed ctx . args)))
  (try (proceed args) with (λ (exn) (log "Log" exn))))

(deploy (call log) log-exn-adv)
(log "Log" 10)
```

Assume a logging infrastructure with access to string-named logs through the `log` function (similar to the Java Logger API), such that an exception is raised if non-existing log name is given as argument. Consider now the `log-exn-adv` advice that implements a default exception handler that logs the occurrence of exceptions.

If by some programming error a logger named `Log` does not exists (*e.g.* it was not initialized properly, or the developer misspelled the name), then the program will enter into an aspect loop. This happens because although the advice is evaluated at level 1, the handler catches an exception from the base program, at level 0.

The reason is that a last-**proceed** application shifts the level of execution to that of the original computation, in this case 0, but because standard exception propagation rules will not shift the level back, the exception will propagate at level 0 and will trigger evaluation of the handler also at level 0. Then, evaluation of the handler triggers again the same aspect and also raises an exception, leading to an infinite loop. Observe that the problem happens even when using level-aware exception handlers, as those described in rules HndEx1 and HndEx2, because it will be the case that the current level will match the level bound to the exception.

Therefore we remark that the level-aware propagation rules of Section 6.8 are crucial to retain the result of Theorem 6.1 in presence of exceptions, whereas the level-aware handlers avoid the issue of exception conflation (Section 2.6).

The Value of Formalization. Although the proof of Theorem 6.1 appears to be "intuitively obvious", its development was valuable. First, it forced us to precisely define the kinds of loops avoided by execution levels, in contrast to the more general (and vague) claim that was made in [43]. In addition, this formal endeavor led us to discover an actual error in the original semantics.

The original formulation of levels uses **down**—instead of a level-capturing function—to define the original computation $P[\![\cdot]\!]$ in rule Weave. To see why this is not correct, consider the following advice:

```
(define (adv (proceed ctx . args))
  (up (proceed args)))
```

Using **down** in the semantics, the original computation would incorrectly execute at level 1, instead of level 0. The problem is that the level of execution when applying a last-proceed function is not necessarily (**succ** l). Therefore, using **down** does not guarantee that the advised computation always runs at its original level l, conversely to what is expected.

We spotted the problem when considering programs that pass **proceed** as argument to higher-order functions. While the original executable semantics were implemented and tested, we did not push the higher-order cases far enough. In fact, all the motivating examples involve only two levels, thus using **down** appeared to be correct. This is yet another illustration of the inherent limitations of test-based validation, as opposed to formal proofs.

7. Execution Levels in Practice

We report on three practical implementations of execution levels in the context of Scheme, JavaScript and Java. The objective of this section is to highlight different implementation techniques and applications of execution levels. The languages we describe represent different points in the design space, adopting specific tradeoffs with respect to flexibility versus efficiency.

7.1. LAScheme

The formal semantics presented in Section 6 are based on the semantics of AspectScheme by Dutchyn and colleagues [19, 18]. AspectScheme is an aspect-oriented extension of Racket (previously called PLT Scheme), implemented with macros. We developed our own variant of AspectScheme with execution levels, called LAScheme[18], briefly described in Section 4.1. LAScheme is the original artefact in which execution levels were first implemented and experimented with. Like AspectScheme, LAScheme is implemented as a Racket language module. LAScheme simplifies AspectScheme by removing advice execution join points and by not exposing `app/prim` to users. Execution levels are modeled using Racket *parameters*, which are thread- and continuation-safe dynamic bindings.

Computational contracts. This is the first and major experiment using LAScheme so far. Computational contracts are an extension of the higher-order contracts of Findler and Felleisen [23], in which contracts can not only check the input-output behavior of functions and their higher-order parameters, but can also check properties of the actual computation that derives from applying these functions [36]. The kinds of properties that can be expressed as computational contracts are very varied, for instance: not accessing the file system, not performing side effects, applying (or not) a given function, following (or not) a given protocol as expressed by a finite state machine, etc. For instance, one can specify that a library function accepts as parameter any function provided that, if it opens a file, it ought to close it before returning.

Computational contracts are integrated (and can be composed) with the standard input-output contracts of Racket. Execution levels are helpful in this

[18]Available online: `http://pleiad.cl/research/scope/levels`

contract system, because they make it possible to differentiate contract computation from base computation and therefore to choose between different contract semantics.

7.2. AspectScript

AspectScript is a language for expressive aspect-oriented programming in JavaScript, which takes full advantage of the higher-order functional programming features of JavaScript [47]. AspectScript was developed concurrently with the execution levels proposal and is the second language to feature execution levels, after LAScheme. In AspectScript, pointcuts and advices are standard JavaScript functions—therefore, levels are crucial to avoid infinite regression. In addition to execution levels, the language supports reentrancy control [39]—which ensures that shifting down computation cannot provoke loops—and expressive scoping of aspects using scoping strategies [40, 41]. Aspect weaving in AspectScript is similar to aspect weaving in LAScheme. However, because JavaScript lacks a macro system, reification of join points is achieved through an ad-hoc source-to-source transformation.

The implementation of levels in AspectScript differs only slightly from the LAScheme implementation discussed previously. First, JavaScript does not support dynamic variables like Racket parameters; the current execution level is implemented as a global variable that is mutated upon level shifting, within a `try/finally` block to ensure that the current level is always consistent. Second, instead of using a single aspect environment where all aspects reside, AspectScript uses one aspect environment per level. At each join point, a quick check is performed to determine if the upper-level environment is empty, in which case reification is avoided altogether. The benefit of this implementation scheme is relative to how much computation is performed in advices, and to the way aspects are deployed at different levels.

Performance and Applications. The primary design goal of AspectScript is expressiveness, not performance. Apart from the per-level check for empty aspect environment described above, no optimizations have been developed. In particular, partial evaluation techniques have not yet been applied. In a benchmark consisting of transforming the jQuery library and running all its (CPU-intensive) tests, Toledo *et al.* [47] report a 5.7x overhead just for generating join points (3.4 million join points), and 13.3x overhead for a global aspect that matches all join points (and just proceeds). This considerable overhead does not prevent this first implementation of AspectScript to be applicable in certain non-computationally-intensive scenarios, in particular with web-based, interactive applications. AspectScript is used to implement the ZAC access control library for JavaScript [49], a fully modular implementation of Java-like stack-based access control [48], which has been proven secure [50].

7.3. AspectJ

To assess whether or not execution levels can be efficiently implemented, the first author collaborated with Walter Binder and his group in order to address the practical and efficient integration of execution levels in AspectJ, and to study their application to black-box composition of dynamic analysis aspects.

7.3.1. Language Design and Implementation

AspectJ does not support dynamic deployment of aspects, as opposed to LAScheme and AspectScript. Therefore, extending AspectJ with levels includes augmenting the static specification of aspects with the levels at which they are deployed. The weaver, called MAJOR2 [45], uses the standard AspectJ compiler as a black-box; once aspects and application classes are compiled, MAJOR2 supports the *static* specification of level deployment. MAJOR2 also makes it possible to deploy the same aspect instance at different levels.

The design of levels for AspectJ was guided by the fact that analysis aspects have only *augmentation* advice (the original computation always executes) and are *independent* (they do not write to fields that others may access) [34]. This means that the default semantics of execution levels makes sense and that, beyond deploying at different levels, there is no need for the explicit level-shifting operators `up` and `down` in the language. Also, because dynamic analyses have to deal with multi-threading, we extend the semantics of execution levels to support multiple threads such that a thread runs at the level at which it is created (threads initially started in the JVM run at level 0).

Levels are implemented using the technique of Polymorphic Bytecode Instrumentation (PBI) [32]. PBI makes it possible to combine different code versions of a method (obtained by applying different transformations) into a single method that starts with a dispatcher. The dispatch logic is responsible for routing execution to the appropriate code version. For execution levels, the dispatcher is the current execution level, and there is one code version per execution level.

Level shifting is introduced by modifying AspectJ-generated code in order to ensure that pointcuts and advices are evaluated one level above the current execution level. The current execution level is a thread-local property implemented as an extra field on the `Thread` class, instead of using the standard `ThreadLocal` class (for efficiency). As in AspectScript, level shifting is done within a `try/finally` block to ensure consistency.

7.3.2. Application of AspectJ with levels

The target application domain of the experiments of AspectJ with levels is that of dynamic analysis aspects. As discussed previously, execution levels are particularly crucial for dynamic analysis, because these aspects usually have very wide pointcuts, intercepting many join points across the complete system. Consider a data race detection aspect called Racer [9], and an aspect for object allocation profiling, called Prof. Racer intercepts all field accesses in order to detect potential races; Prof intercepts all object creations. It turns out that it is impossible to compose both aspects in AspectJ without running into infinite loops; manually modifying them to exclude join points that trigger loops, it is impossible to obtain consistent results.

As a matter of fact, there are multiple possible compositions of these analysis aspects that make sense. Execution levels make it possible to support them, as

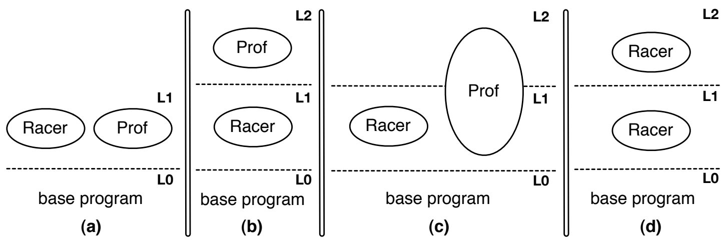

Figure 17: Different deployment and instantiation scenarios to compose two dynamic analysis aspects. (from [45])

illustrated in Figure 17:

- Racer and Prof analyze the base program, without any interference. This is the default, where both aspects are deployed at the same level 1 (a).
- Racer analyzes the base program while Prof profiles object allocation in Racer. This is achieved by deploying Prof at level 2 (b).
- Racer analyzes the base program while Prof profiles object allocation in both the base program and Racer. This is done by deploying the same Prof instance at levels 1 and 2 (c).
- Racer can analyze the base program while another instance of itself (deployed at level 2) analyzes the first one (d).

Being able to take analysis aspects developed by third parties and compose them in such varied ways is definitely a great advantage over the current state of the practice.

7.3.3. Performance

Early benchmarks of MAJOR2, using ad-hoc multiple versions of a method instead of PBI (which was developed later) on the DaCapo suite showed an overhead with execution levels of only 2 to 7% compared to the original AspectJ version [45]. More recent experiments using the PBI implementation of MAJOR2 confirm that the performance is comparable to the manually optimized case in AspectJ (where only lexical checks with `within` are used), incurring slightly less overhead than AspectJ in certain cases. More interestingly, these experiments report that using execution levels is much more efficient than using control flow checks in AspectJ (x5.54 vs. x11.63 on average) [32]. These results are extremely encouraging, especially considering the important gain in semantic stability brought by levels (recall that none of the scenarios of Figure 17 can be correctly obtained with AspectJ).

Finally, the formal results presented in Section 6.10 demonstrate an interesting property of our AspectJ extension. The extension supports neither explicit level shifting nor level-capturing objects, but it does support the possibility to deploy statically the same aspect instance at several levels. This means that programs in this extended AspectJ are by definition level-agnostic. They are, therefore, free of aspect loops. Also, as discussed in Section 5, the AspectJ extension is expressive enough to handle all the aspects from our two studies.

8. Conclusion

The issue of conflation in aspect-oriented programming has been latent since its inception. Neither ad-hoc programming patterns nor primitive mechanisms like `app/prim` and `disable` represent satisfactory solutions. This paper brings to the fore the limitations of these approaches, and proposes a simple mechanism to address conflation properly. By structuring computation in execution levels, it is straightforward to avoid infinite regression in the most common cases. The standard programmer need not even be aware that the runtime system is based on execution levels. Only when fine-grained control is necessary, level-shifting operators make it possible to deploy aspects at higher levels, or move computation up or down, selectively.

On the conceptual side, we believe this work reconciles the (usually unwanted or embarassing) “metaness” of aspects with the (usually unrecognized) “baseness” of runtime metaobject protocols. The key point lies in viewing metaness not as an intrinsic/static property of a piece of program, but as a property of execution flows: the same program element may be used at different levels.

We have developed both the formal foundations and the concrete applications of execution levels. A study of existing AspectJ code, both in a textbook and in a large corpus of AspectJ projects, confirms the relevance of the problem addressed by levels. The formalization, beyond giving confidence in the intuitive properties of levels, helps in understanding both which kind of loops are avoided with execution levels, and which language features imply the possibility of reintroducing these loops. We proved that programs that do not make use of explicit level shifting are free of aspect loops. Our work also shows that levels can be implemented under different flavors, with different efficiency/flexibility tradeoffs. The AspectJ implementation is very competitive with respect to performance, while covering most practical usages.

The main venue for future work lies in exploring whether the strictly layered architecture proposed here is sufficient to deal with aspect visibility requirements found in practice. Initial elements of response can be found elsewhere, where a more flexible of computational membranes is explored [46, 21].

Acknowledgments. We thank: Gregor Kiczales for feedback on this topic and proposal; Paul Leger and Rodolfo Toledo for the discussions related to reentrancy and levels in AspectScript; Walter Binder and his group for the work on implementing levels in AspectJ and applying them to dynamic analyses; Christophe Scholliers for the work on computational contracts; and the reviewers of the Scheme Workshop 2009, AOSD 2010, and this journal, for their constructive and inspiring comments.

References

[1] Danilo Ansaloni, Walter Binder, Alex Villazón, and Philippe Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In AOSD 2010 [2], pages 1–12.

[2] *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, March 2010. ACM Press.

[3] *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, Porto de Galinhas, Brazil, March 2011. ACM Press.

[4] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[5] AspectJ Team. The AspectJ programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/, 2003.

[6] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag, 2006.

[7] Christoph Bockisch, Somayeh Malakuti, Mehmet Akşit, and Shmuel Katz. Making aspects natural: events and composition. In AOSD 2011 [3], pages 285–300.

[8] Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition, 2006.

[9] Eric Bodden and Klaus Havelund. Racer: Effective Race Detection Using AspectJ. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–165, Seattle, WA, USA, July 2008. ACM Press.

[10] Nelio Cacho, Fernando Castor Filho, Alessandro Garcia, and Eduardo Figueiredo. Ejflow: taming exceptional control flows in aspect-oriented programming. In *Proceedings of the 7th international conference on Aspect-oriented software development*, AOSD '08, pages 72–83, New York, NY, USA, 2008. ACM.

[11] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer-Verlag, 1996.

[12] Curtis Clifton and Gary T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:312–374, 2006.

[13] Roberta Coelho, Awais Rashid, Arndt von Staa, James Noble, Uirá Kulesza, and Carlos Lucena. A catalogue of bug patterns for exception handling in aspect-oriented programs. In *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs*, pages 1–13, New York, NY, USA, 2008. ACM.

[14] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3):Article No. 14, May 2008.

[15] Olivier Danvy and Karoline Malmkjaer. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 327–341, Snowbird, Utah, USA, July 1988. ACM Press.

[16] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS Europe*, Lecture Notes in Business and Information Processing, Zurich, Switzerland, July 2008. Springer-Verlag.

[17] Jim des Rivières and Brian C. Smith. The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347, August 1984.

[18] Christopher Dutchyn. *Dynamic Join Points: Model and Interactions.* PhD thesis, University of British Columbia, Canada, November 2006.

[19] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.

[20] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* MIT Press, 2009.

[21] Ismael Figueroa, Nicolas Tabareau, and Éric Tanter. Taming aspects with monads and membranes. In *Proceedings of the 12th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2013)*, pages 1–6, Fukuoka, Japan, March 2013. ACM Press.

[22] Ismael Figueroa and Éric Tanter. A semantics for execution levels with exceptions. In *Proceedings of the 10th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2011)*, pages 7–11, Porto de Galinhas, Brazil, March 2011. ACM Press.

[23] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 48–59, Pittsburgh, PA, USA, 2002. ACM Press.

[24] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In AOSD 2011 [3], pages 227–240.

[25] Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu. Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software. In *International Conference on Automated Software Engineering (ASE 2009)*, pages 575–579, Auckland, New Zealand, November 2009. IEEE/ACM.

[26] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA 92 Workshop on Reflection and Metalevel Architectures*. Akinori Yonezawa and Brian C. Smith, editors, 1992.

[27] Gregor Kiczales. Personal communication, May 2009.

[28] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[30] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Press, 2003.

[31] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[32] Philippe Moret, Walter Binder, and Éric Tanter. Polymorphic bytecode instrumentation. In AOSD 2011 [3], pages 129–140.

[33] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2011)*, pages 59–68, St. Louis, MO, USA, May 2005. ACM Press.

[34] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12)*, pages 147–158. ACM Press, 2004.

[35] Martin P. Robillard and Gail C. Murphy. Designing robust java programs with exceptions. *SIGSOFT Softw. Eng. Notes*, 25:2–10, November 2000.

[36] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Computational contracts. In *Scheme and Functional Programming workshop*, 2011.

[37] Brian C. Smith. Reflection and semantics in a procedural language. Technical Report 272, MIT Laboratory of Computer Science, 1982.

[38] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1):Article 1, June 2010.

[39] Éric Tanter. Controlling aspect reentrancy. *Journal of Universal Computer Science*, 14(21):3498–3516, 2008.

[40] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, April 2008. ACM Press.

[41] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*, pages 3–14, Orlando, FL, USA, October 2009. ACM Press.

[42] Éric Tanter. Higher-order aspects in order. In *Scheme and Functional Programming Workshop*, Boston, MA, USA, August 2009.

[43] Éric Tanter. Execution levels for aspect-oriented programming. In AOSD 2010 [2], pages 37–48.

[44] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Scoping strategies for distributed aspects. *Science of Computer Programming*, 75(12):1235–1261, December 2010.

[45] Éric Tanter, Philippe Moret, Walter Binder, and Danilo Ansaloni. Composition of dynamic analysis aspects. In *Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010)*, pages 113–122, Eindhoven, The Netherlands, October 2010. ACM Press.

[46] Éric Tanter, Nicolas Tabareau, and Rémi Douence. Taming aspects with membranes. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, pages 3–8, Potsdam, Germany, March 2012. ACM Press.

[47] Rodolfo Toledo, Paul Leger, and Éric Tanter. AspectScript: Expressive aspects for the Web. In AOSD 2010 [2], pages 13–24.

[48] Rodolfo Toledo, Angel Núñez, Éric Tanter, and Jacques Noyé. Aspectizing Java access control. *IEEE Transactions on Software Engineering*, 38(1):101–117, Jan./Feb. 2012.

[49] Rodolfo Toledo and Éric Tanter. Access control in JavaScript. *IEEE Software*, 28(5):76–84, Sept./Oct. 2011.

[50] Rodolfo Toledo and Éric Tanter. Secure and modular access control with aspects. In Jörg Kinzle, editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*, pages 157–170, Fukuoka, Japan, March 2013. ACM Press.

[51] Alex Villazón, Walter Binder, Philippe Moret, and Danilo Ansaloni. Comprehensive aspect weaving for Java. *Science of Computer Programming*, 76(11):1015–1036, November 2011.

[52] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: a non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1(1):11–37, 1988.

[53] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

[54] Chris Zimmermann. *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press, 1996.

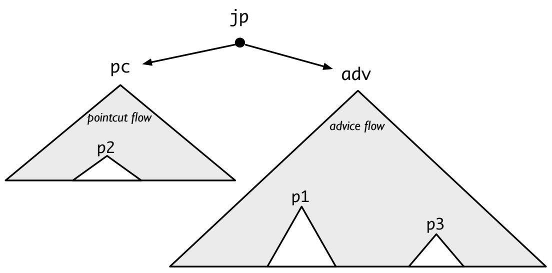

Figure 18: Pointcut and advice flows (in gray).

A. Proof that Level-Agnostic Programs do not (Aspect) Loop

In this section we prove that programs written in the default semantics of execution levels, which we call *level-agnostic* programs because they do not use any level-shifting mechanism explicitly, are free of *aspect loops*. To establish this theorem we first precisely define what we mean by *aspect loops*. Then we prove two lemmas: first, the *level preservation lemma* states that when a term written in user-visible syntax reduces to a value or an exception, the execution level and join point stack have not changed. And second, we prove the *level boundedness lemma*, which specifies that during evaluation, level shifting has a specific lower bound.

A.1. Aspect Loops

To precisely define what we mean by aspect loop, we introduce the notion of *aspect flow*. This definition is based on the correspondence between the trace and the control flow of program evaluation.

Trace Tree The *trace tree* of an expression e is a potentially infinite n-ary tree that represents all function applications performed during reduction or evaluation of e. The applications are registered in postorder in the tree, and the nodes can be labeled as required. Given a node corresponding to a function application, all its subtrees are considered part of the control flow of such application.

Last-Proceed Flow The last-proceed flow is defined as the control flow of any last-proceed application. It corresponds to any trace tree whose root is the application of a last-proceed function. Because last-proceed functions are created by the weaver, it can be assumed that the root nodes are labeled properly.

Pointcut/Advice Flow The pointcut flow (resp. advice flow) of any application of pc (resp. adv) is the *difference* between the trace tree of this application and the trace tree of any last-proceed application. The difference consists in ignoring any subtree that corresponds to a last-proceed flow. Again we assume the weaver properly labels the root nodes in both cases.

Aspect flow The flow of an aspect $\langle l, pc, adv\rangle$ is comprised of the trace tree of its *pointcut flow* and the trace tree of its *advice flow*.

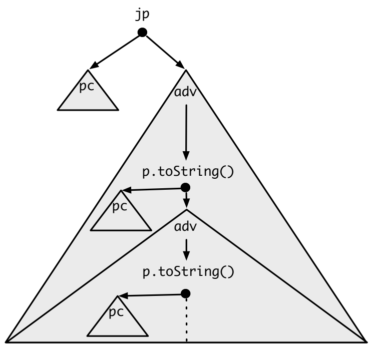

Figure 19: Representation of control flow in an advice loop.

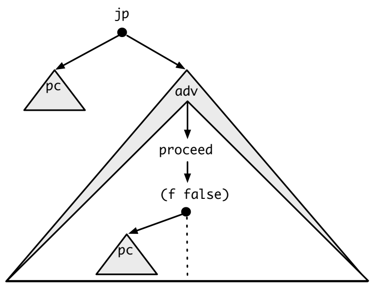

Figure 20: Representation of control flow in a loop that is not an aspect loop.

Note that the above refers to *any* last-proceed application (recall Section 6.7), for both pointcuts and advices. Because of higher-order programming patterns, it is indeed possible to apply at any time a last-proceed that is unrelated to the currently-intercepted join point. Figure 18 intuitively depicts an aspect flow, where several applications of last-proceeds occur, where `p1` $= P[\![$`jp`$]\!]$, and `p2` and `p3` are last-proceeds of join points different from `jp`. With the notion of aspect flow, we can now precisely define what an aspect loop is:

Aspect loop An aspect loop is a loop that occurs due to an aspect seeing a join point from its own aspect flow.

Similarly, we define a *pointcut loop* relative to a pointcut flow, and an *advice loop* relative to an advice flow. Note that the examples of loops in Sections 2.1, 2.2 and 2.3 are indeed aspect loops: they all occur due to the aspect seeing a join point in its aspect flow. To see this consider Figure 19, which depicts the situation of the advice loop of Section 2.1. In contrast, recall the first example of Section 6.10, whose situation is depicted in Figure 20 (in the figures • represents emission of a join point).

A.2. Level preservation

This lemma states that when a term written in user-visible syntax reduces to a value or to an exception, the execution level and join point stack have not changed. This property comes from the fact that the only user-visible expressions that can change the current execution level or join point stack are: `up`, `down`, λ^n and `jp`. As each of these constructs puts a placeholder to remember to go back to the original level or join point stack when a value is computed, and because exception propagation respects these placeholders, the result should not be surprising. If, on the contrary, we allow any expression in e, then the lemma does not hold, as can be seen for instance by considering the term (`in-down` 5).

Lemma A.1 (Level Preservation). *Let e be an expression written in user-visible syntax. For any configuration $\langle l, J, e\rangle$, if*

$$\langle l, J, e\rangle \hookrightarrow^* \langle l', J', v\rangle \qquad \textit{or} \qquad \langle l, J, e\rangle \hookrightarrow^* \langle l', J', (exn\ l''\ v)\rangle$$

where v is a value, then

$$l = l' \text{ and } J = J'.$$

Proof By induction on the length of the reduction, with a case analysis on the first rule. Because we restrict the possible terms appearing in e, the only rules that change the level or the join point stack and can be executed at first in e are InUp, InDown, AppShift and Weave.

- Each rule not mentioned above that can be applied at first does not change the execution level nor the join point stack and produces a term e' in user-visible syntax. So the property holds directly by induction hypothesis.
- Rules InUp or InDown. As both cases are analogous, we only treat Rule InUp. If rule InUp is used, we have that $e = (\texttt{up}\ e')$ and
$$\langle l, J, (\texttt{up}\ e')\rangle \hookrightarrow \langle(\texttt{succ}\ l), J, (\texttt{in-up}\ e')\rangle.$$
By induction hypothesis, if the reduction results in a value
$$\langle(\texttt{succ}\ l), J, e']\rangle \hookrightarrow^* \langle(\texttt{succ}\ l), J, v\rangle.$$
Then, by context preservation and Rule OutUp it follows that
$$\langle(\texttt{succ}\ l), J, (\texttt{in-up}\ e')]\rangle \hookrightarrow^* \langle(\texttt{succ}\ l), J, (\texttt{in-up}\ v)]\rangle \hookrightarrow \langle l, J, v\rangle.$$
If the reduction results in an exception
$$\langle(\texttt{succ}\ l), J, e']\rangle \hookrightarrow^* \langle(\texttt{succ}\ l), J, (\mathit{exn}\ l'\ v)\rangle.$$
Then, by context preservation and Rule OutUpEx it follows that
$$\langle(\texttt{succ}\ l), J, (\texttt{in-up}\ e')]\rangle \hookrightarrow^* \langle(\texttt{succ}\ l), J, (\texttt{in-up}\ (\mathit{exn}\ l'\ v))]\rangle \hookrightarrow \langle l, J, (\mathit{exn}\ l'\ v)\rangle.$$
- Rule AppShift. The initial configuration is of the form
$$\langle l, J, (\texttt{app/prim}\ (\lambda^{l'}(x \cdots)\ e')\ v' \cdots)\rangle$$
which reduces to
$$\langle l', J, (\texttt{in-shift}(l)\ e'\{v' \cdots / x \cdots\})\rangle.$$
By induction hypothesis, if the reduction results in a value
$$\langle l', J, e'\{v' \cdots / x \cdots\}\rangle \hookrightarrow^* \langle l', J, v\rangle.$$
Then, by context preservation and Rule Shift, the expression reduces to $\langle l, J, v\rangle$. On the other hand, if the reduction results in an exception
$$\langle l', J, e'\{v' \cdots / x \cdots\}\rangle \hookrightarrow^* \langle l', J, (\mathit{exn}\ l''\ v)\rangle.$$
By context preservation and Rule ShiftEx, the expression reduces to $\langle l, J, (\mathit{exn}\ l''\ v)\rangle$.
- Rule Weave. We have that $e = (\texttt{jp}\ j)$ for some join point j and
$$\langle l, J, \texttt{jp}\ j\rangle \hookrightarrow \langle l, j + J, (\texttt{in-jp}\ (\texttt{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v \cdots))))\rangle.$$
By induction hypothesis
$$\langle l, j + J, (\texttt{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v \cdots)))\rangle \hookrightarrow^* \langle l, j + J, v\rangle.$$
Then, by context preservation and Rule OutJp it follows that
$$\langle l, j + J, (\texttt{in-jp}\ v)\rangle \hookrightarrow \langle l, J, v\rangle.$$
Exception propagation through `in-jp` is similar to rule OutJp.

A.3. Level boundedness

Before stating the main lemma about level boundedness, we introduce the *shifting lower bound* of a term e, noted $\Omega[\![e]\!]$, as the minimum level with which an instantiated level-capturing function appearing in e is tagged. When a term does not contain any instantiated level-capturing functions, its shifting lower bound is ∞, which means that no level shifts are possible. The idea behind this notion is to keep track of the minimum level that can be reached by applying an instantiated level-capturing function in e.

Formally, $\Omega[\![e]\!]$ is defined by induction on e as

- $\Omega[\![(\lambda^l(x\cdots)e)]\!] = \min(l, \Omega[\![e]\!])$,
- $\Omega[\![(\lambda^\bullet(x\cdots)e)]\!] = \Omega[\![e]\!]$,
- $\Omega[\![(\texttt{jp}\ \lceil l_{jp}, k, v_\lambda, v\cdots\rceil)]\!] = \Omega[\![(v_\lambda v\cdots)]\!]$,
- $\Omega[\![C[e_1,\ldots,e_n]]\!] = \min(\Omega[\![e_1]\!],\ldots,\Omega[\![e_n]\!])$ for any other n-ary constructor C. In particular `try-with` is considered a binary constructor,
- $\Omega[\![C]\!] = \infty$ for any constant term C.

Note that a non-instantiated level-capturing function ($\lambda^\bullet$) does not alter the shifting lower bound of a term as it will be instantiated at the current execution level and so will not be able to shift the execution level down. We extend the notion of shifting lower bound to the global aspect environment pointwisely by

$$\Omega[\![(v_{pc}, v_{adv}) :: \mathscr{A}]\!] = \min(\Omega[\![v_{pc}]\!], \Omega[\![v_{adv}]\!], \Omega[\![\mathscr{A}]\!]).$$

Level-agnostic expression An expression e is *level-agnostic* if it is a user-visible term that does not contain `down`, `up`, or level-capturing functions.

Note that level-agnostic terms have infinite shifting lower bound because by definition they do not contain any instantiated level-capturing function.

We are interested in proving the result on level-agnostic expressions, because they correspond to terms written by a programmer who is unaware of levels. However, during weaving these terms are rewritten in a way that introduces both `up` and instantiated level-capturing functions (due to the last-proceed). We therefore first formulate the lemma for *woven level-agnostic* terms, *i.e.* level-agnostic terms that can contain both `up` and instantiated level-capturing functions. We then specialize the result to level-agnostic terms.

Lemma A.2 (Level Boundedness). *Let e be a woven level-agnostic expression. We assume that for every aspect $\langle l, v_{pc}, v_{adv}\rangle$ contained in $\mathscr{A}$, v_{pc} and v_{adv} are also woven level-agnostic. For any configuration $\langle l, J, e\rangle$, if*

$$\langle l, J, e\rangle \hookrightarrow^* \langle l', J', e'\rangle \text{ then } l' \geq \min(l, \Omega[\![e]\!], \Omega[\![\mathscr{A}]\!]).$$

Proof By induction on the length of the reduction, with a case analysis on the first rule.

- Rules OutUp, InDown, OutDown or Shift are omitted by hypothesis.
- Rules AppPrim or OutJp. Direct by induction hypothesis.
- Rule App. Direct by induction hypothesis, using the fact that

$$\Omega[\![((\lambda(x\cdots)\ e)\ v\cdots)]\!] = \Omega[\![(\texttt{jp}\ \lceil(\texttt{succ}\ l), \texttt{call}, (\lambda(x\cdots)\ e), v\cdots\rceil)]\!].$$

- Rule Capture. Direct by induction hypothesis, using the fact that

$$\min(l, \Omega[\![(\lambda^{\bullet}(x \cdots)\ e)]\!]) = \min(l, \Omega[\![(\lambda^{l}(x \cdots)\ e)]\!].$$

- Rule InUp: we have that $e = (\texttt{up}\ e_0)$ and

$$\langle l, J, (\texttt{up}\ e_0)\rangle \hookrightarrow \langle(\texttt{succ}\ l), J, (\texttt{in-up}\ e_0)\rangle.$$

  Two cases must be considered: (i) Either the final reduction stays inside `in-up`, that is $e' = (\texttt{in-up}\ e'')$ and

$$\langle(\texttt{succ}\ l), J, (\texttt{in-up}\ e_0)\rangle \hookrightarrow^{*} \langle l', J', (\texttt{in-up}\ e'')\rangle.$$

  By induction hypothesis (and context preservation), $l' \geq \min((\texttt{succ}\ l), \Omega[\![e']\!], \Omega[\![\mathscr{A}]\!])$, which is more than we need.
  (ii) Or the final reduction concerns `in-up`, that is

$$\langle(\texttt{succ}\ l), J, (\texttt{in-up}\ e_0)\rangle \hookrightarrow^{*} \langle l', J', v\rangle.$$

  In that case, $l = l'$ by Lemma A.1.

- Rule Deploy: direct by induction hypothesis, using the fact a woven level-agnostic term can only deploy woven level-agnostic aspects.

- Rule Weave: it must hold that $e = (\texttt{jp}\ j)$ for some join point j and, by letting e_W be $(\texttt{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v \cdots))$, we have

$$\langle l, J, \texttt{jp}\ j\rangle \hookrightarrow \langle l, j + J, (\texttt{in-jp}\ e_W)\rangle.$$

  Two cases must be considered: (i) Either the final reduction stays inside `in-jp`, that is $e' = (\texttt{in-jp}\ e'')$ and by induction hypothesis, using the fact that each advice is woven level-agnostic and therefore e_W is a woven level-agnostic expression, we have

$$\langle l, j + J, (\texttt{in-jp}\ e_W)\rangle \hookrightarrow^{*} \langle l', j + J, \texttt{in-jp}\ e'\rangle$$

  with

$$l' \geq \min(l, \Omega[\![(\texttt{in-jp}\ e_W)]\!]).$$

  We conclude using the fact

$$\Omega[\![(\texttt{in-jp}\ e_W)]\!] = \min(l, \Omega[\![\texttt{jp}\ j]\!], \Omega[\![\mathscr{A}]\!])$$

  because the only new instantiated level-capturing function $P[\![j]\!]$ in e_W is at level l. (ii) Or the final reduction concerns `in-jp`, that is

$$\langle(\texttt{succ}\ l), J, (\texttt{in-jp}\ e_W)\rangle \hookrightarrow^{*} \langle l', J', v\rangle.$$

  In that case, $l = l'$ by Lemma A.1.

- Rule AppShift: The initial configuration is of the form

$$\langle l, J, (\texttt{app/prim}\ (\lambda^{l_0}(x \cdots)\ e_0)\ v \cdots)\rangle$$

  which reduces to

$$\langle l_0, J, (\texttt{in-shift}(l)\ e_0\{v \cdots / x \cdots\})\rangle.$$

Two cases must be considered: (i) Either the final reduction stays inside $\texttt{in-shift}(l)$, that is $e' = \texttt{in-shift}(l)\ e''$ and

$$\langle l_0, J, (\texttt{in-shift}(l)\ e_0\{v\cdots/x\cdots\})\rangle \hookrightarrow^* \langle l', J', (\texttt{in-shift}(l)\ e'')\rangle.$$

Then by induction hypothesis, $l' \geq \min(l_0, \Omega[\![e_0\{v\cdots/x\cdots\}]\!], \Omega[\![\mathscr{A}]\!])$. We conclude using

$$\Omega[\![(\texttt{app/prim}\ (\lambda^{l_0}(x\cdots)\ e_0)\ v\cdots)]\!] = \min(l_0, \Omega[\![e_0\{v\cdots/x\cdots\}]\!]).$$

(ii) Or the final reduction concerns $\texttt{in-shift}(l)$ so e' is a value and $l = l'$ by Lemma A.1.

Observe that to reach an execution level smaller that the starting execution level (that is $l' < l$) it must be the case that e' is in the control flow of a level-capturing function application. This can be easily checked by looking at the proof above and noticing that no applicable rule but AppShift makes the current level go down.

Theorem 6.1 (No Aspect Loops). *Let e be a level-agnostic term. If all aspects in $\mathscr{A}$ are also level-agnostic, the reduction of e is free of aspect loops.*

Proof An aspect deployed at level ($\texttt{succ}\ l$) can only see join points generated at level l. Thus, using Lemma A.2, this means that an aspect can see its own join points only inside a level-shifting function application. But the only level-shifting functions that appear during the execution of e are last-proceed functions, whose applications are by definition not part of the aspect flow.

A.4. Extension to references

Lemma A.2 can be transposed to a language with references in a straightforward way. We first extend pointwisely the notion of shifting lower bound to the store s by posing

$$\Omega[\![s]\!] = \min_{\alpha \in \mathcal{L}} \Omega[\![s(\alpha)]\!]$$

where $\mathcal{L}$ is the set of locations of the store. We then weaken the statement of the lemma by asserting that

$$l' \geq \min(l, \Omega[\![e]\!], \Omega[\![\mathscr{A}]\!], \Omega[\![s]\!]),$$

which means that the execution level can be shifted down also by instantiated level-capturing functions present in the store. The proof is the same, the rules for reference assignment and dereferencing behave in the same way than rules Deploy and Weave. Theorem 6.1 requires that every function in the store is level-agnostic.