Execution Levels for Aspect-Oriented Programming

Éric Tanter *

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Santiago, Chile
etanter@dcc.uchile.cl

Abstract

In aspect-oriented programming 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 in all cases, and give advanced programmers the means to override this default using level shifting operators. We implement our proposal as an extension of AspectScheme, and formalize its semantics. This work recognizes that different aspects differ in their intended nature, and shows that structuring execution contexts helps tame the power of aspects and metaprogramming.

1. Introduction

In the pointcut-advice model of aspect-oriented programming, as embodied in *e.g.* AspectJ [16] and AspectScheme [11], 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 originally pointcuts were conceived as purely "meta" predicates that cannot have any interaction with base level code [28], 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 consider a pointcut as a first-class, higher-order function like any other, thus giving the full computational power of the base language to express both pointcuts and advices.

While pointcuts were initially conceived of as pure metalevel predicates, advices were seen as a piece of base-level functionality [28]. In other words, an advice is just like an ordinary function or method, that happens to be triggered "implicitly" whenever the associated pointcut predicate matches. Considering advice as base-level code clearly distinguishes AOP from runtime metaobject protocols (to many, the ancestors of AOP). Indeed, a metaobject runs, by definition, at the metalevel [18]. This makes it possible to consider metaobject activity as fundamentally different from base level computation, and this can be used to get rid of infinite regression [8]. In AOP, infinite regression can also happen, and does happen, easily[1]: it is sufficient for a piece of advice to trigger a join point that is potentially matched by itself (either directly or indirectly). This is one of the reasons why a specific kind of join point, which denotes advice execution, has been introduced in AspectJ [28]. This join point allows one to rule out join points that are caused by executing an advice.

In recent work, we analyze this issue further and show that AspectJ fails to properly recognize the possibility of infinite regression due to pointcut evaluation [22]. We proposed a solution that consists in introducing a pointcut execution join point, and a defensive default that avoids aspects matching against their own execution. Other languages like AspectScheme and AspectML [6] introduce special primitives to control infinite regression. For instance, AspectML suggests a **disable** primitive to evaluate an expression without generating any join point. However, all these solutions rely on control flow checks, which are eventually unable to properly discriminate aspect computation from base computation.

Since all these issues are reminiscent of conflation of levels in reflective architectures [3], we choose to question the basic assumption that pointcut and advice are *intrinsically* either base or meta. For instance, looking at how programmers use advices, it turns out that some advices are clearly base code, while some are not: *e.g.* generic aspects, advices that use `thisJoinPoint` (reification of the current join point to be used in the advice), etc. To get rid of this tension between AOP and MOPs, or between "all is base" and "all is meta", we propose a reconciliating approach in which the metaness concern is decoupled from the pointcut-advice mechanism. This is done by introducing in the core execution model a notion of *level of execution*. Aspects are bound to observe execution of particular levels. To alleviate the task for non-expert pro-

* Partially funded by FONDECYT projects 11060493 & 1090083.

AOSD 2010 Submission

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

grammers, we adopt a defensive default that avoids regression in all cases by making aspect computation happen at a higher level. For the advanced programmer, level shifting operators provide complete control over what aspects see and where they run (*i.e.* who sees them). Execution levels seamlessly address all the issues of current proposals, while maintaining extreme simplicity in the most common cases.

This paper is structured as follows: Section 2 describes several issues with the current state of affairs regarding aspect weaving. Section 3 briefly discusses current attempts at addressing these issues. Section 4 connects these issues to the fundamental issue of conflation. Section 5 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 seamlessly addressed. We formalize the operational semantics of our proposal in Section 6, by modeling a higher-order aspect language with execution levels. Section 7 discusses related work and Section 8 concludes.

2. A Plethora of Issues

This section briefly visits 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 rely on control flow checks, has almost acquired the status of a pattern. The second issue we discuss, pointcut loops, is only doubtfully and partially addressed, while the last three issues 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.

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);
 }
}
```

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 knows 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, but which 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(Active))` 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(Active)` is too "loose", since it only rejects join points that occur *lexically* in the advice; this would clearly be insufficient in our example, because the *execution* of `toString` is not lexically in the advice (only the *call* is).

2.2 Pointcut Loops

Let us refine the `Activity` aspect such that only point objects 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)) ... {
   ...
 }
}
```

We use `this(p)` to get a hold on the currently-execution point object and 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. In this case however, we cannot use a precise cflow check because there is no way to refer to a pointcut execution (in AspectJ or in any aspect language we know of).

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))`. While something equivalent would work in AspectScheme, this is totally impossible with current AspectJ compilers (in the absence of a complete formal semantics of the language, compilers dictate). Surprisingly, both `ajc` and `abc` *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 to the advice(s).

2.3 Confusion all Around

To add to the already-large confusion and complexity, control flow checks, when possible, interfere in unpleasing ways with the *kind* of advice bound to a pointcut.

Up to now, we have only used *before* advice in the examples. Aspect languages generally support *around* advice as well, with the following equivalence[2]:

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

is equivalent to:

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

Consider the following tracing aspect:

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

With before advice, `Activity` prints 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

[2] The degree to which this equivalence is explicitly recognized and accepted differs according to the language. For instance, in AspectScheme, before advice is really but 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.

the point object, it is subject to an advice loop that can only be avoided using a control flow check (Section 2.1). As it turns out, it is *impossible* to obtain that same behavior (tracing all executions) with an around advice! The reason is that the advice now has to call `proceed` in order to trigger the original base computation. This means that the advice execution control flow check, whose purpose is to avoid the advice loop, also discards *all subsequent join points of the nested base program execution*. 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`. 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.

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 `Activity` call `refresh`, and `Coalescing` ought to be aware of all of them. For that, 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

Finally, control-flow checks completely brake in the 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. 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(p.toString());
  }
}

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

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 clearly not the general case, and in addition, as argued in Section 2.1, the pattern is too strict.

3. Preliminary Solutions

There have been several attempts to address the above issues, either by selectively and explicitly disabling aspect weaving, or by implicitly detecting looping situations and avoiding them. Other related proposals are discussed in Section 7.

Explicit disabling of weaving. AspectScheme supports a *primitive function application*, **app/prim**, which does not generate a join point. This is required in AspectScheme to address obvious looping issues: because pointcuts and advice are standard first-class functions, even applying the `proceed` function itself can generate an infinite loop. Therefore, **app/prim** is typically used to apply the `proceed` function as well as to apply functions in pointcuts. Unfortunately, **app/prim** does not help in any of the issues presented here 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 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.

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* [22]. We distinguish 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.

While they seem to address the looping issues, all these approaches are (at best) based on control flow checks and therefore fail when considering the three last issues presented in Section 2. None of them considers the issue of confusing base and advice execution through `proceed` (the 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. In summary, we believe that relying on control-flow checks is inherently flawed. This analysis points towards a fundamental issue, yet to be identified. Ideally, a proper solution to the fundamental issue would make it possible to straightforwardly address all of them in an elegant and robust manner.

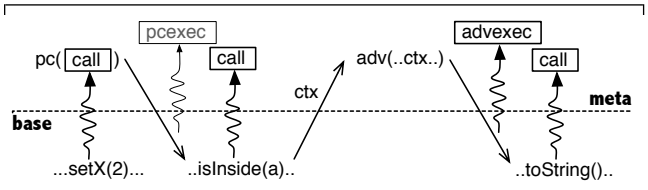

Figure 1. Join points and aspect execution in aspect languages with `if` pointcuts or higher-order pointcuts, and base-level advice.

4. Stepping Back: Conflation

Let us look back a little:

> *"very often, the concepts that are most natural to use at the meta-level cross-cut those provided at the base level."* [13]

This visionary sentence from a seminal paper that reveals the seed of what is now known as aspect-oriented programming is intriguing. It 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 [28]), advice is just another—probably misnamed—piece of code that has the same ontological status as a method [14].

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 our new language, AspectScript [17]. 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 application logic, we also acknowledge the fact that AOP *can* be (and is) used for metaprogramming. Many applications that used to be considered as illustrative of MOPs [29], like synchronization and monitoring, are now programmed using aspects, mostly due to the practical benefits of pointcut languages.

Why does it matter? History. As a matter of fact, the issues we have been exposing up to now are reminiscent of the issue of *meta-circularity*, which has long been identified in reflective architectures [9]. 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). The widely-used and ad hoc solution to this problem is to add base checks that stop regression, such as explicit control-flow checks in AspectJ, or the default reentrancy control we proposed previously [22]. Another solution is to introduce a more primitive mechanism that is *not* subject to redefinition, like AspectScheme's **app/prim** and AspectML's **disable**.

This said, the meta-object protocol literature has recognized that these approaches eventually fall short. In particular, Chiba, Kiczales and Lamping show that they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [3]. That these ad-hoc approaches fail is precisely the point we have made in this paper so far. 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 ($n + 1$) than "base" code (n). On the one hand, this allows for a stable semantics, where issues of conflation can be avoided [3, 8]. On the other hand, this boils down to reconsidering AOP as just a form of metaprogramming. Only Bodden *et al.* have looked at this issue in AOP and proposed a solution based on placing aspects at different levels of execution, recognizing advice execution as a meta activity [2]. However, seeing advice as *inherently* meta defeats the original idea of AOP [14].

We propose to resolve this conflict by decoupling the "metaness" concern from the pointcut and advice mechanism. We introduce *execution levels* in the language, in order to structure computation. We opt for a *default* semantics regarding pointcuts and advices that favors stability. That is, by default, we consider both pointcut and advice execution as higher-level computation, invisible to aspects. This arguable choice is purely motivated by a defensive concern: the unaware programmer should not face potential interferences unless she consciously chooses to. Beyond our own experience and that of others [2], a brief study of the AspectJ examples included with the standard distribution shows that this default does make sense. As we will see, when necessary in order to observe aspectual computation, aspects can be explicitly deployed at higher levels. In addition, we provide level shifting operators in the language, so that advanced programmers can specify their intent with respect to the ontological status of their pointcuts and advices (and by extension, of any expression).

5. Execution Levels

In this section, we introduce execution levels and discuss how they can be used in conjunction with aspects. Section 5.1 exposes the default way in which pointcuts and advices are evaluated, highlighting one of the major difference between relying on execution levels vs. relying on control flow checks. Section 5.2 gives more control to programmers by introducing level shifting expressions. Section 5.3 defines a notion of control flow that is sensitive to execution levels. It also highlights the second fundamental difference between execution levels and control flow, in that it is possible to *capture* and later *reinstate* an execution level. Section 5.4 explains how it is possible, using execution level shifting and higher-order programming, to overwrite our defensive default to revert to the view of pointcuts as meta, advice as base. Section 5.5 summarizes the benefits of execution levels and how they make it possible to address all the issues we have raised so far. Section 5.6 briefly discusses an interesting perspective raised by the introduction of execution levels.

5.1 Aspects and Levels: Default

Figure 2 depicts the default evaluation of pointcuts and advice with level shifting. As before, we adopt the convention that the evaluation of base code (at level 0) generates join points at level 1 (*e.g.* the call box), where aspects can potentially match and trigger advice. Pointcut and advice execution join points are generated, but at level 2. Similarly the whole evaluation of pointcuts and advices is done at level 1, so the join points produced in the dynamic extent of these evaluations are generated at level 2. This ensures that the call of `isInside` done during pointcut evaluation of `Activity` is not seen at the same level as the call to `setX` (level 0). The same holds for the call to `toString` in the advice. The default semantics therefore addresses both issues raised in Sections 2.1 and 2.2.

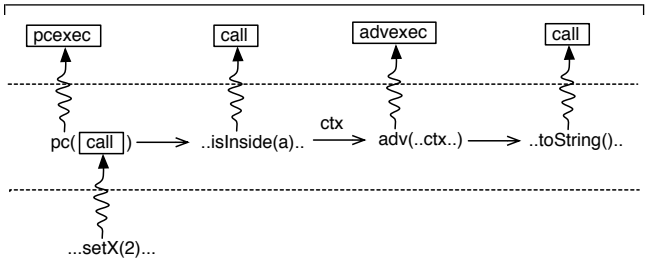

Figure 2. Running pointcut and advice at a higher level of execution.

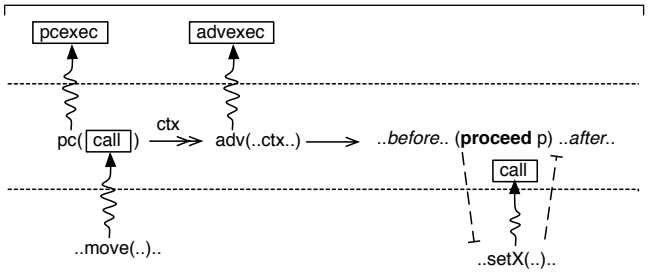

Figure 3. Proceeding to the original computation is done at the lower level.

Proceed. As briefly explained in Section 3, an advice can *proceed* to the computation originally described by the join point. Logically, 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 or not, and no matter the advice kind. There is no reason why using around advice (with `proceed`) rather than before advice should change the status of the underlying computation.

In order to address this crucial issue, it is also important to remember that when several aspects match the same join point, the corresponding advices are chained such that calling `proceed` in advice k triggers advice $k + 1$. The original computation is performed only when the *last* advice proceeds.

Therefore, our default semantics ensures that the *last call* to `proceed` in a chain of advices triggers the original computation at the lower 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 3, 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 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. For instance, with *stratified aspects* [2], this is done by declaring certain aspects as $meta[n]$ where n is the level at which the aspect stands. The following section introduces a more uniform and flexible solution to this issue.

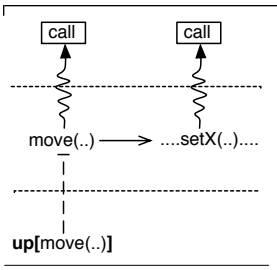

Figure 4. Shifting up.

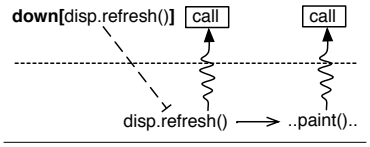

Figure 5. Shifting down.

5.2 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 [28, 14]. In some cases, advice execution should be visible to aspects that observe base level execution. This approach is more compatible with the traditional AO view that "advices are base". From an engineering viewpoint, it allows the implementor of an aspect to declare that some part of its (pointcut and/or) advice should be considered as standard base code. Other aspects then do not need to be explicitly deployed at a higher level; 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/advice mechanism, and can be used to move any computation.

Figure 4 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 5.

Using **up** and **down**, it is possible to control where aspectual computation is performed, relative to the default semantics described in Section 5.1. One can also use these level shifting operators to actually *deploy* aspects at a particular level.

Deploying aspects of aspects. In Section 5.1 we mentioned the fact that the default semantics requires aspects of aspects to be deployed at a higher level. To illustrate this, as well as to start connecting with our upcoming formalization and implementation (Section 6), we briefly introduce a simple aspect-oriented extension to the higher-order procedural programming language Scheme, which considers only one kind of join points, function application.

An aspect is defined by two functions, a pointcut function and an advice function. An aspect is deployed globally using the `deploy` primitive. For instance:

```
(deploy pc adv)
```

Assuming the above expression is evaluated at level 0, its effect is to deploy an aspect defined by pointcut `pc` and advice `adv` at level 1. This aspect then observes base level computation. 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 eval-

uated. 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.

Shifting some aspect computation. One can use level shifting operators directly within the definitions of pointcut and advice. Briefly, 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. An advice takes as parameters a `proceed` function, a list of context values (coming from the pointcut), and the arguments at the join point[3].

The following code defines a `point-in-area` pointcut, which checks whether the first argument at the join point is a point structure (using `Point?`), and if that point is within a given area (using `is-inside`). The `activity` advice writes out the point object, refreshes the display, and `proceed`s. Finally, the aspect is deployed (with global scope).

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

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

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

In the definition of the advice, we use the level shifting operator **down** to move the computation of `display-refresh` down to the base level. This allows another aspect, like `Coalescing`, to take effect and optimize that computation. Note that evaluating the pointcut does not cause infinite loops, because the application of both the `Point?` predicate and the `is-inside` function remain at the meta level and are therefore not observable by aspects at the same level[4]. 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. This is because the join points corresponding to the lowered computation are seen on the same level as where the aspect resides. To avoid these self-caused loops (caused by explicitly moving down a computation), *reentrancy control* is needed. Such control would be simpler than our previous work [22] because having execution levels only leaves open that specific case. An in-depth and formal treatment of reentrancy control in conjunction with execution levels is however outside the scope of this paper. (AspectScript [17] integrates both.)

[3] This modeling follows—save some details about currying of advice—the model of AspectScheme [11].

[4] 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 could lead to loops in AspectScheme, if **app/prim** were not used.

5.3 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 place. We now extend the traditional notion of control flow to take levels into account. Finally, we show how the ability to capture execution levels in certain functions makes it possible to address the concurrency issue of traditional control flow checks (Section 2.5).

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[5]. 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 [11, 26]. The only modification needed to make these PCDs 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.

Capturing execution levels. We now turn our attention to the last, still unresolved issue, that of concurrency (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*. Intuitively, this corresponds to dynamic scoping, and fits with the notion that the execution level is a property of a flow of execution. The counterpart of this dynamic scoping strategy for execution levels is static scoping: executing a function at the level at which *it was defined*. As it turns out, this is precisely the feature we need to track delayed advice execution[6].

[5] We do not consider state-based (as opposed to stack-based) implementation of control flow checks here [19]. It is straightforward to extend our argument to state-based `cflow`.

[6] The idea of level-capturing functions is directly inspired by the reflective language Blond [7], which supports two different kinds of reflective procedures (more on this in Section 7).

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

Table 1. Benefits of execution levels to address the issues of Section 2.

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 at which it was *defined*.

```
(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, ensures 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.

5.4 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 AO view holds: pointcuts at the meta level, and advice at the base level.

We can certainly take advantage of the fact that we are dealing with 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:

```
(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 express the original AO view according to which pointcuts are metalevel predicates and advice is base code. We can define a syntactic sugar `deploy-aj` as follows:

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

To conclude, this exercise illustrates once again the fundamental difference between execution levels and traditional/conflating control flow checks. The possibility to shift up/down the `proceed`ing computation is fundamental in order to avoid the confusion raised by conflation.

5.5 Summary: Benefits of Execution Levels

Table 1 summarizes the benefits of execution levels compared to current aspect-oriented programming practice. The columns refer to the different issues described in Section 2, in order. The first row describes the situation of "plain" AOP, that is, without using any particular defense against loops; unsurprisingly, infinite loops are not avoided. Using control flow checks, one avoids advice loops. However, current AspectJ compilers do not make it possible to avoid pointcut loops. (These are avoided in [22], and can be avoided using **disable** [6].) Also, it is not possible to properly discriminate aspect computation from base computation, due to `proceed` conflation. In both cases, computation of aspects is always visible to other aspects, *i.e.* it is not possible to hide some aspectual computation. Also, using `cflow`, it is not possible to recognize aspect computation that has been delayed or delegated to some other thread.

Introducing execution levels in an aspect-oriented language seamlessly addresses all the issues described. Programmers do not have to use defensive programming patterns to avoid loops: loops are avoided by default, simply by having pointcut and advice computation occurring at a higher level. Because the last call to `proceed` in a chain of aspects shifts down back to the original level, there is no confusion between aspect and base computation. In addition, using control flow in aspect definitions does not introduce any risk of conflation, because a level-sensitive control flow pointcut is used. By default, aspectual computation is hidden from other aspects, but it can be made visible either by deploying aspects at a higher level, or by lowering just the relevant part of the aspect computation. (In that case, reentrancy control is needed to avoid self-caused loops.) Finally, level-capturing functions make it possible to discriminate aspect computation even when it is executed by another thread.

5.6 Perspective: Level Shifting and Information Hiding

As we have explained, moving (parts of) pointcuts and advices up and down allows one to control their visibility with respect to other aspects. As it turns out, level shifting is orthogonal to the pointcut/advice mechanism, to the extent that it applies to any expression, not only expressions within pointcuts and advice bodies. This mechanism can therefore be used to run any arbitrary piece of code at another level of execution.

For instance, if a function invokes a security manager each time it is applied in order to ensure that its execution is authorized, it

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

Figure 6. Syntax of the core language.

can "hide" the invocation and execution of the security manager from aspects observing its execution level by pushing it to a higher level. This means that level shifting can be used to address, to some extent, the issue of information hiding violation that has been raised with respect to standard aspect languages. For instance, in Open Modules [1], only join points explicitly exposed through pointcuts declared in the interface of a module are visible to aspects of other modules.

Finally, the level-shifting operators **up** and **down** are relative only, making it possible to shift execution one level up or down, respectively. One could consider a **bottom** operator that moves execution down to level 0, as well as a **top** operator that moves execution to the uppest level, so that execution is invisible to all aspects. It remains to be determined through practical experience whether the operators we propose are sufficient[7]. The semantics we present in the following section only considers **up** and **down**, though it would be straightforward to accomodate others.

6. Semantics

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 6.6).

Figure 6 presents the user-visible syntax of the core language, *i.e.* without aspects nor execution levels. The language is a simple 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.

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.

[7] Our yet-unextensive experience with both our Scheme implementation (Section 6.6) and AspectScript [17], is that in most cases, explicit level shifting is not necessary and therefore just 2 levels suffice. When aspects of aspects are required, for instance for access control, we make use of a third level. We have not found the need for `bottom`, and our recent work on access control through aspects strongly discourages the presence of `top`.

$$
\begin{array}{rlcll}
Expr & e & ::= & \ldots \mid (\textbf{up}\ e) \mid (\textbf{down}\ e) \mid & \\
 & & & (\texttt{in-up}\ e) \mid (\texttt{in-down}\ e) & \\
EvalCtx & E & ::= & \ldots \mid (\texttt{in-up}\ E) \mid (\texttt{in-down}\ E) & \\
 & \langle l, J, E[(\textbf{up}\ e)] \rangle & \hookrightarrow & \langle l+1, J, E[(\texttt{in-up}\ e)] \rangle & \textsc{InUp} \\
 & \langle l, J, E[(\texttt{in-up}\ v)] \rangle & \hookrightarrow & \langle l-1, J, E[v] \rangle & \textsc{OutUp} \\
 & \langle l, J, E[(\textbf{down}\ e)] \rangle & \hookrightarrow & \langle l-1, J, E[(\texttt{in-down}\ e)] \rangle & \textsc{InDwn} \\
 & \langle l, J, E[(\texttt{in-down}\ v)] \rangle & \hookrightarrow & \langle l+1, J, E[v] \rangle & \textsc{OutDwn}
\end{array}
$$

Figure 7. Shifting execution levels.

In the following we describe the semantics of execution levels, join points, aspects and their deployment, as well as the weaving semantics. By convention, when we introduce new user-visible syntax (*e.g.* the aspect deployment expression), we use **bold** font. Extra expression forms added only for the sake of the semantics are written in `typewriter` font.

6.1 Execution Levels

The language supports explicit execution level shifting forms, **up** and **down** (Figure 7). Correspondingly, there are two (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).

6.2 Join Points

We follow Clifton and Leavens [4] in the modeling of the join point stack (Figure 8). 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 (not user-visible) 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 8).

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 8, 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 ($l+1$), 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 stack J, marking the expression with `in-jp`, and then triggers weaving. Poping 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.

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

$$
\begin{array}{l}
\langle l, J, E[((\lambda(x \cdots)\ e)\ v \cdots)] \rangle \qquad \textsc{App} \\
\hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l+1, \texttt{call}, (\lambda(x \cdots)\ e), v \cdots \rceil] \rangle \\
\\
\langle l, j + J, E[\texttt{in-jp}\ v] \rangle \quad \hookrightarrow \quad \langle l, J, E[v] \rangle \qquad \textsc{OutJp}
\end{array}
$$

Figure 8. Join points: stack, creation and disposal.

$$
\begin{array}{rrcl}
\mathit{Aspects} & \mathscr{A} & = & \{\langle l_i, pc_i, adv_i \rangle \mid i = 1, \ldots, |\mathscr{A}|\} \\
\mathit{Pointcut} & pc & \in & \mathscr{J} \to \{\#f\} \cup \mathscr{L} \\
\mathit{Advice} & adv & \in & (\mathscr{V}^* \to \mathscr{V}) \times \mathscr{L} \times \mathscr{V}^* \to \mathscr{V} \\
 & prim & ::= & \ldots \mid \textbf{deploy}
\end{array}
$$

$$
\begin{array}{l}
\langle l, J, E[(\textbf{deploy}\ v_{pc}\ v_{adv})] \rangle \qquad \textsc{Deploy} \\
\hookrightarrow \langle l, J, E[\textit{unspecified}] \rangle \text{ and } \mathscr{A} = \{\langle l+1, v_{pc}, v_{adv} \rangle\} \cup \mathscr{A}
\end{array}
$$

Figure 9. Aspects and deployment (global environment $\mathscr{A}$).

6.3 Aspects and Deployment

As described on Figure 9, 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. Following [10, 11], 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 (we omit the currying of advice).

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 9). 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[8] (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 environments, see [23]. 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 $l + 1$ (which in turn denote computation of level l); (**up** (**deploy** $p\ a$)) deploys the aspect a level above, such that it sees join points of level $l + 2$, *i.e.* which denote execution at level $l + 1$.

[8] The complete semantics we provide properly includes the aspect environment in the evaluation steps (Section 6.6).

$$
\begin{array}{l}
\langle l, J', E[\texttt{jp}\ \lceil l, k, v_\lambda, v \cdots \rceil] \rangle \qquad \textsc{Weave} \\
\quad \hookrightarrow \langle l, J, E[(\texttt{in-jp}\ (\textbf{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v \cdots)))] \rangle \\
\qquad\qquad \text{where } J = j + J' \\
\\
\text{and, with } J = \lceil l, k, (\lambda(x \cdots)\ e), v \cdots \rceil + J' \text{:} \\
W[\![0]\!]_J \quad = \quad (\lambda(a \cdots) \\
\qquad (\textbf{down}\ (\texttt{app/prim}\ (\lambda(x \cdots)\ e)\ a \cdots))) \\
W[\![i]\!]_J \quad = \quad (\texttt{app/prim}\ (\lambda(p) \\
\qquad (\textbf{if}\ (\textbf{eq?}\ l_i\ l) \\
\qquad\quad (\textbf{let}\ ((c\ (pc_i\ J))) \\
\qquad\qquad (\textbf{if}\ c \\
\qquad\qquad\quad (\lambda(a \cdots)(adv_i\ p\ c\ a \cdots)) \\
\qquad\qquad\quad p)) \\
\qquad\quad p)) \\
\qquad\quad W[\![i-1]\!]_J)
\end{array}
$$

Figure 10. Aspect weaving, with level shifting.

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

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

Figure 11. Primitive application.

6.4 Weaving

We now turn to the semantics of aspect weaving. The WEAVE rule describes the process (Figure 10). 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 only differs in that we deal with execution levels, and introduce both pointcut and advice join points. 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 dispatched.

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 parameterized by the function to proceed with, p. This function is passed to the advice, and if an aspect does not apply, then W simply returns this function. The base case, $W[\![0]\!]_J$ corresponds to the execution of the original function. Note that it is performed by **down**ing the execution level, to reflect the fact that while pointcuts and advice run at an upper level, the original function runs at its original level of application.

The WEAVE rule uses a primitive application form, `app/prim`, described in Figure 11. This form denotes an application that does

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

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

$$
\langle l_1, J, E[(\texttt{app/prim}\ (\lambda^{l_2}(x \cdots)\ e)\ v \cdots)]\rangle \hookrightarrow \langle l_2, J, E[(\texttt{in-shift}(l_1)\ e\{v \cdots / x \cdots\})]\rangle \quad \textsc{AppShift}
$$

$$
\langle l_2, J, E[(\texttt{in-shift}(l_1)\ v)\rangle \hookrightarrow \langle l_1, J, E[v]\rangle \quad \textsc{Shift}
$$

Figure 12. Level-capturing functions.

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. Finally, `app/prim` is necessary to eventually perform the original function application, when all aspects (if any) have `proceed`ed (see $W[\![0]\!]_J$). Note that contrary to AspectScheme, `app/prim` is *not* in user-visible syntax, thanks to execution levels. Also note that in W, the pointcut and advice functions are applied using a standard function application.

6.5 Level-capturing functions

Figure 12 extends the semantics of the language with level-capturing funtions. There is a new syntactic form to define a level-capturing function, $\lambda^{\bullet}$, and a new value form, λ^{l}, which represents a function that is always executed at level l. The capturing of the level is performed by the rule Capture. In order to keep track of the level shifting incurred by applying a level-capturing function, there is an extra expression `in-shift` that captures 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).

6.6 Availability

We have defined the complete semantics of our language using PLT Redex, a domain-specific language for specifying executable reduction semantics [12]. The full definition along with executable test cases, as well as the automatically-generated rendering of the language grammar, reduction relation and weaving metafunction W, are available at: `http://pleiad.cl/research/scope`

We have also implemented our language as an extension of AspectScheme (*i.e.* a language module extending PLT Scheme using macros), available at the same website. The language supports both call and execution join points. Level shifting forms are implemented simply as macros that handle a dynamically-scoped parameter. The language includes different scoping semantics for aspects (statically and dynamically scoped) in addition to global, top-level deployment.

In addition to these definitional artefacts, our group is actively developing the AspectScript language for expressive aspect-oriented programming in JavaScript. AspectScript [17] takes full advantage of the higher-order functional programming features of JavaScript, and relies at its core on the work presented here on execution levels, as well as reentrancy control [22] and expressive scoping of aspects using scoping strategies [23, 24].

7. Related Work

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 [20] with 2-Lisp and 3-Lisp, and different flavors of it were subsequently explored, with languages like Brown [27] and Blond [7].

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, 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*—a direct inspiration for the level-capturing functions we introduced here.)

In this framework, one could describe the pointcut-advice mechanism as follows, at least in its original form [28]. Pointcuts are reflective procedures, that take as parameter (a representation of) the current join point. In contrast to reflective procedures in reflective languages, they are not explicitly applied; rather, they are "installed" in the interpreter, and their application is triggered by the interpreter at each join point. A pointcut runs at the upper level and, if it matches, returns bindings that are consequently used for the (base-level) execution of the advice.

The level shifting operators we introduce in this work differ from level shifting in the reflective tower in a number of ways. Most importantly, there is no tower of interpreters at all: execution levels are just properties of execution flows. Only aspects (more precisely, pointcuts) are sensitive to this property of execution flows. Pointcuts and advices are all evaluated by the very same interpreter that evaluates the whole program. Level shifting operators just taint the execution flow such that the produced join points are only visible to aspects sitting at the corresponding level. This "illusion of the tower" also explains why there is no explicit wrapping and unwrapping of values between levels (as opposed to *e.g.* 2-Lisp).

Infinite regression. The issue of infinite regression in metalevel architectures has long been identified [9, 15]. Chiba, Kiczales and Lamping recognized the ad hoc nature of regression checks, identifying the more general issue of metalevel *conflation* [3]. 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 level clearly separated.

In previous work, we studied similar issues with a particular kind of aspects, which perform structural adaptations (*a.k.a.* inter-type declarations or introductions). We proposed a mechanism of *visibility* of structural changes introduced by aspects [21, 25]. The visibility system, implemented in the Reflex AOP kernel, allows one to declare which aspects see the changes made by which other aspects, or to declare that changes made by an aspects are globally visible or globally hidden. While more flexible that a strict layered architecture like the meta-helix, this system is harder to reason about and specifications can easily conflict with each other. Also,

in this proposal, it is impossible for base level code to hide certain members so they are not visible to (some) aspects.

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. With stratified aspects, aspects are associated with levels, and the scope of pointcuts is restricted to join points of lower levels [2]. The work focuses on advice-triggered reentrancy only, and does not mention the issue related to *e.g.* `if` pointcuts. A more fundamental issue with stratified aspects is that levels are *statically* declared and determined. That is, classes live at level 0, aspects at level 1, meta-aspects at level 2, and so forth. This means that stratified aspects fail to recognize that levels are a property of *execution flows*, not of static declared entities. 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.

Controlling reentrancy. We have already extensively related to our previous on controlling aspect reentrancy [22]. The bottom line is that reentrancy control needs to be based on execution levels in order to avoid unfortunate conflation, in particular when around advice is involved. Dually, allowing to lower aspect computation implies that reentrancy control is a necessity to avoid self-caused loops. The AspectScript language is a first example of a practical aspect-oriented programming language that combines both [17]. Its formalization is future work.

The meta context. Recently, Denker *et al.* introduced the idea of passing an implicit "meta-context" argument to metaobjects such that they can determine at which level they run [8]. This generalizes the idea of the meta-helix and recognizes that levels are a property of execution flows. In their system, metaobjects always run at their level, and execution only shift downs when a metaobject calls `proceed` on the reification of an execution event (*i.e.* a join point in AO terminology). While close to ours, the work really remains in the domain of metalevel architectures and therefore cannot reconcile with the original AO view, according to which advice is base level. Here, in addition, we uncouple level shifting from the behavioral reflection/pointcut-advice mechanism. Finally, the level of execution of activation conditions (the equivalent of pointcut residues in that model) is left unspecified.

8. Conclusion

The issue of conflation in aspect-oriented programming has been latent since its inception. Neither control flow 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. 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, ultimately under control of the programmer.

In order to further empirically validate the usefulness of execution levels, our group is developing AspectScript [17], an expressive aspect-oriented extension JavaScript, with execution levels built in. An AspectJ extension is also under development, in order to study different implementation strategies for execution levels. Finally, in addition to the benefits exposed in this paper, execution levels seem to find application in several other areas. A particularly interesting one is to address the many ambiguities arising from the unwanted interplay of base code and aspects in the presence of exceptions [5].

Acknowledgments. We thank Gregor Kiczales, Paul Leger, and Rodolfo Toledo, for discussions on this topic and proposal, as well as the anonymous reviewers of the Scheme and Functional Programming workshop (one of which brought the issue presented in Section 2.5 to our attention).

References

[1] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, number 3586 in Lecture Notes in Computer Science, pages 144–168, Glasgow, UK, July 2005. Springer-Verlag.

[2] 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.

[3] 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.

[4] 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.

[5] Roberta Coelho, Awais Rashid, Alessandro Garcia, Nélio Cacho, Uirá Kulesza, Arndt Staa, and Carlos Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In Jan Vitek, editor, *Proceedings of the 22nd European Conference on Object-oriented Programming (ECOOP 2008)*, number 5142 in Lecture Notes in Computer Science, pages 207–234, Paphos, Cyprus, july 2008. Springer-Verlag.

[6] 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.

[7] 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.

[8] 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. To appear.

[9] 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.

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] Paul Leger, Rodolfo Toledo, and Éric Tanter. The AspectScript language. http://pleiad.cl/aspectscript, 2009.

[18] Pattie Maes. Concepts and experiments in computational reflection. In Norman Meyrowitz, editor, *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, pages 147–155, Orlando, Florida, USA, October 1987. ACM Press. ACM SIGPLAN Notices, 22(12).

[19] 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.

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

[21] Éric Tanter. Aspects of composition in the Reflex AOP kernel. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113, Vienna, Austria, March 2006. Springer-Verlag.

[22] Éric Tanter. Controlling aspect reentrancy. *Journal of Universal Computer Science*, 14(21):3498–3516, 2008. Best Paper Award of the Brazilian Symposium on Programming Languages (SBLP 2008).

[23] 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.

[24] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th ACM Dynamic Languages Symposium (DLS 2009)*, Orlando, FL, USA, October 2009. ACM Press. To appear.

[25] Éric Tanter and Johan Fabry. Supporting composition of structural aspects in an AOP kernel. *Journal of Universal Computer Science*, 15(3):620–647, 2009.

[26] Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Expressive scoping of distributed aspects. In *Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development (AOSD 2009)*, pages 27–38, Charlottesville, Virginia, USA, March 2009. ACM Press.

[27] 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.

[28] 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.

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