# Higher-Order Aspects in Order

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 pointcuts and advice as 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 (sometimes insufficient) ad-hoc mechanisms, which make aspect-oriented programming more complex. This paper proposes to clarify the situation by introducing explicit 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 give programmers the means to override this default through explicit level shifting expressions. 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 [Masuhara et al. 2003, Wand et al. 2004], as embodied in *e.g.* AspectJ [] and AspectScheme [Dutchyn et al. 2006], 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 [Wand et al. 2004], 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 pointcuts.

* Partially funded by FONDECYT projects 11060493 & 1090083.

While pointcuts were initially conceived of as pure metalevel predicates, advices were seen as a piece of base-level functionality [Wand et al. 2004]. 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 [Maes 1987]. 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 [Denker et al. 2008]. 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 [Wand et al. 2004].

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 [Tanter 2008a]. 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. In a language like AspectScheme, controlling regression in both pointcuts and advice is done using a special primitive (**app/prim**), which makes it possible to apply a function without generating an application join point. This solution however does not scale to join points that are produced in the dynamic extent of the evaluation of pointcuts and advices.

Since all these issues are reminiscent of conflation of levels in reflective architectures [Chiba et al. 1996], 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 *execution levels* are managed explicitly (if needed) in a program. By doing so, we allow different (parts of) pointcuts and advice to be run at different levels, and aspects are bound to observe execution of particular levels. This gives programmers complete control over what aspects see and where they run (*i.e.* who sees them). To alleviate the task for non-expert programmers, we also choose a defensive default that avoids regression. In addition, since we decouple pointcut and advice from execution levels, it becomes possible to use execution level shifting

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

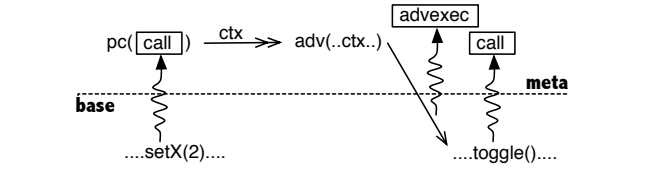


**Figure 1.** Join points and aspect execution in aspect languages with "meta" pointcuts and base-level advice.

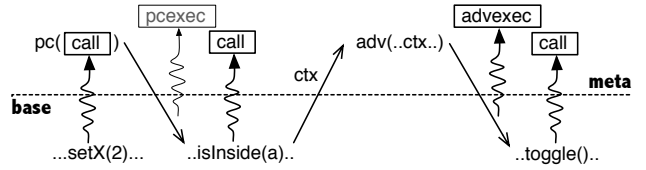


**Figure 2.** Join points and aspect execution in aspect languages with `if` pointcuts or higher-order pointcuts, and base-level advice.

to explicitly move certain parts of a program execution to a higher level, thereby hiding them from lower-level aspects.

This paper is structured as follows: Section 2 describes the current state of affairs regarding aspect weaving, illustrating the issue of conflation. Section 3 introduces execution levels along with a safe default for aspects, which allows aspects of aspects, but prevents an aspect from stepping on its own tail. Section 3.2 gives control back to the programmer by introducing explicit level shifting expressions, which can be used to move any expression evaluation from a level to another. We formalize the operational semantics of our proposal in Section 4, by modeling a higher-order aspect language with explicit execution levels. Section 5 discusses related work and Section 6 concludes. We include the complete formalization of our proposal in Appendix A.

## 2. Background and Motivation

First of all, let us consider a simple example that illustrates some of the issues at stake (in AspectJ). We define an `Activity` aspect that highlights whenever a `Point` object is active, that is, when one of its methods is executing. Furthermore, we are interested in highlighting only points that are inside a pre-determined `area`. This aspect could be defined in AspectJ as follows:

```
public aspect Activity {
  Area area = ...;
  Object around(Point p) :
    execution(* Point.*(..)) && this(p)
    && if(p.isInside(area)){
      p.toggle(); // start highlight
      Object r = proceed(p);
      p.toggle(); // stop highlight
      return r;
  }
}
```

This defines exactly the aspect we are interested in. Note that the `this(p)` pointcut is used to expose the currently executing object `p` to both the `if` pointcut and the advice. The advice toggles highlighting, then `proceed`s, *i.e.* executes the original computation on `p`, gets the result, stops highlighting, and then returns the result.

### 2.1 Issues

The definition of the `Activity` aspect, though natural, is however flawed due to three kinds of reentrancy [Tanter 2008a].

***Base-triggered reentrancy.*** First, if a method of a `Point`, say `move`, internally calls other methods of `Point`, like `setX`, the advice is going to apply several times, yielding repeated toggling of highlighting resulting in incorrect behavior. This is called based-triggered reentrancy. This can be avoided by excluding all join points that are in the control flow of a matched join point. In AspectJ this is done by adding a `!cflowbelow` condition in the pointcut definition, which rules out all join points that occur in the dynamic extent of a matched join point.

***Advice-triggered reentrancy.*** Second, the aspect is subject to advice-triggered reentrancy: highlighting a point object is done by calling its `toggle` method, whose execution is going to be matched by the same aspect, and so on infinitely. To solve advice-triggered reentrancy, similarly to the solution to base-triggered reentrancy, the idea is to exclude join points that occur in the control flow an advice execution. For that, AspectJ includes a specific kind of join point kind called "advice execution". A solution therefore consists in extending the above pointcut as follows:

```
execution(* Point.*(..)) && ...
  && !cflow(adviceexecution());
```

Figure 1 depicts the situation. 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, which returns either false (if there is no match), or a list of bindings (`ctx`) if there is a match. The evaluation of the pointcut occurs entirely at the meta-level [Wand et al. 2004] (without considering `if` pointcuts). The bindings are used to expose context information to the advice. The advice is then called, and runs at the base level. This means another call occurring in the dynamic extent of the advice execution is reified as a call join point, just as visible as the first one.

***Pointcut-triggered reentrancy.*** The definition of the `Activity` aspect also fails because of pointcut-triggered reentrancy: calling `isInside` on the point object within the `if` pointcut results in another method execution that is potentially matched by the aspect. This is because `if` pointcuts are evaluated at the base level in AspectJ. In a higher-order aspect language like AspectScheme, this problem is exacerbated by the fact that there is no such thing as a special `if` pointcut, rather, a pointcut is just a function that runs –as any other– at the base level. The situation with higher-order aspects (and first-order aspects with `if` pointcuts) is depicted on Figure 2: execution of (part of) the pointcut is performed at the base level, therefore the join points produced while executing the pointcut are visible exactly like any other. Consider the following code in AspectScheme.

```
(deploy (let ((area ...))
          (lambda (jp)
             (let ((x (jp-arg 0 jp)))
               (and (Point? x)
                    (is-inside x area)))))
        trace)
```

This (simplified) code defines a pointcut that matches all function applications where the first argument is a value matching the `Point?` predicate, if ever the point resides within a given area. Since the pointcut function runs at the base level, the applications of both `Point?` and `is-inside` generate a call join point that is matched against the same pointcut function, infinitely. In

AspectScheme, one would have to use the special `app/prim` form to apply functions in a way that does not generate join points.

### 2.2 Controlling Reentrancy

The analysis of the three kinds of reentrancy was formulated in a previous article [Tanter 2008a]. We show that reentrancy can be avoided using well-known patterns (*i.e.* `cflow` checks). However, adding these checks to pointcut definitions makes them much more complex than they should be. Also, current AspectJ compilers (ajc and abc [Avgustinov et al. 2006]) make it impossible to get rid of pointcut-triggered reentrancy without completely refactoring the aspect definition, because they hide join points that occur *lexically* in `if` pointcuts.

We therefore proposed a revised semantics for `if` pointcuts, such that their execution is fully visible to all aspects. In addition, we make it clear that, similarly to the advice case, it is necessary to have a pointcut execution join point in order to discriminate the join points that are produced by pointcut evaluation, and therefore getting rid of pointcut-based reentrancy.

Finally, we adopt a new default semantics according to which an aspect never sees a reentrant join point. This implies that the above definition of `Activity` is correct *as is* with our modified semantics. We introduce means to control reentrancy at a more fine-grained level, as required.

While reentrancy control solves the issues of self-references, it does not solve the more general problem of mutual visibility among aspects. For instance, let us consider a second aspect, `Mirroring`, in charge of maintaining mirrors of certain point objects:

```
aspect Mirroring {
  void around(Point p) :
    execution(* Point.set*(..)) && this(p) {
      proceed(p);
      proceed(lookupMirror(p));
    }
}
```

Whenever a state changing method (denoted with the syntactic method pattern `set*(..)`) executes, the aspect not only proceeds with the original object, but also proceeds with the mirror, in order to keep it in sync with the original point. The fact that AspectJ supports this powerful mechanism is reminiscent of reflective method invocation in Java, and in general, metaprogramming[2]. Indeed, `Mirroring` is an aspect that is similar in many respects to what used to be implemented with a typical metaobject protocol [Zimmermann 1996].

When an object is changed, both `Mirroring` and `Activity` aspects match. Using aspect *precedence* declaration, we can make sure that `Activity` runs first, so as to highlight the point object before doing anything else. Recall that the composition of around advices is a nested chaining, such that when `Activity` calls `proceed`, `Mirroring` advice is executed, and when `Mirroring` in turn calls `proceed`, since it is the last advice in the chain, the original computation is performed.

The precedence declaration does not however make it possible to solve another looping issue that appears: when `Mirroring` advice invokes reflectively the method on the mirror point, `Activity` is going to match. If we have reentrancy control as defined in [Tanter 2008a], execution does not enter an infinite loop, but still, the resulting highlighting behavior is incorrect, because `toggle` is applied twice on the mirror point.

[2] http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg03353.html

### 2.3 Conflation

As a matter of fact, all this situation is reminiscent of the issue of meta-circularity, which has long been identified in reflective architectures [des Rivières and Smith 1984]. Broadly from the perspective of reflection, the problem is that of meta-circularity: we are trying to use all the power of higher-order functions to redefine, via pointcuts and advice, the meaning of some function applications. Or, in the case of AspectJ, we are using all the power of Java to implement pointcuts and advice. The proven, but ad hoc, solution to this problem is to add base checks that stop regression, such as `!cflow(adviceexec(..))` in AspectJ, or the default reentrancy control summarized above. Another solution is to introduce a more primitive mechanism that is *not* subject to redefinition, like AspectScheme's **app/prim**.

However, as clearly identified by Chiba *et al.*, these approaches eventually fall short, for they fail to address the fundamental problem, which is that of *conflating* levels that ought to be kept separate [Chiba et al. 1996]. As it turns out, an in-depth inspection of the use of control-flow based checks to avoid reentrancy in AspectJ shows that this mechanism is brittle and does not always work. While we postpone the detailed description of these issues to a revised and extended version of this paper, it is easy to see that `cflow` checks do not help if the advice triggers some behavior in a separate thread. Also, confusion arises due to the fact that, with around advice, the execution of the base code (through `proceed`) is also in the control flow of the advice execution. On Figures 1 and 2, conflation is represented by the fact that all join points (boxes) are present at the same "level", *i.e.* they are all similarly visible to all the defined aspects.

This therefore suggests to place 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 [Chiba et al. 1996, Denker et al. 2008]. On the other hand, this boils down to reconsidering AOP as just a form of metaprogramming, a somewhat unpopular view in the AO community. 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 [Bodden et al. 2006]. However, seeing advice as *inherently* meta defeats the original idea of AOP, where an advice is just another (probably misnamed) piece of code that has the same ontological status as a method [Kiczales 2009].

Recognizing that AOP *can* be (and is) used also for metaprogramming, we propose to resolve this conflict by decoupling the "metaness" concern from the pointcut and advice mechanism. We introduce explicit level shifting in the language, so that programmers can specify their intent with respect to the ontological status of their pointcuts and advices. Also, aspects can be explicitly deployed at higher levels, in order to observe higher-level computation. This said, 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. This arguable choice is purely motivated by a defensive concern: the unaware programmer should not face infinite regression unless she consciously chooses to.

## 3. Execution Levels

In this section, we introduce execution levels and discuss how they can be used in conjunction with aspects. Section 3.1 exposes the default way in which pointcuts and advices are evaluated. Section 3.2 gives more control to programmers by exposing level shifting expressions. Section 3.3 briefly discusses an interesting perspective raised by the introduction of execution levels.

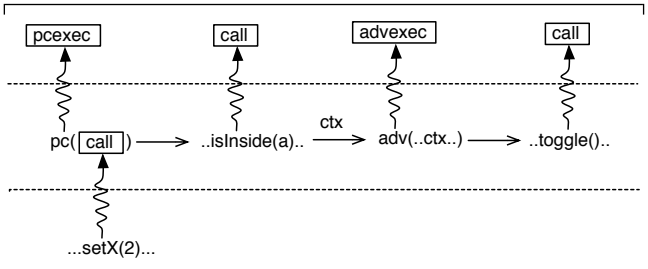


**Figure 3.** Running pointcut and advice at a higher level of execution.

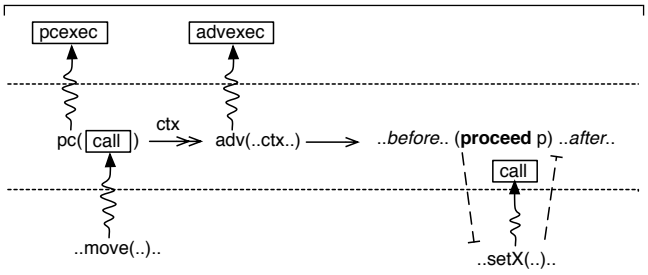


**Figure 4.** Proceeding to the original computation is done at the lower level.

### 3.1 Aspects and Levels: Default

Figure 3 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 `toggle` in the advice.

***Proceed.*** As briefly explained in Section 2.2, an advice can *proceed* to the computation originally described by the join point. When several aspects match the same join point, like `Activity` and `Mirroring` on `setX`, the corresponding advices are chained such that calling proceed in advice $k$ triggers advice $k + 1$. Only when the last advice proceeds is the original computation performed. The original computation clearly belongs to the same level as the original call. This means that in our default semantics, the last call to proceed in a chain of advices runs the original computation at the lower 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.

***Aspects of aspects.*** The default semantics of computing pointcut and advice at a higher-level ensures that other aspects do not see these computations. In our example with `Activity` and `Mirroring`, this is fine for the case of `Mirroring`: the execution of its advice, which performs a reflective invocation on the mirror object, is done at level 1, so the method execution of the mirror is not visible to `Activity`.

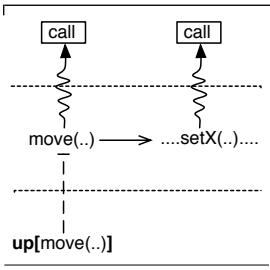


**Figure 5.** Shifting up.

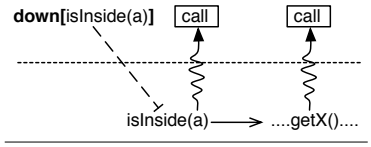


**Figure 6.** Shifting down.

However, this layering also implies that `Mirroring` cannot see the advice computation of `Activity`; therefore the mirror object is not highlighted when it ought to. 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, in [Bodden et al. 2006], 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.

### 3.2 Controlling 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 `Mirroring` 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. In some cases, advice execution should be visible to aspects that observe base level execution. This is the case for instance of the `Activity` aspect: its advice, which highlights points, should be executed at the base level. This allows `Mirroring` to coherently update the mirror objects. This second alternative is more compatible with the traditional AO view that "advices are base". From an engineering viewpoint, it allows the implementor of the `Activity` aspect to declare that some part of its pointcut and/or advice should be considered as standard base code. `Mirroring` does not need to be aware that some other aspect performs computation of interest to it.

***Up and down.*** In order to reconcile both approaches, we introduce explicit level shifting expressions 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 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[3].

***Aspects in order.*** Using **up** and **down**, it is possible to control where aspects are run. One can either use these level shifting expression directly within the definitions of pointcut and advice functions, or use wrappers that embed a whole function in a level shifting expression.

For instance, suppose that the computation of `is-inside` in the pointcut of `Activity` is expensive, and that a memoization aspect is used. It is possible to move the computation of `is-inside` down

[3] For completeness, we consider that execution starts at level 0 and that evaluating a **down** expression at level 0 has no effect.

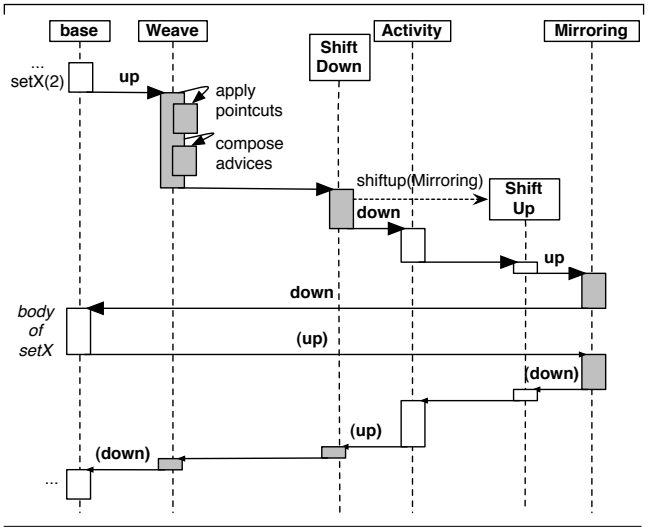


**Figure 7.** Level shifting with two advices, Activity and Mirroring. Level 0 execution is white, level 1 execution is grey.
(Backward thin arrows are "returns", with their associated level shift.)

to the base level, while still ensuring that the check of the `Point?` predicate is not considered base level computation:

```
(deploy (let ((area ...))
          (lambda (jp)
             (let ((x (jp-arg 0 jp)))
               (and (Point? x)
                    (down (is-inside x area))))))
        trace)
```

We can also define higher-order wrapper functions, like `shift-up`, which takes a function and returns a new function that runs the given function one level above:

```
(define (shift-up f)
  (lambda args (up (apply f args))))
```

It is possible to depart from the chosen default semantics, to express the original AO view according to which pointcuts are metalevel predicates and advice is base code. We can use a `deploy-aj` sugar defined as:

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

Note that here we cannot simply use `shift-down` (similar to `shift-up` defined previously) to transform the advice. Indeed, multiple advices are chained together by means of `proceed`. In a higher-order aspect language , an advice is a function that takes a `proceed` function, context information, and a variable number of arguments at the join point [Dutchyn et al. 2006]. The `proceed` function is 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. One should rather use the advice shifting function below:

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

`adv-shift-down` ensures that the execution levels are maintained appropriately 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`.

Figure 7 shows the evolution of control flow with our two aspects `Activity` and `Mirroring` that both apply on a call to `setX`. The evaluation of pointcuts is done at level 1, since this is the default. The advices that apply are chained. In this example, the `Activity` advice has been wrapped (using `adv-shift-down`) so as to execute at level 0. When the Shift Down wrapper runs, it creates a Shift Up wrapper (with `shift-up`) for the following advice in the chain. Therefore, when the `Activity` advice proceeds, execution shifts up, and the `Mirroring` advice runs at level 1 (the default). When it proceeds, since it is the last advice, the body of `setX` is run, at level 0.

### 3.3 Level Shifting and Information Hiding

By moving pointcuts and advices up and down, one actually controls 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 pointcuts and advice bodies. This mechanism can therefore be used to run any arbitrary piece of code at another level of execution[4].

For instance, if a function invokes a security manager each time it is applied in order to ensure that its execution is authorized, it 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 [Aldrich 2005], only join points explicitly exposed though pointcuts of the interface of a module are visible to aspects of other modules. This connection suggests interesting extensions of our work towards a flexible notion of "execution domains" (not necessarily sequential levels) that could be used for similar purposes.

Also, the level-shifting operators **up** and **down** are relative only, making it possible to shift execution one level up or down, respectively. It remains to be determined through practical experience whether these operators are sufficient. One could indeed 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. The semantics we present in the following section does not consider these operators, though it would be straightforward to accomodate them.

## 4. 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 consolidated formal description of the language is provided in appendix.

Figure 8 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.

[4] In this work, level shifting is useful only in the presence of aspects, since only aspects are sensitive to levels.

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

**Figure 8.** Syntax of the core language.

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

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

**Figure 9.** Shifting execution levels.

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 $l \in \mathscr{L}$, 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 [Matthews and Findler 2008] and not presented here. See the appendix for details.

In the following we describe the semantics of execution levels, the join point stack, aspects and their deployment, and 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.

### 4.1 Execution Levels

The language supports explicit execution level shifting forms, **up** and **down** (Figure 9). 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 shift down expression is done similarly (see rules InDown and OutDown).

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

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

**Figure 10.** The join point stack.

### 4.2 Join Point Stack

We follow [Clifton and Leavens 2006] in the modeling of the join point stack (Figure 10). 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 $v\cdots$. We consider three kinds of join points, `call` for function applications, `pc` for pointcut executions, and `adv` for advice executions[5].

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 10).

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, the reduction rule for `call` join points is as follows (we will see other kinds later on):

$$
\begin{array}{rl}
\langle l, J, E[((\lambda(x\cdots)\ e)\ v\cdots)]\rangle & \textsc{App} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{call}, (\lambda(x\cdots)\ e), v\cdots\rceil]\rangle &
\end{array}
$$

This means 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 current execution level $l$, its kind `call`, the function definition, and the values passed to it. A later rule 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.

### 4.3 Aspects and Deployment

As described on Figure 11, an aspect is a tuple $\langle l, pc, adv\rangle$ where $l$ denotes the execution level at which it is defined, $pc$ 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 [Dutchyn et al. 2006, Dutchyn 2006], higher-order advice is modeled as a function receiving first a function to apply whenever

[5] For simplicity and conciseness, we do not distinguish call and execution join points here. Our implementation (see Section 4.5) does make this difference, and therefore supports four join point kinds.

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

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

**Figure 11.** Aspects and deployment (global environment $\mathscr{A}$).

$$
\begin{array}{l}
\langle l, J', E[\texttt{jp}\ \lceil l, k, v_\lambda, v \cdots \rceil]\rangle \qquad \textsc{Weave} \\
\quad \hookrightarrow \langle l, J, E[(\texttt{in-jp}\ (\textbf{up}\ (\texttt{app/prim}\ W[\![|\mathscr{A}|]\!]_J\ v \cdots)))]\rangle \\
\qquad\qquad \text{where } J = j + J'
\end{array}
$$

and, with $J = \lceil l, k, (\lambda(x \cdots)\ e), v \cdots \rceil + J'$:

$$
\begin{array}{rcl}
W[\![0]\!]_J & = & (\lambda(a \cdots) \\
 & & \quad (\textbf{down}\ (\texttt{app/prim}\ (\lambda(x \cdots)\ e)\ a \cdots))) \\
W[\![i]\!]_J & = & (\texttt{app/prim}\ (\lambda(p) \\
 & & \quad (\textbf{if}\ (\textbf{eq?}\ l_i\ l) \\
 & & \quad\quad (\textbf{let}\ ((c\ (\texttt{app/pc}\ pc_i\ J))) \\
 & & \quad\quad\quad (\textbf{if}\ c \\
 & & \quad\quad\quad\quad (\lambda(a \cdots)(\texttt{app/adv}\ adv_i\ p\ c\ a \cdots)) \\
 & & \quad\quad\quad\quad p)) \\
 & & \quad\quad p)) \\
 & & \quad\quad W[\![i-1]\!]_J))
\end{array}
$$

**Figure 12.** Aspect weaving, with level shifting.

the advice wants to proceed, a list of values exposed by the pointcut, and the arguments passed at the original join point.

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 11). 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[6] (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 [Tanter 2008b]. When an aspect is deployed, it captures its execution level of definition. This means that, when executing at level $n$, (**deploy** $p\ a$) deploys the aspect such that it sees join points representing execution of level $n$, and (**up** (**deploy** $p\ a$)) deploys the aspect a level above, such that it sees join points that denote execution at level $n + 1$.

### 4.4 Weaving

We now turn to the semantics of aspect weaving. The WEAVE rule describes the process. A `jp` expression is converted 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

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

$$
\begin{array}{l}
\langle l, J, E[((\lambda(x \cdots)\ e)\ v \cdots)]\rangle \qquad \textsc{App} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{call}, (\lambda(x \cdots)\ e), v \cdots \rceil]\rangle \\
\langle l, J, E[(\texttt{app/pc}\ (\lambda(x \cdots)\ e)\ v \cdots)]\rangle \qquad \textsc{AppPc} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{pc}, (\lambda(x \cdots)\ e), v \cdots \rceil]\rangle \\
\langle l, J, E[(\texttt{app/adv}\ (\lambda(x \cdots)\ e)\ v \cdots)]\rangle \qquad \textsc{AppAdv} \\
\quad \hookrightarrow \langle l, J, E[\texttt{jp}\ \lceil l, \texttt{adv}, (\lambda(x \cdots)\ e), v \cdots \rceil]\rangle \\
\langle l, J, E[(\texttt{app/prim}\ (\lambda(x \cdots)\ e)\ v \cdots)]\rangle \qquad \textsc{AppPrim} \\
\quad \hookrightarrow \langle l, J, E[e\{v \cdots / x \cdots\}]\rangle
\end{array}
$$

**Figure 13.** Different kinds of application.

the list of advice functions that match the correct join point properly chained together, to the original arguments. Note that the weaving rule applies uniformly over the different kinds of join points.

Our 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 (Figure 12).

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, by default, pointcuts and advice run at an upper level, the original function runs at its original level of application.

The WEAVE rule uses a number of different application forms, namely `app/pc`, `app/adv` and `app/prim`, described in Figure 13. These forms are introduced to be able to distinguish the different manners of applying a function in the language. Note that these forms are not in user-visible syntax. However, since we use the base language to weave, we need to be able to denote the primitive application of a function, *i.e.* such that it does not trigger a join point. The APPPRIM rule simply performs the classical $\beta_v$ reduction. `app/prim` is used to hide the activity of the pointcut matcher and the advice dispatcher, as well as to perform an actual function application (*i.e.* when all aspects have proceeded, see $W[\![0]\!]_J$)[7].

$W$ uses two dedicated application forms to apply pointcuts and advice, respectively, `app/pc` and `app/adv`. These forms are introduced so as to generate join points of different kinds, corresponding to different uses of a function. Rule APPPC creates a join point of kind `pc`, and rule APPADV creates a join point of kind `adv`.

[6] The complete semantics given in the appendix properly includes the aspect environment in the evaluation steps.

[7] Note that contrary to AspectScheme, thanks to management of execution levels, it is not necessary for `app/prim` to be in user-visible syntax.

### 4.5 Availability

We have defined the complete semantics of our language using PLT Redex, a domain-specific language for specifying reduction semantics [Felleisen et al. 2009]. The automatically-generated rendering of the complete language grammar, reduction relation, and weaving metafunction $W$ are given in Appendix A. The full definition, as well as a number of executable test cases can be obtained from: `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, in addition to pointcut and advice execution, and includes the different scoping semantics for aspects (statically and dynamically scoped) in addition to global, top-level deployment. It also includes level shifting forms (as macros that handle a dynamically-scoped parameter).

## 5. 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 [Smith 1982] with 2-Lisp and 3-Lisp, and different flavors of it were subsequently explored, with languages like Brown [Wand and Friedman 1988] and Blond [Danvy and Malmkjaer 1988].

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[8]. 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.

In this framework, one could describe the pointcut-advice mechanism as follows, at least in its original form [Wand et al. 2004]. 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 operations 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 operations 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).

[8] Interestingly, 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.*** The issue of infinite regression in metalevel architectures has been long identified [des Rivières and Smith 1984, Kiczales et al. 1991]. Chiba, Kiczales and Lamping recognized the ad hoc nature of regression checks, identifying the more general issue of metalevel *conflation* [Chiba et al. 1996]. 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 [Tanter 2006, Tanter and Fabry 2009]. 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 [Bodden et al. 2006]. 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.

***The meta context.*** Recently, Denker *et al.* introduced the idea of passing an implicit "meta-context" argument to meta-objects such that they can determine at which level they run [Denker et al. 2008]. 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.

***Controlling reentrancy.*** In previous work, we analyzed the issue of unwanted applications of aspects in a general setting [Tanter 2008a]. We identify three kinds of aspect reentrancy: base-triggered reentrancy (*e.g.* caused by a recursive program), pointcut-triggered reentrancy (*e.g.* caused by an `if` pointcut), and advice-triggered reentrancy. After showing the somewhat surprising strategies of current AspectJ compilers with respect to `if` pointcuts, we propose a safe default semantics for aspects, according to which the activity of an aspect is invisible to itself, and an aspect is immune to iterative/recursive refactorings. To deal with pointcut-triggered reentrancy, we introduce the pointcut execution

join point. We also allow for well-defined scoped reentrancy to be introduced, for instance to match reentrant join points whenever the executing objects differ. The reentrancy control proposal does not deal with levels at all. It is only concerned with properly dealing with the conflation when it occurs (as indicated by the original AO view). In fact, in an aspect language with execution levels as proposed in this paper, reentrancy control is still needed, for cases where the programmer causes conflation to occur (*e.g.* by running an advice at a lower level).

## 6. Conclusion

We have proposed a higher-order aspect language design in which pointcuts and advices are regular functions and yet, by default, infinite regression never occurs. This is done by introducing a notion of *levels of execution* that help discriminate the context in which functions are being used. Explicit level shifting expressions make it possible to control the visibility of computation. 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. We are working on adding execution levels to practical aspect languages like AspectJ in order to empirically validate the usefulness of the proposed design.

***Acknowledgments.*** We thank Gregor Kiczales for discussions on this topic and proposal, as well as the anonymous reviewers for their insightful comments and suggestions.

## References

[Aldrich 2005] Aldrich, J. (2005). Open modules: Modular reasoning about advice. In Black, A. P., 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. Springer-Verlag.

[Avgustinov et al. 2006] Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., and Tibble, J. (2006). 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.

[Bodden et al. 2006] Bodden, E., Forster, F., and Steimann, F. (2006). Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition.

[Chiba et al. 1996] Chiba, S., Kiczales, G., and Lamping, J. (1996). 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.

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

[Danvy and Malmkjaer 1988] Danvy, O. and Malmkjaer, K. (1988). 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. ACM Press.

[Denker et al. 2008] Denker, M., Suen, M., and Ducasse, S. (2008). The meta in meta-object architectures. In *Proceedings of TOOLS Europe*, Lecture Notes in Business and Information Processing, Zurich, Switzerland. Springer-Verlag. To appear.

[des Rivières and Smith 1984] des Rivières, J. and Smith, B. C. (1984). The implementation of procedurally reflective languages. In *Proceedings of the Annual ACM Symposium on Lisp and Functional Programming*, pages 331–347.

[Dutchyn 2006] Dutchyn, C. (2006). *Dynamic Join Points: Model and Interactions*. PhD thesis, University of British Columbia, Canada.

[Dutchyn et al. 2006] Dutchyn, C., Tucker, D. B., and Krishnamurthi, S. (2006). Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239.

[Felleisen et al. 2009] Felleisen, M., Findler, R. B., and Flatt, M. (2009). *Semantics Engineering with PLT Redex*. The MIT Press. To appear.

[Kiczales 2009] Kiczales, G. (2009). Personal communication.

[Kiczales et al. 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press.

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

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

[Matthews and Findler 2008] Matthews, J. and Findler, R. B. (2008). An operational semantics for Scheme. *Journal of Functional Programming*, 18(1):47–86.

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

[Tanter 2006] Tanter, É. (2006). Aspects of composition in the Reflex AOP kernel. In Löwe, W. and Südholt, M., 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. Springer-Verlag.

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

[Tanter 2008b] Tanter, É. (2008b). 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. ACM Press.

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

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

[Wand et al. 2004] Wand, M., Kiczales, G., and Dutchyn, C. (2004). A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910.

[Zimmermann 1996] Zimmermann, C. (1996). *Advances in Object-Oriented Metalevel Architectures and Reflection*. CRC Press.

## A. Complete definition of the semantics

This appendix includes the complete grammar of the language (Figure 15), the weaving function $W$ (Figure 14), and the complete reduction relation (Figure 16), all automatically generated from the PLT Redex definition.

```
      e ::= (e e ...) | (if e e e) | x | v
          | (let ((x e) ...) e)
          | (begin e e ...)
          | (up e) | (down e)
          | (in-up e) | (in-down e)
          | (jp j) | (in-jp e)
          | (f e e ...)
      f ::= app/pc | app/adv | app/prim
      v ::= (λ (x ...) e) | (list v ...) | number | #t | #f
          | string | prim | unspecified
   prim ::= + | - | list | cons | car | cdr | empty? | eq? | deploy
      P ::= (l J A E)
      l ::= number
      J ::= (stack j ...)
      j ::= (l k v v ...)
      k ::= "call" | "pc" | "adv"
      A ::= (asps a ...)
      a ::= (l v v)
      E ::= (v ... E e ...)
          | (if E e e)
          | (begin E e e ...)
          | (in-up E) | (in-down E) | (in-jp E)
          | (f v ... E e ...)
          | []
      x ::= variable-not-otherwise-mentioned
```

**Figure 15.** Complete grammar (generated automatically from PLT Redex).

```
W[[(l1 k (λ (x ...) e) v ...), (asps (l2 v1 v2) a ...)]] = (app/prim (λ (p)
                                                            (if (eq? l1 l2)
                                                              (let ((c (app/pc v1 (list k (λ (x ...) e) v ...))))
                                                                (if c
                                                                  (λ (x ...) (app/adv v2 p c x ...))
                                                                  p))
                                                              p))
                                                          W[[(l1 k (λ (x ...) e) v ...), (asps a ...)]] )
W[[(l1 k (λ (x ...) e) v ...), (asps)]]                 = (λ (x ...) (down (app/prim (λ (x ...) e) x ...)))
```

**Figure 14.** Weaving meta-function (generated automatically from PLT Redex).

$P[(+\ number_1\ number_2)] \longrightarrow P[(+\ number_1\ number_2)]$ [+]

$P[(-\ number_1\ number_2)] \longrightarrow P[(-\ number_1\ number_2)]$ [-]

$P[(\text{if } \#f\ e_1\ e_2)] \longrightarrow P[e_2]$ [if-f]

$P[(\text{if } v_1\ e_1\ e_2)] \longrightarrow P[e_1]$ [if-t]
where $v_1$

$P[(\text{let } ((x\ e)\ ...)\ e_0)] \longrightarrow P[(\text{app/prim } (\lambda\ (x\ ...)\ e_0)\ e\ ...)]$ [let]

$P[(\text{begin } v\ e_1\ e_2\ ...)] \longrightarrow P[(\text{begin } e_1\ e_2\ ...)]$ [seq]

$P[(\text{begin } e)] \longrightarrow P[e]$ [outseq]

$P[(\text{eq? } v_1\ v_2)] \longrightarrow P[\#t]$ [eq-t]
where $(\text{eq? } v_1\ v_2)$

$P[(\text{eq? } v_1\ v_2)] \longrightarrow P[\#f]$ [eq-f]
where $(\text{not } (\text{eq? } v_1\ v_2))$

$P[(\text{cons } v_0\ (\text{list } v_1\ ...))] \longrightarrow P[(\text{list } v_0\ v_1\ ...)]$ [cons]

$P[(\text{car } (\text{list } v_1\ v_2\ ...))] \longrightarrow P[v_1]$ [car]

$P[(\text{cdr } (\text{list } v_1\ v_2\ ...))] \longrightarrow P[(\text{list } v_2\ ...)]$ [cdr]

$P[(\text{empty? } (\text{list}))] \longrightarrow P[\#t]$ [empty?-t]

$P[(\text{empty? } v_1)] \longrightarrow P[\#f]$ [empty?-f]
where $(\text{not } (\text{equal? } v_1\ (\text{list})))$

$(l_1\ J\ A\ E[(\text{up } e)]) \longrightarrow ((\text{add1 } l_1)\ J\ A\ E[(\text{in-up } e)])$ [InUp]

$(l_1\ J\ A\ E[(\text{in-up } v)]) \longrightarrow ((\text{sub1 } l_1)\ J\ A\ E[v])$ [OutUp]

$(l_1\ J\ A\ E[(\text{down } e)]) \longrightarrow ((\text{sub1 } l_1)\ J\ A\ E[(\text{in-down } e)])$ [InDwn]

$(l_1\ J\ A\ E[(\text{in-down } v)]) \longrightarrow ((\text{add1 } l_1)\ J\ A\ E[v])$ [OutDwn]

$(l_1\ J\ (\text{asps } a\ ...)\ E[(\text{deploy } v_1\ v_2)]) \longrightarrow (l_1\ J\ (\text{asps } (l_1\ v_1\ v_2)\ a\ ...)\ E[\text{unspecified}])$ [Deploy]

$P[(\text{app/prim } (\lambda\ (x\ ...)\ e)\ v\ ...)] \longrightarrow P[\text{subst-n}[[(x\ v), ..., e]]\ ]$ [AppPrim]

$(l_1\ J\ A\ E[((\lambda\ (x\ ...)\ e)\ v\ ...)]) \longrightarrow (l_1\ J\ A\ E[(\text{jp } (l_1\ \text{“call”}\ (\lambda\ (x\ ...)\ e)\ v\ ...))])$ [App]

$(l_1\ J\ A\ E[(\text{app/pc } (\lambda\ (x\ ...)\ e)\ v\ ...)]) \longrightarrow (l_1\ J\ A\ E[(\text{jp } (l_1\ \text{“pc”}\ (\lambda\ (x\ ...)\ e)\ v\ ...))])$ [AppPc]

$(l_1\ J\ A\ E[(\text{app/adv } (\lambda\ (x\ ...)\ e)\ v\ ...)]) \longrightarrow (l_1\ J\ A\ E[(\text{jp } (l_1\ \text{“adv”}\ (\lambda\ (x\ ...)\ e)\ v\ ...))])$ [AppAdv]

$(l_1\ (\text{stack } j\ ...)\ (\text{asps } a\ ...)\ E[(\text{jp } (l_1\ k\ (\lambda\ (x\ ...)\ e)\ v\ ...))]) \longrightarrow (l_1\ (\text{stack } (l_1\ k\ (\lambda\ (x\ ...)\ e)\ v\ ...)\ j\ ...)\ (\text{asps } a\ ...)\ E[(\text{in-jp } (\text{up } (\text{app/prim } W[[(l_1\ k\ (\lambda\ (x\ ...)\ e)\ v\ ...), (\text{asps } a\ ...)]]\ v\ ...)))])$ [Weave]

$(l_1\ (\text{stack } (l_1\ k\ v_0\ v\ ...)\ j\ ...)\ A\ E[(\text{in-jp } v_1)]) \longrightarrow (l_1\ (\text{stack } j\ ...)\ A\ E[v_1])$ [OutJp]

**Figure 16.** Complete set of reduction rules (generated automatically from PLT Redex).