Science of Computer Programming 80 (2014) 311–342

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Execution levels for aspect-oriented programming: Design, semantics, implementations and applications ☆

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

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

HIGHLIGHTS

- We introduce execution levels in order to address the issue of infinite regression of aspect-oriented programs.
- This issue is prevalent in existing AspectJ projects and a textbook code.
- A formalization of execution levels shows that certain kinds of loops are indeed avoided.
- Execution levels can be implemented using different techniques, possibly very efficiently.

ARTICLE INFO

Article history:
Received 19 April 2012
Received in revised form 30 August 2013
Accepted 2 September 2013
Available online 16 September 2013

Keywords:
Aspect-oriented programming
Meta-programming
Infinite regression
Execution levels

ABSTRACT

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

☆ Earlier versions of the main matter of this article appeared in the informal proceedings of the Scheme and Functional Programming Workshop 2009 [48], and in the Proceedings of the 9th International Conference on Aspect-Oriented Software Development [49]. Additional content on exception handling appears in the Proceedings of the Foundations of Aspect Languages Workshop 2011 [24]. Accompanying material and notes are available online: http://pleiad.cl/research/scope/levels. Éric Tanter is partially funded by FONDECYT Project 1110051. Ismael Figueroa is funded by a CONICYT—Chile Doctoral Scholarship. This work was partially funded by the Inria Associate Teams RAPIDS and REAL.

* Corresponding author.
E-mail addresses: etanter@dcc.uchile.cl (É. Tanter), ifiguero@dcc.uchile.cl (I. Figueroa), nicolas.tabareau@inria.fr (N. Tabareau).

0167-6423/$ – see front matter
http://dx.doi.org/10.1016/j.scico.2013.09.002

1. Introduction

In the pointcut-advice model of aspect-oriented programming (AOP) [35,59], as embodied in *e.g.* AspectJ [31] and AspectScheme [21], crosscutting behavior is defined by means of pointcuts and advices. A pointcut is a predicate that matches program execution points, called join points; and an advice is the action to be taken at a join point matched by a pointcut. An aspect is a module that encompasses a number of pointcuts and advices.

A major challenge in aspect language design is to cleanly and concisely express where and when aspects should apply. To this end, expressive pointcut languages have been devised. While pointcuts were initially conceived as purely "meta" predicates that cannot have any interaction with base level code [59], the needs of practitioners have led aspect languages to include more expressive pointcut mechanisms. This is the case of the `if` pointcut in AspectJ, which takes an arbitrary Java expression and matches at a given join point only if the expression evaluates to true. Going a step further, higher-order aspect languages like AspectScheme and AspectScript [53] consider a pointcut as a first-class function like any other, thus giving the full computational power of the base language to express both pointcuts and advices. In these cases pointcut evaluation is performed at the base level, emitting its own join points.

On the other hand, advices were initially seen as a piece of base-level functionality [59]. In other words, an advice is just like an ordinary function or method, that happens to be triggered implicitly whenever the associated pointcut matches. Indeed, considering advice as base-level code clearly distinguishes AOP from runtime meta-object protocols (MOPs), considered by many as the ancestors of this form of AOP.

Because aspects observe evaluation of base computation, evaluating advices and pointcuts at the base level can trigger infinite regression. This is a widely-recognized[1] problem that can happen easily: it is sufficient for the evaluation of an advice or a pointcut to trigger a join point that is potentially matched by the same aspect, either directly or indirectly.

Although the potential for infinite regression is a direct consequence of considering advice and pointcut execution as base computation, most existing solutions are not based on this insight. Instead, most of them rely on control flow checks, which are eventually unable to properly discriminate aspect computation from base computation.

To address this issue we choose to question the basic assumption that pointcut and advice are *intrinsically* either base computation or meta computation. Looking at how programmers use advices, it turns out that while some advices are clearly base code, some are used to implement concerns like synchronization and monitoring, which were previously considered as forms of meta-programming. Inspired by a solution to infinite regression in MOPs [13], we propose a reconciliating approach in which the metaness concern is decoupled from the pointcut-advice mechanism, by introducing a notion of *level of execution* to structure computation in the core execution model. This idea is similar to the work of Bodden and colleagues [9], although with fundamental differences that are discussed later.

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

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

Observe that infinite regression can also be caused by subtle interactions between advice and the base code, or other reasons like an advice that evaluates a non-terminating function. Indeed, execution levels do not avoid all loops that can be caused by aspects. We denote the kind of loops that are addressed by execution levels as *aspect loops*. Intuitively, an aspect loop is "a loop that happens when an aspect matches a join point that is coming from its own activity (either pointcut or advice)". We formally describe aspects loops in Section 6.

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

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