A monadic interpretation of execution levels and exceptions for AOP

Nicolas Tabareau

To cite this version:

Nicolas Tabareau. A monadic interpretation of execution levels and exceptions for AOP. Modularity: AOSD'12, Mar 2012, Postdam, Germany. inria-00592132v2

HAL Id: inria-00592132

https://inria.hal.science/inria-00592132v2

Submitted on 13 May 2011 (v2), last revised 26 Jan 2012 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A monadic interpretation of execution levels and exceptions for AOP

Nicolas Tabareau
INRIA, France
nicolas.tabareau@inria.fr

Abstract

Aspect-Oriented Programming (AOP) started ten years ago with the remark that modularization of so-called crosscutting functionalities is a fundamental problem for the engineering of large-scale applications. Originating at Xerox PARC, this observation has sparked the development of a new style of programming featured that is gradually gaining traction. However, AOP lacks theoretical foundations to clarify new ideas showing up in its wake. This paper proposes to put a bridge between AOP and the notion of 2-category to enhance the conceptual understanding of AOP. Starting from the connection between the λ-calculus and the theory of categories, we provide an internal language for 2-categories and show how it can be used to define the first categorical semantics for a realistic functional AOP language, called MinAML. We then take advantage of this new categorical framework to introduce the notion of computational 2-monads for AOP. We illustrate their conceptual power by defining a 2-monad for Éric Tanter's execution levels—which constitutes the first algebraic semantics for execution levels—and then introducing the first exception monad transformer specific to AOP that gives rise to a non-flat semantics for exceptions by taking levels into account.

1. Introduction

Aspect-Oriented Programming (AOP) [10] promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns. Much of the research on aspect-oriented programming has focused on applying aspects in various problem domains and on integration of aspects into full-scale programming languages such as Java. However, aspects are very powerful and the development of a weaving mechanism becomes rapidly a very complex task. While some research efforts [7, 20, 21] have made significant progress on understanding some of the semantic issues involved, the algebraic explanation of aspect features has never reached the beauty and simplicity of the connection between the λ-calculus and Cartesian closed categories that gave birth to functional programming. We believe that this is the main reason why AOP never found its place in theoretical computer science fields.

Giving a precise meaning to aspects in AOP is a fairly tangled task because the definition of a single piece of code can have a very rich interaction with the rest of the program, whose effect can come up at anytime during the execution. The main purpose of this paper is to formalize this interaction—in particular for two recent sophisticated weaving definitions of aspects and base computation: (1) execution levels [19] that enable to stratify the computation space in order to prevent from basic infinite regression; (2) execution levels with exceptions [4] that enable to avoid unexpected catching of exceptions in this stratified space.

More precisely, we propose to put a bridge between AOP and the notion of 2-category. Starting from the connection between the λ-calculus and category theory [12], we propose to see an aspect as a 2-cell, that is as a morphism between morphisms. In the programming language point of view, this means that an aspect can be seen as a program which transforms the execution of other programs. In this perspective, a weaving algorithm that defines the interaction of a collection of aspects with a given program will be understood as the computation of a normal form in the underlying 2-category of interest. Thus, an algorithm that is usually defined by hand and described coarsely in AOP systems becomes here a basic notion of rewriting theory. This fact will be made explicit by presenting *the first categorical semantics* for a real functional AOP language called MinAML—semantics obtained by translating of a program written in this language into a term of the λ_2-calculus. Then, we show that program evaluation corresponds exactly to normal form computation in the 2-category induced by the λ_2-term.

To capture the more complex weaving of aspects provided by execution levels, we define an extension of the standard notion of computational monads [15] to aspectual com-

[Copyright notice will appear here once 'preprint' option is removed.]

putation. Such 2-monads are given by a 2-dimensional version of the categorical definition of monads. Using monads for AOP enables to give *the first algebraic semantics for execution levels*. Indeed, it appears that execution levels can be understood as normal aspectual computation in presence of a specialization of the state 2-monad that stratifies the computation space. Formally, we extend MinAML with execution levels, $\text{MinAML}_{\text{EL}}$, and show that evaluation in $\text{MinAML}_{\text{EL}}$ corresponds to normal form computation in the λ_2-calculus extended with this new 2-monad. We finally show that execution levels with exceptions can be understood as aspectual computation in presence of a stratified version of the exception monad transformer, thus defining *the first exception monad transformer specific to AOP*. The use of such algebraic semantics should allow to import directly equational reasoning induced by the 2-monad at the language level. We believe that this line of work should turn out to be very fruitful for proving program equivalence in presence of aspects.

2. Aspect-Oriented Programming in the light of 2-categories and 2-monads

λ-calculus and Cartesian closed categories. Category theory and programming languages are closely related. It is now folklore that the typed λ-calculus is the internal language of Cartesian closed categories. Recall that a *category* $\mathcal{C}$ is a class of objects equipped with a class $\mathcal{C}(A, B)$ of morphisms between any two objects A and B of $\mathcal{C}$. Going one dimension higher, a 2-category $\mathcal{C}$ is basically a category in which the class $\mathcal{C}(A, B)$ of morphisms between any two objects A and B is itself a category. In other words, a 2-category is a category in which there exists morphisms

$$f : A \to B$$

between objects , and also morphisms

$$\alpha : f \Rightarrow g$$

between morphisms. The morphisms $f : A \to B$ are called *1-cells* and the morphisms $\alpha : f \Rightarrow g$ are called *2-cells*.

In the computer science point of view, objects of the category correspond to types in the typed λ-calculus and morphisms between objects A and B of the category correspond to λ-terms of type B with (exactly) one free variable of type A. The composition of morphisms corresponds to substitution, a notion that is at the heart of β-reduction—the fundamental rule of the λ-calculus.

This interpretation of the λ-calculus started in the early 80's from the work of John Lambek and Philip Scott [12, 13]. Soon later, Robert Seely proposed a 2-categorical interpretation of the λ-calculus [17] where β-reduction constructs 2-cells between terms and their β-reduced versions. This perspective is in line with the thought that 2-cells can be seen as rewriting rules between morphisms (or terms). This idea has been pushed further by Barnaby Hilken in [5] where he developed a 2-dimensional λ-calculus that corresponds to the free 2-category with lax exponentials.

Seely's interpretation shows how typed λ-calculus can naturally be viewed as a 2-category. In this paper, we define an advised λ-calculus extending the typed λ-calculus with 2-dimensional primitives that enable to describe any 2-cell of a Cartesian closed 2-category. Those additional primitives construct a kind of 2-dimensional terms that we will (by extension) call aspects. The resulting language, called λ_2-calculus, defines an internal language for Cartesian closed 2-category and will be the base of our explanation of aspects in AOP.

AOP and 2-categories. The keystone of this paper is to consider aspects in AOP as 2-cells in a 2-category just as functions (more precisely λ-terms) are interpreted as morphisms in a category. But this simple idea raises interesting and difficult issues: (i) What are the 2-dimensional notions of β-reduction and variables? (ii) How to describe vertical and horizontal composition of a 2-category in the language of typed λ-calculi? Interestingly, while developing an internal language for Cartesian closed 2-categories [18], the author has discovered that Tom Hirschowitz were independently working on the same structure in its study of higher-order rewriting [6]. This should not appear has a surprise because higher-order rewriting and program transformation are closely related. In this paper, we present the calculus in the style of [6] because we found the presentation more elegant and the connection to Cartesian closed 2-categories more precise but the resulting calculus in [6] and in [18] are identical.

Given the λ_2-calculus, it becomes simpler to describe the interaction of an aspect with the rest of a program. Indeed, the 2-dimensional constructors of the λ_2-calculus enable to faithfully describe all situations in which an aspect can be applied to a given program.

Of course, existing AOP languages do not look like the λ_2-calculus so we show how programs of a simple functional language with aspects—introduced by David Walker and colleagues in [20] and called MinAML—can be translated into the λ_2-calculus. As claimed at the beginning of the paper, the semantics of such programs is provided by a *weaving* algorithm that corresponds to the computation of a normal form in the underlying 2-category.

Note that the work of Kovalyov [11] on modeling aspects by category theory is in accordance with the school of category theory for software design. In this paper, category theory is used as a foundational model for programming languages, which is a completely different line of work.

Using monads in AOP. A very interesting and fruitful use of the correspondence between λ-calculi and categories has been to express computational effects—such as side-effects, exceptions or non-determinism—using the categorical notion of monad, leading to computational monads of Eugenio Moggi [15]. This notion of monad has since been used

in functional programming, with Haskell as a spearhead, to accommodate *pure* programs with effects. Namely, given a computational monad T, programs of pure type A are seen as *values* of type A while programs of type TA are seen as *computations* of type A.

Exploiting our connection between λ_2-calculi and 2-categories, we can define a notion of computational 2-monads for AOP as a 2-categorical version of computational monads to extend programs and aspects with effects. We can in particular extend most of existing monads to 2-monads in order to add the corresponding effect in the λ_2-calculus. But most interestingly, this new point of view leads us to the definition of an original 2-monad corresponding to *execution levels*, a notion quite specific to AOP, and a new exception 2-monad transformer that give rise to *execution level with exception*.

Note that monads have already been used in AOP to define effective advices [16]. In this work, advices are encoded as open functions in Haskell and weaving corresponds to the application of a fixpoint combinator on open functions. They can then use monads available in Haskell to add effects to advices (i.e. open functions). Here, we propose to define 2-monads at the top of any functional language with aspects, without the short-hand of defining advices as open functions. In this way, 2-monads can be constructed directly to enhance AspectScheme, AspectML or any AOP functional language.

A monad for execution levels. Execution levels have been proposed by Éric Tanter [19] to structure program computations into *levels*. This stratification enables to prevent infinite regression and unwanted interference between aspects when aspectual computation is visible to all aspects—including themselves. The standard behaviour is that computation happening at level n produces join points observable by aspects deployed at level $n+1$ only. This means that an aspect can not see its own join points anymore.

Interestingly, we can define a computational 2-monad for execution levels by

$$\mathbf{EL}(A) \quad = \quad \mathbb{N}\mathrm{at} \to (A \times \mathbb{N}\mathrm{at}) \tag{1}$$

that corresponds to a particular case of the so-called *state monad* for $S = \mathbb{N}\mathrm{at}$. For that 2-monad, a value is a simple computation oblivious of the current level of execution, whereas a computation from A to B has type (after uncurrying)

$$A \times \mathbb{N}\mathrm{at} \to B \times \mathbb{N}\mathrm{at}$$

and can thus adapt its behaviour to the level and even change the current level of execution.

The idea is that the notion of execution level introduced in [19] can be translated directly in the calculus induced by $\mathbf{EL}$, where execution levels are managed abstractly through the monadic interpretation. For instance, $\mathbf{up}$ and $\mathbf{down}$ operations of type $\mathbf{EL}(\mathbf{Unit})$ can be defined using the successor and predecessor functions on $\mathbb{N}\mathrm{at}$. This corresponds to level shifting operators defined in [19].

Mixing exceptions and execution levels monads. As it is the case for traditional functional programming language, we can define also define 2-monad transformers, and in particular the exception 2-monad transformer that transform a 2-monad T into a 2-monad

$$T_{\text{EX}}A \quad = \quad T(A + \mathbb{E}) \tag{2}$$

where $\mathbb{E}$ is a set of exceptions. It appears that—when combined with the execution-level 2-monad—the induced notion of exception is *flat* in the sense that they are oblivious of the current level. This means that an aspect can typically catch an exception raised by the base computation and reciprocally, which is in particular what happens in AspectJ. In a recent paper, Ismael Figueroa and Éric Tanter [4] have proposed a calculus where exceptions and execution levels can be mixed in such a way that the basic behavior is the following: an aspect only sees exceptions that have been raised at its level. The default behavior of an advice is thus to catch none of the exceptions raised by the base computation.

At the end of this paper, we show that this calculus with exception can be obtained by combining the execution-level 2-monad $\mathbf{EL}$ with a modified version of the exception 2-monad transformer that takes levels into account.

Plan of the paper. We introduce (§3) the language of Cartesian closed 2-categories, define (§4) the λ_2-calculus, an extension of the λ-calculus with 2-dimensional primitives and present a canonical weaving algorithm based on the computation of a normal form in the underlying 2-category. We then recall a functional AOP language called MinAML (§5), define a translation into the λ_2-calculus and show that weaving in MinAML corresponds to weaving of the translated program in the λ_2-calculus. Finally, we introduce the notion of 2-monad for AOP and use it to define a 2-monad for execution level (§6) and a specialisation of the exception 2-monad transformer (§7) that gives rise to execution levels with exception as introduced in [4].

3. Cartesian Closed 2-Categories in a Nutshell

In this section, we introduce Cartesian closed 2-categories. They are the basis of our conceptual understanding of AOP and we try consequently to explain each construction (2-categories, Cartesian product and closure) in terms of AOP notions. Nevertheless, the reader not comfortable with category theory can skip this section and understand the rest of the paper only in terms of programming language principles.

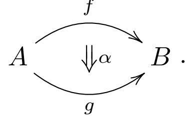
3.1 A glance at 2-categories

2-categories [8] can be viewed as categories enriched over $\mathbf{Cat}$, the category of categories (for more details about enriched category theory, see the monograph of Max Kelly [9]). This means that a 2-category is a category for which the set of morphisms between two objects is itself a category. More

concretely, a 2-category $\mathcal{C}$ has a class of objects (also called 0-cells), usually noted $A, B, \ldots$, a class of morphisms (also called 1-cells) between objects, usually noted $f : A \to B$ and a class of morphisms between morphisms (also called 2-cells), usually noted

$$\alpha : (f \Rightarrow g) :: A \to B$$

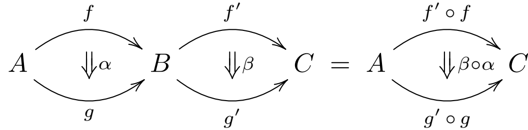
(or simply $\alpha : f \Rightarrow g$ no confusion is possible). A 2-cell $\alpha : (f \Rightarrow g) :: A \to B$ is generally diagrammatically represented as a 2-dimensional arrow between the 1-dimensional arrows f and g



0- and 1-cells form a category called the underlying category of $\mathcal{C}$ – with identity on A denoted by id_A and composition of morphisms f and g denoted by $g \circ f$. 2-cells may be composed "horizontally" and "vertically". We write

$$\beta \circ \alpha : f' \circ f \Rightarrow g' \circ g$$

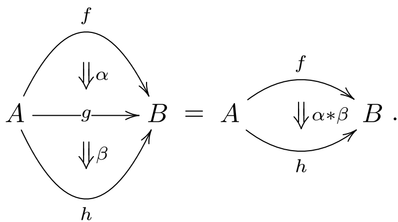
for the horizontal composite of two 2-cells $\alpha : f \Rightarrow g$ and $\beta : f' \Rightarrow g'$, represented diagrammatically as



and we write

$$\alpha * \beta : f \Rightarrow h$$

for the vertical composite of two 2-cells $\alpha : f \Rightarrow g$ and $\beta : g \Rightarrow h$, represented diagrammatically as



From an AOP point of view, horizontal composition corresponds to functional application and vertical composition corresponds to composition of aspects.

The vertical and horizontal composition laws are required to define categories—they are associative and there are identities

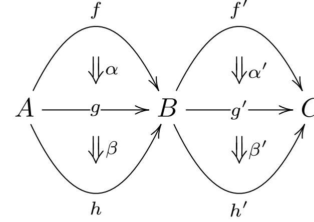
$$\mathbf{1}_f : f \Rightarrow f$$

for each 1-cell $f : A \to B$. The identity for the horizontal composition is given by $\mathbf{1}_{\mathrm{id}_A}$, and one requires that

$$\mathbf{1}_{g \circ f} = \mathbf{1}_g \circ \mathbf{1}_f.$$

Note that the horizontal composition is extended to a composition between a 2-cell α and a 1-cell f by implicitly regarding the 1-cell f as the identity 2-cell $\mathbf{1}_f$.

There is one remaining law of compatibility between the horizontal and the vertical composition. This law, called the *interchange law*, guarantees that the two ways of reading the (labelled pasting) diagram



are equal—which means that

$$(\alpha' * \beta') \circ (\alpha * \beta) = (\alpha' \circ \alpha) * (\beta' \circ \beta).$$

Putting on the AOP hat, this property guarantees that applications of advices at disjoint part of a program do not interfere with each other. This means that we can still reason modularly in presence of aspects as long as the transformation deals with disjoint part of the program.

The different associativity, unit and interchange laws guarantee a fundamental property of 2-categories: *each labelled pasting diagram has a unique composite*. From an AOP point of view, this means that the application of a piece of advice (without side effect) at one point of a program must not perturb the application of other advices at other points of the same program. We will see in Sections 6 and 7 how to allow effectful aspects using 2-monads.

To state precisely the connection between Cartesian closed 2-categories and the λ_2-calculus, we need to introduce the standard notions of functors and natural transformations in a 2-dimensional setting.

There is a canonical notion of morphisms between 2-categories, called *2-functor*. Just as a 2-category is a $\mathbf{Cat}$-category, a 2-functor consists of a functor enriched over $\mathbf{Cat}$. In other words, a 2-functor from $\mathcal{C}$ to $\mathcal{D}$ is a map from i-cells to i-cells (i being 0,1 and 2) that preserves all the structure of a 2-category on the nose. In particular, each 2-functor defines a functor between the underlying categories. As it is the case for functors between categories, there is a notion of *2-natural transformation* transformations between two 2-functors that enables to relate one 2-functor to another.

3.2 Cartesian closed 2-categories

We now present the notion of Cartesian product and closure in a 2-categories that corresponds to the existence of product and functions in functional programming.

A 2-category $\mathcal{C}$ is said to be Cartesian when every pair of objects A_1 and A_2 is equipped with two projection morphisms

$$\pi_1 : A_1 \times A_2 \to A_1 \qquad \pi_2 : A_1 \times A_2 \to A_2.$$

satisfying the following universal property: for every pair of 2-cells

$$\alpha_1 : f_1 \Rightarrow g_1 : X \to A_1 \qquad \alpha_2 : f_2 \Rightarrow g_2 : X \to A_2$$

there exists a unique 2-cells

$$\langle \alpha_1, \alpha_2 \rangle : \langle f_1, f_2 \rangle \Rightarrow \langle g_1, g_2 \rangle : X \to A_1 \times A_2$$

satisfying the two equalities (where i is either 1 or 2)

$$\pi_i \circ \langle \alpha_1, \alpha_2 \rangle \quad = \quad \alpha_i.$$

We also require that $\mathcal{C}$ has a particular object 1, called the *terminal* object, such that there exists a unique 1-cell $\mathbf{skip}_A : A \to 1$ and $\mathbf{1}_{\mathbf{skip}_A} : \mathbf{skip}_A \Rightarrow \mathbf{skip}_A$ is the unique 2-cell of that type. The underlying category of a Cartesian 2-category is also Cartesian.

From an AOP point of view, the object 1 corresponds to the type $\mathbf{Unit}$ and the unique 1-cell of that type is the constant term $\mathbf{skip}$. Then, the unique aspect on $\mathbf{skip}$ is the identity aspect.

A Cartesian 2-category $\mathcal{C}$ is *closed* when it is equipped with a family of functors $\Lambda_{A,B,C} : \mathcal{C}(A \times B, C) \to \mathcal{C}(B, C^A)$ and a family of morphisms $\mathrm{eval}_{A,B} = A \times B^A \to B$ such that

$$\mathrm{eval} \circ (\mathbf{1}_A \times \Lambda(\alpha)) = \alpha \quad \text{and} \quad \Lambda(\mathrm{eval} \circ (\mathbf{1}_A \times \beta)) = \beta$$

for every 2-cell

$$\alpha : f \Rightarrow g : A \times B \to C \quad \text{and} \quad \beta : f' \Rightarrow g' : B \to C^A.$$

Again, we remark that the underlying category of a Cartesian closed 2-category is also Cartesian closed.

4. The λ_2-calculus

In this section, we present the λ_2-calculus—independently studied by Tom Hirschowitz in [6]—and connected to the notion of 2-λ calculi in the sense of Barnaby Hilken [5]. We present the calculus in the style of [6] because we found the presentation more elegant and the connection to Cartesian closed 2-categories more precise. Sections 4.1–4.3 are a synthesis of [6]; more detailed definitions and proofs can be found there. We then focus at the end of this section on the connection to AOP through an abstract definition of the weaving algorithm.

4.1 2-dimensional signatures

A *2-dimensional signature* Σ consists in a set of base types Σ_0, a set of constant terms $\Sigma_1 : X_1 \to \mathbf{L}_0(\Sigma_0)$ indexed by types constructed over Σ_0 and a set of aspects $\Sigma_2 : X_2 \to \mathbf{L}_1(\Sigma_1)_{\|}$ indexed over parallel terms constructed over Σ_1. The set of types $\mathbf{L}_0(\Sigma_0)$ is generated from element S of Σ_0, the unit type, product and arrow types:

$$A ::= S \mid \mathbf{Unit} \mid A \times A \mid A \to A$$

where S is any element of Σ_0. The set of parallel terms $\mathbf{L}_1(\Sigma_1)_{\|}$ is constituted by closed terms (modulo β-η) of the same type obtained from Σ_1 and the 1-dimensional typing rules of Figure 1 corresponding to the traditional λ-calculus.

4.2 λ_2-calculus and permutation equivalence.

Given a 2-signature Σ, we consider the λ_2-calculus $\mathbf{L}(\Sigma)$ whose types are $\mathbf{L}_0(\Sigma_0)$, terms are $\mathbf{L}_1(\Sigma_1)$ and aspects are constructed over Σ_2 using the 2-dimensional typing rules of Figure 1. The typing judgment

$$\Gamma \vdash \alpha : (t \Rightarrow t') :: A$$

says that the aspect α transforms the term t (modulo β-η) of type $\Gamma \vdash A$ into the term t' (modulo β-η) of type $\Gamma \vdash A$; where Γ is a context constituted of a list of a variable and a type, with no variable appearing more than once. All constructions for pairing, abstraction and horizontal composition are extended to aspects and there is a notion of vertical composition $\alpha * \beta$ which corresponds to sequential composition of α and β.

To reflect the structure of Cartesian closed categories, the simply typed λ-calculus is equipped with a notion of equivalence over terms that deals with products and with the closure using the so-called β and η-equivalences. The λ_2-calculus follows the same design principles but is far more bureaucratic. This is not surprising because the notion of Cartesian closed 2-categories requires many diagram commutations that must be captured by equivalences in the calculus. The confluence of such complex equivalences is handle with the use of *permutation equivalences* developed by Sander Bruggink in his study of higher-order rewriting [2]. For instance, the associativity of the vertical composition is reflected by the equivalence

$$\frac{\Gamma \vdash \alpha_i : (t_i \Rightarrow t_{i+1}) :: A \qquad i \in \{1,2,3\}}{\Gamma \vdash (\alpha_1 * \alpha_2) * \alpha_3 \equiv \alpha_1 * (\alpha_2 * \alpha_3) : (t_1 \Rightarrow t_4) :: A}$$

In the same way, the interchange law is captured by the equivalence

$$\frac{\begin{array}{ll}\Gamma \vdash \alpha : (u_1 \Rightarrow u_2) :: A & \Gamma \vdash \beta : (t_1 \Rightarrow t_2) :: A \to B \\ \Gamma \vdash \alpha' : (u_2 \Rightarrow u_3) :: A & \Gamma \vdash \beta' : (t_2 \Rightarrow t_3) :: A \to B\end{array}}{\begin{array}{l}\Gamma \vdash (\beta \circ \alpha) * (\beta' \circ \alpha') \equiv (\beta * \beta') \circ (\alpha * \alpha') \\ \qquad\qquad : (t_1(u_1) \Rightarrow t_3(u_3)) :: B\end{array}}$$

The complete set of permutations equivalences can be found in [6]. The only equivalence that is not a direct translation of the corresponding categorical structure is the β-equivalence

$$\frac{\Gamma, x : A \vdash \alpha : (t_1 \Rightarrow t_2) :: B \qquad \Gamma \vdash \beta : (u_1 \Rightarrow u_2) :: A}{\Gamma \vdash (\lambda x.\, \alpha) \circ \beta \equiv \alpha[\beta/x]}$$

which requires to define a 2-dimensional notion of substitution. Although very intuitive, the definition appears to be a bit technical because it somehow reflects the two equivalent order of composition in the interchange law. As it is not important for our technical development on AOP, we do not present the definition here and refer the interested reader to [6] for details.

1-DIMENSIONAL TYPING RULES.

$$\textsc{Constant}\quad \frac{\Sigma_1(f) = A}{\Gamma \vdash f : A} \qquad \textsc{Variable}\quad \frac{}{\Gamma, x : A, \Delta \vdash x : A} \qquad \textsc{Abstraction}\quad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \textsc{Application}\quad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B}$$

$$\textsc{Bottom}\quad \frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{Unit}} \qquad \textsc{Pairing}\quad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B} \qquad \textsc{Projection}\quad \frac{}{\Gamma \vdash \pi_i^{A_1, A_2} : A_1 \times A_2 \to A_i}$$

2-DIMENSIONAL TYPING RULES.

$$\textsc{Constant}\quad \frac{\Sigma_2(\alpha) = (f \Rightarrow g) :: A}{\Gamma \vdash \alpha : (f \Rightarrow g) :: A} \qquad \textsc{2-identity}\quad \frac{\Gamma \vdash t : A}{\Gamma \vdash t : (t \Rightarrow t) :: A}$$

$$\textsc{2-Pairing}\quad \frac{\Gamma \vdash \alpha : (t \Rightarrow t') :: A \qquad \Gamma \vdash \beta : (u \Rightarrow u') :: B}{\Gamma \vdash \langle \alpha, \beta \rangle : (\langle t, u \rangle \Rightarrow \langle t', u' \rangle) :: A \times B} \qquad \textsc{2-Abstraction}\quad \frac{\Gamma, x : A \vdash \alpha : (t \Rightarrow t') :: B}{\Gamma \vdash \lambda x.\, \alpha : (\lambda x.\, t \Rightarrow \lambda x.\, t') :: A \to B}$$

$$\textsc{2-Application}\quad \frac{\Gamma \vdash \beta : (t \Rightarrow t') :: A \to B \qquad \Gamma \vdash \alpha : (u \Rightarrow u') :: A}{\Gamma \vdash \beta \circ \alpha : (t(u) \Rightarrow t'(u')) :: B} \qquad \textsc{Vertical-composition}\quad \frac{\Gamma \vdash \alpha : (t_1 \Rightarrow t_2) :: A \qquad \Gamma \vdash \beta : (t_2 \Rightarrow t_3) :: A}{\Gamma \vdash \alpha * \beta : (t_1 \Rightarrow t_3) :: A}$$

Figure 1. Typing rules of the λ_2-calculus

4.3 Cartesian closed 2-categories and the λ_2-calculus

The definition above leaves a lot of freedom. There are many λ_2-calculus. Namely, given a 2-signature Σ, the λ_2-calculus constructed on this 2-signature corresponds to the free Cartesian closed 2-category generated by this signature, as stated by the following proposition.

PROPOSITION 1. *The construction* $\mathbf{L}$ *that computes a* λ_2*-calculus from a 2-signature induces a functor* $\mathbf{C}$ *from the category* $\mathbf{2\text{-}Sig}$ *of 2-signatures and signature-preserving morphisms to the category* $\mathbf{2\text{-}CCC}$ *of Cartesian closed 2-categories and strictly structure preserving 2-functors. This functor is the left component of the adjunction*

$$\mathbf{2\text{-}Sig} \underset{\mathbf{S}}{\overset{\mathbf{C}}{\rightleftarrows}} \mathbf{2\text{-}CCC} \quad (\mathbf{C} \dashv \mathbf{S})$$

This means that λ_2-calculi correspond exactly to Cartesian closed 2-categories where no additional equality has been added to structural equalities.

4.4 Weaving in the λ_2-calculus

Using the correspondence between the λ_2-calculus and Cartesian closed 2-categories, we can now define a weaving algorithm in terms of categorical rewriting.

As sketched in the introduction, given a term $t(x) : B$ of a λ_2-calculus $\mathbf{L}(\Sigma)$ where x is of type A, we will consider all the possible interactions of aspects defined in $\mathbf{L}(\Sigma)$ with $t(x)$ by considering the category $\mathbf{C}(\Sigma)(A, B)$. This category contains all aspects that transform terms of type $A \to B$ and so the execution of an aspect corresponds to the application of a morphism in that category. The result of the weaving algorithm is thus given by *all* normal forms of the image of $t(x)$ in that category.

More precisely, the set of woven terms is defined as

$$\begin{aligned}\mathbf{weave}(t(x)) = \{(t'(x), \alpha) \mid\ & (x, t(x)) \xrightarrow{\alpha} (x, t'(x)) \\ & \text{is a maximal reduction in the category } \mathbf{C}(\Sigma)(A, B)\}.\end{aligned}$$

Of course, such a normal form has no reason to be unique or even to exist. The purpose of specific AOP languages is often to provide more advanced definitions of aspects that guarantee the uniqueness and sometimes the existence of such a normal form so that the set $\mathbf{weave}(t(x))$ is a singleton for every term $t(x)$.

Uniqueness of the normal form. Observe that all the work on *aspect composition* can be understood as a way to combine aspects while conserving uniqueness of the definition of the woven program. Indeed, uniqueness of the normal form corresponds to *determinism* of the weaving algorithm, which is most of the time a property required by a programmer.

For example, when multiple advices can be applied at the same join point, precedence orders are (arbitrarily) defined in AspectJ, based on the order in which definitions of advices syntactically appear in the code. Those orders, even arbitrary, enables a programmer to keep track of which as-

pect should be applied at a given time, as soon as he knows the way orders are computed.

More algebraic approaches have been proposed (see e.g. [14]) were explicit operators for advices and aspects composition have been introduced.

Existence of a normal form. The absence of a normal form is often understood as a *circularity* in the application of aspects. This problem is difficult to overcome and can arise even in simple programs. For instance, the work of Éric Tanter on execution levels (reunderstood in Section 6 using 2-monad) is precisely a way to introduce a hierarchy in the application of aspects and thus to avoid basic circular definition. Other lines of work have proposed to restricted the power of advices (for example using a typing system [3]) in order to guarantee that the execution of the program is not critically perturbed.

Weaving on an example. Suppose that we have defined an aspect

$$\alpha : sqrt \Rightarrow sqrt \circ abs$$

which rewrites all calls to a square root function to ensure that inputs are non-negative. . The effect of α on the program

$$p = \lambda x.\ sqrt(sqrt(x))$$

will be described by the composed 2-cell

$$\beta = \lambda x.\ \alpha \circ \alpha \circ x : p \Rightarrow p'$$

that transforms the program p into the program

$$p' = \lambda x.\ sqrt(abs(sqrt(abs(x)))).$$

The aspect β is automatically generated from the aspect α and constructors of the λ_2-calculus. Note that one could argue that this violates one of the primary design goals of AOP, which is to allow separation of cross-cutting concerns. Indeed, each aspect is monomorphic in the sense that the aspect β in the example above is specific to the program p. It could seem unfortunate because an important goal of AOP languages is that the aspect may be oblivious to the target program – in 2-categorical/λ-calculus terms what seems needed is naturality/parametricity in the scaffolding. However, this is not the point of view adopted in the λ_2-calculus. The idea is that a single definition of an aspect will generate all the possible interactions of this aspect with any program.

5. MinAML

This section shows that the weaving process of a concrete AOP language called MinAML can be understood through a translation into the λ_2-calculus.

MinAML is a version (without conditionals and `before` and `after` advice) of the language introduced in [20] to give a first AOP language with a formal semantics. The absence of `before` and `after` advice is unimportant because they can both be encoded with an `around` advice. But the main difference between the original MinAML is that we do not address here the question of scoping of aspects. This scoping mechanism is orthogonal to the question addressed in this paper and would introduce unnecessary complications in definition of the associated λ_2-calculus and of the weaving mechanism. Indeed, the definition of the underlying 2-category associated to a program in MinAML, as well as the corresponding rewriting system, would have to evolve with the change of scope. We have thus decided to omit this mechanism in the definition of the language and work with a global scope.

5.1 Syntax

MinAML is an extension of the λ-calculus with products in two steps. The first extension is usual: we introduce declaration names that can be used to define names for terms of the language with the `let` constructor

$$\texttt{let}\, f = e.$$

We suppose given a set of declaration names, noted $f, g, \ldots$

The second extension is the introduction of aspects with the constructor

$$\texttt{around}\ f(x) = e$$

which indicates that at execution, the application of the function f with argument x is replaced by the term e. Using the terminology standard AOP terminology, the term $f(x)$ defines the *pointcut* of the aspect and the term e defines its *advice*.

When declaring advices, the programmer can choose either to replace f entirely or to perform some computations interleaved with one (or more) execution of f (possibly with new arguments) using the keyword `proceed`. When multiple aspects intercept the same function f, one must define on order in the weaving mechanism. For simplicity, we have decided to choose the order of declaration in the program.

The grammar of MinAML is fully described in Figure 2. A program p is constituted by a list of declarations d_s, a list of aspects a_s and a term e. The fact that there is only a global scope for aspects in our calculus is enforced by the stratified structure of a program. The term $[\ ]$ stands for the empty list, $[h]$ stands for the singleton list with element h and $l \cdot l'$ denotes the concatenation of lists.

5.2 A simple example

Let us now express in this language the example developed in Section 4—of an aspect that ensures that all calls to the $sqrt$ function are performed with non-negative values. The following program of MinAML (where we use some usual primitives on integers) defines such an aspect and run $sqrt$ on the negative value -4.

$$\begin{aligned}\mathbb{P} \ = \ & [\texttt{let}\, sqrt = \lambda x.\ \sqrt{x},\ \ \texttt{let}\, abs = \lambda x.\ |x|] \cdot \\ & [\texttt{around}\ sqrt(x) = \texttt{proceed}(abs(x))] \cdot \\ & [sqrt(-4)]\end{aligned}$$

types	A	$::=$	$S \mid \mathbf{Unit} \mid A \times B \mid A \to B$	aspects	a_s	$::=$	$[\,] \mid [\texttt{around}\, f(x) = e] \cdot a_s$
values	v	$::=$	$\mathbf{skip} \mid \lambda x.\, e$	declarations	d_s	$::=$	$[\,] \mid [\texttt{let}\, f = e] \cdot d_s$
expr.	e	$::=$	$v \mid x \mid \texttt{proceed} \mid e(e)$	programs	p	$::=$	$d_s \cdot a_s \cdot e$

Figure 2. The grammar of the MinAML

5.3 Typing

The typing rules for λ-terms presented in Figure 3 are standard (rules for product and $\mathbf{Unit}$ have been omitted). Programs are typed in the presence of a context $\Gamma; \Delta$. Γ stipulates the type of variables and Δ stipulates the type of declaration names. This dichotomy enables to force free variables appearing in the definition of a piece of advice to be associated with declaration names only. Note that this fact was also enforced by the stratified nature of a program, which is a set of declaration names d_s, then a set of aspects a_s and finally a term t. Thus, when trying to type a name's declaration or an aspect, the context Γ is necessary empty. Nevertheless, we have chosen to make this explicit in the typing so that introducing a more general scoping mechanism would not require any change in the typing rules.

Rule BINDING for the `let` binder requires that the open variables appearing in t are related to declaration names.

In Rule AROUND, one assume that a declaration name f of type $A \to A'$ is already defined in Δ and check that t (where every occurrence of `proceed` is replaced by f) has type A' assuming that the argument x of $f(x)$ has type A and is the only variable in the environment Γ. In that case, the program $\texttt{around}\ f(x) = t; p$ is given the same type as the program p.

It is important that declaration names can only be bound to terms defined on declaration names. In this way, aspects in MinAML are not be able to intercept terms with free variables in the same way as constant aspects in λ_2-calculus cannot be defined between open terms.

5.4 Operational semantics.

We define the operational semantics of MinAML using a reduction relation $\rightharpoonup$ which described a call-by-value small step semantics between configurations of the form $\langle d_s, a_s, e\rangle$. This small step reduction is described by

$$\langle d_s, a_s, E(f(v))\rangle \rightharpoonup \langle d_s, a_s, E(\mathrm{Weave}(a_s, f, d_s))\rangle$$
$$\langle d_s, a_s, E(\lambda x.e(v))\rangle \rightharpoonup \langle d_s, a_s, E(e[v/x])\rangle$$

where E is an evaluation context (i.e. an expression with a hole) and $\mathrm{Weave}(a_s, f, d_s)$ is the weaving function

$$\mathrm{Weave}([\texttt{around}\, f(x) = e] \cdot a_s, f, d_s) = e[\mathrm{Weave}(a_s, f, d_s)/\texttt{proceed}]$$
$$\mathrm{Weave}([\texttt{around}\, f'(x) = e] \cdot a_s, f, d_s) = \mathrm{Weave}(a_s, f, d_s) \qquad (f \neq f')$$
$$\mathrm{Weave}([\,], f, d_s \cdot [\texttt{let}\, f = e] \cdot d'_s) = e$$

The application of a bound name f to a value v is first woven with respect to the list of aspects a_s (in their apparition order) and, when their is no more aspects in the list, each call to `proceed` is replaced by the term bound to f.

5.5 A translation into the pure λ_2-calculus

We now present the translation of a typed program

$$p = d_s \cdot a_s \cdot e$$

into the λ_2-calculus. More precisely, we will define a λ_2-calculus $\mathbf{L}_p$ based on declarations present in d_s and aspects present in a_s and a translation $[\![e]\!]$ of the term e, see Figure 4. The construction of the 2-signature Σ_p of $\mathbf{L}_p$—presented in Figure 4 in Ocaml style—goes in two steps:

(a) we produce a list of aspects $[\![a_s]\!]$ and a mapping γ from declaration names in d_s to integers (Figure 4-a). Because a declaration name f can be intercepted by more than one aspect, we introduce a fresh declaration name f_i each time we translate an aspect whose pointcut relies on f. That is, the ith aspect $\texttt{around}\ f(x) = t$ that intercept f will thus be translated into the constant aspect

$$a_{f_i} : f_i \Rightarrow \lambda x.\ [\![t[f_{i+1}/\texttt{proceed}]]\!]$$

that intercepts f_i and proceeds with f_{i+1}. In this way, we construct a sequence of declaration names that drives the list of aspects that can intercept applications of the function f.

(b) we define a list of aspects $[\![ds]\!]$ (Figure 4-b) by translating each declaration $\texttt{let}\, f = t$ into the constant aspect $a_f : f_{\gamma(f)} \Rightarrow [\![t]\!]$.

For example, the 2-signature $\Sigma_{\mathbb{P}}$ corresponding to $\mathbb{P}$ is given by the constant terms

$$sqrt_1, sqrt_2, abs_1$$

and by the three aspects

$$a_1 : \quad sqrt_1 \Rightarrow \lambda x.\ sqrt_2(abs_1(x))$$
$$a_2 : \quad sqrt_2 \Rightarrow \lambda x.\ \sqrt{x}$$
$$a_3 : \quad abs_1 \Rightarrow \lambda x.\ |x|$$

5.6 Weaving in MinAML

Once the λ_2-calculus $\mathbf{L}_p$ has been generated, the weaving algorithm is defined as in Section 4.4. Namely, we compute the normal form (if it exists) in the corresponding category—observe that the ordering of advices guarantees that there is at most one normal form. The correction of this interpretation is stated by the following proposition.

VARIABLE	NAME	ABSTRACTION	APPLICATION
$\dfrac{}{\Gamma, x:A; \Delta \vdash x:A}$	$\dfrac{}{\Gamma; \Delta, f:A \vdash f:A}$	$\dfrac{\Gamma, x:A; \Delta \vdash e:B}{\Gamma; \Delta \vdash \lambda x.e : A \to B}$	$\dfrac{\Gamma; \Delta \vdash e' : A \to B \quad \Gamma; \Delta \vdash e : A}{\Gamma; \Delta \vdash e'(e) : B}$

BINDING

$$\frac{;\Delta \vdash e : A \quad ;\Delta, f : A \vdash p : B \quad (f \notin \Delta)}{;\Delta \vdash \texttt{let}\, f = e; p : B}$$

AROUND

$$\frac{x : A; \Delta, f : A \to A' \vdash e[f/\texttt{proceed}] : A' \quad ;\Delta, f : A \to A' \vdash p : B}{;\Delta, f : A \to A' \vdash \texttt{around}\, f(x) = e; p : B}$$

Figure 3. Typing rules of MinAML

$$\begin{array}{rcl} \Sigma_p & = & (\{S, \mathbf{Unit}\}, \{f_i \mid f \in d_s \text{ and } 1 \leq i \leq \gamma(f)\}, [\![a_s]\!] \cdot [\![d_s]\!]) \\ [\![e]\!] & = & e[f_1/f] \qquad \text{(for all } f \text{ occurring in } e) \end{array}$$

(a)

```
let trans_asp(a,b) = match (a,b) with
  | ((γ,A),(around f(x) = t)) -> (γ[f ↦ γ(f) + 1],
        A·[ a_{f_{γ(f)}} : f_{γ(f)} ⇒ λx. [[t[f_{γ(f)+1}/proceed]]]])
in (γ,[[a_s]]) = fold_l(trans_asp,(let γ x = 1,[]),a_s)
```

(b)

```
let trans_eq(d) = match d with
     | let f = t -> [a_f : f_{γ(f)} ⇒ [[t]]]
in [[d_s]] = map trans_eq d_s
```

Figure 4. Translation of MinAML into the λ_2-calculus

PROPOSITION 2 (Translation of MinAML). *Given a program $d_s \cdot a_s \cdot e$ of type A, the configuration $\langle d_s, a_s, e\rangle$ reduces to $\langle d_s, a_s, v\rangle$ for some value v of type A if and only if*

$$\mathbf{weave}([\![e]\!]) = \{[\![v]\!]\}.$$

5.7 Weaving on a simple example

Let us now explained the behavior of the weaving algorithm on the simple example $\mathbb{P}$. The computation can be described by the following sequence of reduction (where some extra β-reduction has been performed to make the reading easier):

$$\begin{array}{lcl} sqrt_1(-4) & \xrightarrow{a_1 \circ \mathbb{I}(-4)} & sqrt_2(abs_1(-4)) \\ & \xrightarrow{a_2 \circ \mathbb{I}(abs_1(-4))} & \sqrt{abs_1(-4)} \\ & \xrightarrow{\mathbb{I}(\sqrt{-}) \circ a_3 \circ \mathbb{I}(-4)} & \sqrt{|-4|} = 2 \end{array}$$

REMARK 1. *In the rest of this paper, we suppose that our calculus (and the corresponding 2-category) is equipped with a proper notion of natural numbers of type* $\mathbb{N}\text{at}$ *(with equality, successor (*$\texttt{Succ}(n)$*), predecessor (*$\texttt{Pred}(n)$*) and conditionals (which corresponds to coproducts at the 2-categorical level) written as* '$\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$'*. We use standard primitives on integers and if-then-else constructor in the expected way.*

6. The Execution-Level Monad

In this section, we introduce a 2-dimensional version of Moggi's *computational monads*. To illustrate this new powerful tool for AOP, we extend MinAML with Tanter's execution levels and show that this new calculus can be translated in the λ_2-calculus extended with the execution-level 2-monad **EL**.

6.1 Computational 2-monads

A (strict) *2-monad* [1] can be seen as a monad in the 2-category of 2-categories, 2-functors and 2-natural transformations. In its Kleisli presentation, a 2-monad T on a 2-category $\mathcal{C}$ is a 2-functor from $\mathcal{C}$ to $\mathcal{C}$ equipped with a 2-natural transformation $\eta_A : A \to TA$ and a transformation * that transports every 2-cell $\alpha : (f \Rightarrow g) :: A \to TB$ into the 2-cell $\alpha^* : (f^* \Rightarrow g^*) :: TA \to TB$ satisfying

- $\eta_A^* = \mathrm{id}_{TA}$
- $\alpha^* \circ \eta_A = \alpha \quad$ for $\alpha : (f \Rightarrow g) :: A \to TB$
- $\beta^* \circ \alpha^* = (\beta^* \circ \alpha)^* \quad$ for $\alpha : (f \Rightarrow g) :: A \to TB$ and $\beta : (g \Rightarrow h) :: B \to TC$

For compatibility with products in Cartesian closed 2-categories, we define a *computational 2-monad* as a strict 2-monad equipped with a strength

$$A \times TB \xrightarrow{t_{A,B}} T(A \times B)$$

subject to usual identity and associativity laws.

EXAMPLE 1. *Most of usual computational monads can be extended to the 2-dimensional setting, in particular:*

- **side-effect** *: $TA = S \Rightarrow (A \times S)$, where S is the set of states*
- **exceptions** *: $TA = A + \mathbb{E}$, where $\mathbb{E}$ is the set of exceptions*

This 2-categorical machinery justifies the following notion of computational 2-monad for AOP as a metalanguage for the λ_2-calculus. For the simplicity of the development, we will forget about the underlying *Kleisli interpretation* in the

Kleisli 2-category induced by the 2-monad and only present the results from a programming language point of view.

A *computational 2-monad* on the λ_2-calculus consists in:

- A function T that associates a computational type TA to any type A.
- A lifted aspect
$$[\alpha]_T$$
of type $([e]_T \Rightarrow [e']_T) :: TA$ for any aspect α of type $(e \Rightarrow e') :: A$. The idea is that $[e]_T$ is the computation that simply returns the value e and $[\alpha]_T$ is an aspect between two such computations. This corresponds to the unit of the 2-monad.
- A let-binder
$$\mathtt{let}_T\ x \Leftarrow \alpha\ \mathtt{in}\ \beta$$
of type $((\mathtt{let}_T\ x \Leftarrow e_1\ \mathtt{in}\ e_2) \Rightarrow (\mathtt{let}_x\ e'_1 \Leftarrow e'_2\ \mathtt{in}\ )) :: TB$ for α of type $(e_1 \Rightarrow e'_1) :: TA$ and β of type $(e_2 \Rightarrow e'_2) :: TB$ assuming that x has type A. The idea is that $\mathtt{let}_T\ x \Leftarrow e_1\ \mathtt{in}\ e_2$ is a computation that evaluates e_1 first and binds the result to x in e_2 and $\mathtt{let}_T\ x \Leftarrow \alpha\ \mathtt{in}\ \beta$ is an aspect between two such computations. This corresponds to the composition $\beta^* \circ \alpha$ in the Kleisli 2-category.
- A function
$$\mathtt{run}_{T,A} : TA \to A$$
that runs a computation of type A and returns its result.

Those new constructions are subject to equality that mimics the commutative diagram in the categorical setting (see [15] for details). Depending on the considered 2-monad, there can be new constructors in the language that enable to handle the computation effect. For instance, for the state 2-monad, there are functions `lookup` and `update` that respectively returns and updates the current state. In the case of the exception 2-monad, there are `raise` and `handle` constructors that respectively raises and catches an exception.

6.2 MinAML$_{\mathrm{EL}}$: MinAML with execution level

We now extend MinAML with a notion of execution levels as defined in [19]. As explained in introduction, the idea behind execution levels is to enhance computation with an integer that represents the *current level* of execution. This level is then used to make join points produced by computation happening at level n observable by aspects deployed at level $n + 1$. Note that in the original paper, the deployment of aspects is entirely dynamic, whereas the deployment in MinAML is static. This is not an issue because the management of execution levels in MinAML can still be dynamic. This shows by the way that a dynamic deployment of aspects is not required to have a proper notion of dynamic execution levels.

MinAML$_{\mathrm{EL}}$ is an extension of MinAML with four new constructors

$$e ::= \dots \mid \mathtt{up}(e) \mid \mathtt{down}(e) \mid \mathtt{in_up}(e) \mid \mathtt{in_down}(e)$$

where $\mathtt{in_up}(e)$ and $\mathtt{in_down}(e)$ are only here to define the operational semantics and are not user-visible. The typing system is extended straightforwardly to those new constructors.

Operational semantics of MinAML$_{\mathrm{EL}}$ is defined between configurations of the form $\langle l, d_s, a_s, e\rangle$, where l is the current level of execution. Because the deployment of aspects is static, we can simplify the semantics. By default an aspect is always deployed at level 1 and deployment at a higher level l must be explicitly express by the user using

$$\mathtt{around}\ (l, f(x)) = e.$$

The small step semantics is extended accordingly

$$\begin{array}{l}
\langle l, d_s, a_s, E(\mathtt{up}(e))\rangle \rightharpoonup \langle \mathtt{Succ}(l), d_s, a_s, E(\mathtt{in_up}(e))\rangle \\
\langle l, d_s, a_s, E(\mathtt{in_up}(v))\rangle \rightharpoonup \langle \mathtt{Pred}(l), d_s, a_s, E(v)\rangle \\
\langle l, d_s, a_s, E(\mathtt{down}(e))\rangle \rightharpoonup \langle \mathtt{Pred}(l), d_s, a_s, E(\mathtt{in_down}(e))\rangle \\
\langle l, d_s, a_s, E(\mathtt{in_down}(v))\rangle \rightharpoonup \langle \mathtt{Succ}(l), d_s, a_s, E(v)\rangle \\
\langle l, d_s, a_s, E(f(v))\rangle \rightharpoonup \\
\qquad \langle l, d_s, a_s, E(\mathtt{up}(\mathrm{Weave}(a_s, (\mathtt{Succ}(l), f), d_s)))\rangle
\end{array}$$

where $\mathrm{Weave}(a_s, (l, f), d_s)$ is extended as

$$\begin{array}{l}
\mathrm{Weave}([\mathtt{around}\ (l, f(x)) = e] \cdot a_s, (l, f), d_s) = \\
\qquad e[\mathrm{Weave}(a_s, (l, f), d_s)/\mathtt{proceed}] \\
\mathrm{Weave}([\mathtt{around}\ (l', f'(x)) = e] \cdot a_s, (l, f), d_s) = \\
\qquad \mathrm{Weave}(a_s, (l, f), d_s) \quad (f \neq f' \text{ or } l \neq l')
\end{array}$$

The reduction of $\mathtt{up}(e)$ increases the level and places the marker `in_up` in the execution context. When the nested expression is reduced to a value, this marker is disposed while decreasing the level back (and dually for $\mathtt{down}(e)$ and $\mathtt{in_down}(e)$). The weaver is a modified version of the weaver of Section 5.4 where the level at which the join point as been emitted is checked to match the level of deployment of the aspect.

Note that the only difference with execution levels defined by Éric Tanter is the absence of level-capturing functions. They could be easily added to MinAML$_{\mathrm{EL}}$ but their translation in the monadic language would require non-canonical constructors whose definitions have been left for future work.

6.3 The execution-level monad

It is possible to define a 2-monad on the λ_2-calculus in order to recover execution levels and interpret MinAML$_{\mathrm{EL}}$. As explained in introduction, **EL** (defined in Equation (1)) is a restriction of the state 2-monad where the state only contains information on the current level. The lifting, let-binder and run function are given by

$$\begin{array}{lcl}
[\alpha]_{\mathbf{EL}} & = & \lambda n.\, (\alpha, n) \\
\mathtt{let}_{\mathbf{EL}}\ x \Leftarrow \alpha\ \mathtt{in}\ \beta & = & \lambda n.\, \mathtt{let}\ (a, n') \Leftarrow \alpha \circ n\ \mathtt{in} \\
 & & \qquad (\beta \circ a) \circ n' \\
\mathtt{run}_{\mathbf{EL},A}\ c & = & c(0)
\end{array}$$

and we can define three operations specific to the execution-level 2-monad that respectively returns, upgrades or downgrades the current level:

$$\begin{array}{llcl} lookup: & \mathbf{EL}(\mathbb{N}\mathrm{at}) & = & \lambda n.\ (n, n) \\ up: & \mathbf{EL}(\mathbf{Unit}) & = & \lambda n.\ (\mathbf{skip}, \mathtt{Succ}(n)) \\ down: & \mathbf{EL}(\mathbf{Unit}) & = & \lambda n.\ (\mathbf{skip}, \mathtt{Pred}(n)) \end{array}$$

6.4 Interpreting MinAML$_{\mathrm{EL}}$ using the execution-level monad

We now translate MinAML$_{\mathrm{EL}}$ into the λ_2-calculus extended with $\mathbf{EL}$. The structure of the translation is the same as the translation of MinAML given in Figure 4, the only differences take place in the definition of $[\![e]\!]_{\mathbf{EL}}$ and the translation of $\mathtt{around}\ (l, f(x)) = e$. The translation of a term of MinAML is given by lifting and the translation of applications is given by the let-binder

$$\begin{array}{lcll} [\![e]\!]_{\mathbf{EL}} & = & [[\![e]\!]]_{\mathbf{EL}} & (e \in \mathrm{MinAML}) \\ [\![e_2(e_1)]\!]_{\mathbf{EL}} & = & \mathtt{let}_{\mathbf{EL}}\ x \Leftarrow [\![e_1]\!]_{\mathbf{EL}}\ \mathtt{in}\ [\![e_2]\!]_{\mathbf{EL}}(x) & \end{array}$$

The translation of up- and down-lifters is given by

$$\begin{array}{lcl} [\![\mathtt{in_up}(e)]\!]_{\mathbf{EL}} & = & \mathtt{let}_{\mathbf{EL}}\ a \Leftarrow [\![e]\!]_{\mathbf{EL}}\ \mathtt{in} \\ & & \mathtt{let}_{\mathbf{EL}}\ () \Leftarrow down\ \mathtt{in}\ [a]_{\mathbf{EL}} \\ [\![\mathtt{up}(e)]\!]_{\mathbf{EL}} & = & \mathtt{let}_{\mathbf{EL}}\ () \Leftarrow up\ \mathtt{in}\ [\![\mathtt{in_up}(e)]\!]_{\mathbf{EL}} \end{array}$$

and dually for $\mathtt{in_down}(e)$ and $\mathtt{down}(e)$. It remains to define the constant aspect associated to $\mathtt{around}\ (l, f(x)) = e$,

$$\begin{array}{l} a_{f_i} : f_i \Rightarrow \mathtt{let}_{\mathbf{EL}}\ n \Leftarrow lookup\ \mathtt{in} \\ \qquad \mathtt{if}\ (l == n)\ \mathtt{then}\ (\lambda x.\ [\![t[f_{i+1}/\mathtt{proceed}]]\!]_{\mathbf{EL}}) \\ \qquad\qquad \mathtt{else}\ f_{i+1} \end{array}$$

Proposition 2 can be extended to MinAML$_{\mathrm{EL}}$ in the expected way.

PROPOSITION 3 (Translation of MinAML$_{\mathrm{EL}}$). *Given a program $d_s \cdot a_s \cdot e$ of MinAML$_{\mathrm{EL}}$ of type A, the configuration $\langle 0, d_s, a_s, e\rangle$ reduces to $\langle 0, d_s, a_s, v\rangle$ for some value v of type A if and only if*

$$\mathbf{weave}(\mathtt{run}_{\mathbf{EL},A}\ [\![e]\!]_{\mathbf{EL}}) = \{\mathtt{run}_{\mathbf{EL},A}\ [\![v]\!]_{\mathbf{EL}}\}.$$

7. The Execution-Level-with-Exceptions Monad

The usual management of exceptions in AOP (e.g. in AspectJ) is *flat* in the sense that an advice can typically catch an exception raised by the base computation and conversely. Starting from this observation, Ismael Figueroa and Éric Tanter have proposed to take levels into account when raising exceptions [4]. In this section, we present an extension MinAML$_{\mathrm{EL}}$ with exceptions sensitive to execution levels and show that this extension can be seen as the use of a particular exception 2-monad transformer.

7.1 MinAML$_{\mathrm{EL}}$ with exceptions

We extend MinAML$_{\mathrm{EL}}$ with three new constructors

$$e ::= \ldots \mid \mathtt{raise}\ ex \mid \mathtt{raise}_l\ ex \mid \mathtt{try}\ e\ \mathtt{with}\ e$$

where ex belongs to a special set $\mathbb{E}$ of exceptions and $\mathtt{raise}_l$ is not user-visible. The typing system is defined in a standard way and the small step reduction is extended as

$$\begin{array}{l} \langle l, d_s, a_s, E(\mathtt{raise}\ ex)\rangle \rightharpoonup \langle l, d_s, a_s, E(\mathtt{raise}_l\ ex)\rangle \\ \langle l, d_s, a_s, E(\mathtt{try}\ v\ \mathtt{with}\ e)\rangle \rightharpoonup \langle l, d_s, a_s, E(v)\rangle \\ \langle l, d_s, a_s, E(\mathtt{try}\ (\mathtt{raise}_l)\ ex\ \mathtt{with}\ e)\rangle \rightharpoonup \langle l, d_s, E(e(ex))\rangle \\ \langle l, d_s, a_s, E(\mathtt{try}\ (\mathtt{raise}_{l'}\ ex)\ \mathtt{with}\ e)\rangle \rightharpoonup \\ \qquad\qquad \langle l, d_s, E(\mathtt{raise}_{l'}\ ex)\rangle \qquad (l \neq l') \end{array}$$

7.2 The execution-level-with-exception monad transformer

If we apply the traditional exception 2-monad transformer (as defined in Equation (2)) to the execution-level 2-monad, we end up with a *flat* notion of exception that is oblivious to the current level. As in AspectJ, such a flat notion is not convenient because it typically allows advices to intercept exceptions raised by the base computation. To recover the semantics of MinAML$_{\mathrm{EL}}$ with exception, we need to define a 2-monad transformer

$$T_{\mathbf{EX}}\ A \quad = \quad T\ (A + (\mathbb{E} \times \mathbb{N}\mathrm{at}))$$

for any 2-monad T. This 2-monad transformer generates a notion of exceptions at the top of T that enables to attach an integer to an exception. In the case of the execution-level monad, we will use this integer to store the level at which the exception has been raised. $T_{\mathbf{EX}}$ is defined as (specific constructors of T are lifted straightforwardly)

$$\begin{array}{lcl} [\alpha]_{T_{\mathbf{EX}}} & = & [in_L \alpha]_T \\ \mathtt{let}_{T_{\mathbf{EX}}}\ x \Leftarrow \alpha\ \mathtt{in}\ \beta & = & \mathtt{let}_T\ u \Leftarrow \alpha\ \mathtt{in}\ \mathtt{case}\ u\ \mathtt{of} \\ & & \quad a \Rightarrow \beta \circ a \mid e \Rightarrow [in_R\ e]_T \\ \mathtt{run}_{T_{\mathbf{EX}},A}\ c & = & \mathtt{case}\ (\mathtt{run}_{T,A}\ c)\ \mathtt{of}\ a \Rightarrow a \end{array}$$

and we can define the two classical operations specific to the raiser and the handler:

$$\begin{array}{l} raise_A : (\mathbb{E} \times \mathbb{N}\mathrm{at}) \rightarrow T_{\mathbf{EX}}\ A \\ raise_A\ (e, n) \quad = \quad [in_R\ (e, n)]_T \end{array}$$

$$\begin{array}{l} handle_A : T_{\mathbf{EX}}\ A \rightarrow ((\mathbb{E} \times \mathbb{N}\mathrm{at}) \rightarrow T_{\mathbf{EX}}\ A) \rightarrow \mathbf{EX}\ A \\ handle_A\ c\ f \quad = \quad \mathtt{let}_T\ u \Leftarrow c\ \mathtt{in}\ \mathtt{case}\ u\ \mathtt{of} \\ \qquad\qquad\qquad a \Rightarrow [a]_{T_{\mathbf{EX}}} \mid e \Rightarrow f(e) \end{array}$$

7.3 Interpreting MinAML$_{\mathrm{EL}}$ with exceptions using the execution-level-with-exception monad

We now show how to interpret MinAML$_{\mathrm{EL}}$ with exceptions in the λ_2-calculus on the 2-monad $\mathbf{EL}_{\mathbf{EX}}$. The translation is the same as in Section 6, it just remains to lift the translation for terms of MinAML$_{\mathrm{EL}}$

$$\begin{array}{lcl} [\![e]\!]_{\mathbf{EX}} & = & \mathtt{let}_{\mathbf{EL}}\ x \Leftarrow [\![e_1]\!]_{\mathbf{EL}}\ \mathtt{in}\ [x]_{\mathbf{EX}} \\ [\![e_2(e_1)]\!]_{\mathbf{EX}} & = & \mathtt{let}_{\mathbf{EX}}\ x \Leftarrow [\![e_1]\!]_{\mathbf{EX}}\ \mathtt{in}\ [\![e_2]\!]_{\mathbf{EX}}(x) \end{array}$$

and translate the raiser and the handler

$$[\![\texttt{raise}\ e]\!]_{\mathbf{EX}} = \texttt{let}_{\mathbf{EL_{EX}}}\ n \Leftarrow \mathit{lookup}\ \texttt{in}\ \mathit{raise}_A(e, n)$$

$$[\![\texttt{try}\ c\ \texttt{with}\ f]\!]_{\mathbf{EL}} = \mathit{handle}_A\ c\ \tilde{f}$$

with $\tilde{f} = \lambda(e, n).\ \texttt{let}_{\mathbf{EL_{EX}}}\ l \Leftarrow \mathit{lookup}\ \texttt{in}$
$\texttt{if}\ (l == n)\ \texttt{then}\ [\![f]\!]_{\mathbf{EX}}(e)\ \texttt{else}\ \mathit{raise}_A\ (e, n)$

Proposition 3 can be extended to $\text{MinAML}_{\text{EL}}$ with exception in the expected way.

PROPOSITION 4 (Translation of $\text{MinAML}_{\text{EL}}$). *Given a program $d_s \cdot a_s \cdot e$ of $\text{MinAML}_{\text{EL}}$ with exception of type A, the configuration $\langle 0, d_s, a_s, e\rangle$ reduces to $\langle 0, d_s, a_s, v\rangle$ for some value v of type A if and only if*

$$\mathbf{weave}(\texttt{run}_{\mathbf{EL_{EX}},A}\ [\![e]\!]_{\mathbf{EX}}) = \{\texttt{run}_{\mathbf{EL_{EX}},A}\ [\![v]\!]_{\mathbf{EX}}\}.$$

8. Conclusion

The keystone of this paper is to approach AOP (and more generally type-preserving program transformation) from a category-theoretic prospective, in order to complement the software engineering approach. We believe that this approach could have substantial benefit at the level of conceptual understanding of what AOP actually is. More precisely, we identify (Cartesian closed) 2-categories as a suitable setting in which programs can be seen as 1-cells and aspects can be seen as 2-cells. To make this analogy precise, we present an internal language for Cartesian closed 2-categories called the λ_2-calculus—a 2-dimensional extension of the traditional λ-calculus. We formulate a notion of weaving inside the λ_2-calculus and demonstrate the applicability of our construction by translating a more realistic functional AOP language called MinAML into the λ_2-calculus. This translation enables to interpret a program of MinAML in a Cartesian closed 2-category and to define the weaving algorithm as the computation of a normal form in a rewriting system based on that 2-category. Finally, we introduce the notion of 2-monads for AOP, which are the direct extension of computational monads used in functional programming. We illustrate the conceptual power of 2-monads by defining an execution-level 2-monad that corresponds to Tanter's *execution levels* and a new exception monad transformer that give rise to *execution level with exception*. We believe that the development of 2-monads for AOP is the starting point of a new semantic understanding of the complex interaction between AOP mechanisms and traditional notions of computation.

Acknowledgments

The author wants to thank Tom Hirschowitz and Éric Tanter for valuable discussions.

References

[1] R. Blackwell, G. Kelly, and A. Power. Two-dimensional monad theory. *Journal of Pure and Applied Algebra*, 59(1):1–41, 1989.

[2] S. Bruggink. *Equivalence of Reductions in Higher-Order Rewriting*. PhD thesis, Utrecht University, 2008.

[3] D. Dantas and D. Walker. Harmless advice. In *8th*, volume 41, page 396, 2006.

[4] I. Figueroa and É. Tanter. A semantics for execution levels with exceptions. In *Proceedings of FOAL*, pages 7–11, Porto de Galinhas, Brazil, Mar. 2011. ACM Press.

[5] B. Hilken. Towards a proof theory of rewriting: the simply typed 2λ-calculus. *Theoretical Computer Science*, 170(1-2):407–444, 1996.

[6] T. Hirschowitz. Cartesian closed 2-categories and permutation equivalence in higher-order rewriting. submitted.

[7] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings of ECOOP*, pages 54–73. Springer-Verlag, 2003.

[8] G. Kelly and R. Street. Review of the elements of 2-categories. In *Category Seminar*, pages 75–103. Springer, 1974.

[9] M. Kelly. *Basic Concepts of Enriched Category Theory*, volume 64 of *Lecture Notes in Mathematics*. Cambridge University Press, 1982.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP*, volume 1241. Springer-Verlag, 1997.

[11] S. Kovalyov. Modeling Aspects by Category Theory. *Proceedings of FOAL*, page 63, 2010.

[12] J. Lambek. Cartesian closed categories and typed lambda-calculi. In *13th Spring School on Combinators and Functional Programming Languages*, page 175. Springer-Verlag, 1985.

[13] J. Lambek and P. Scott. *Introduction to higher order categorical logic*. Cambridge University Press, 1988.

[14] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *Proceedings of PEPM*, page 77. ACM, 2006.

[15] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[16] B. C. d. S. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: disciplined advice with explicit effects. In *Proceedings of AOSD*, pages 109–120, New York, NY, USA, 2010. ACM.

[17] R. Seely. Modelling computations: a 2-categorical framework. In *2nd*, pages 65–71, 1987.

[18] N. Tabareau. Aspect oriented programming: a language for 2-categories. In *Proceedings of FOAL*, pages 13–17, New York, NY, USA, 2011. ACM.

[19] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of AOSD*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

[20] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *8th*, volume 38, pages 127–139, 2003.

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