Aspect Oriented Programming: a language for 2-categories

Nicolas Tabareau

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Aspect-Oriented Programming: a language for 2-categories

Nicolas Tabareau

N° 7527

February 2011

Distributed Systems and Services

ISSN 0249-6399 ISRN INRIA/RR--7527--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
RENNES - BRETAGNE ATLANTIQUE

Aspect-Oriented Programming: a language for 2-categories

Nicolas Tabareau*

Theme : Distributed Systems and Services
Networks, Systems and Services, Distributed Computing
Équipes-Projets Ascola

Rapport de recherche n° 7527 — February 2011 — 27 pages

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, as it is the case for the related concept of code injection, in the guise of frameworks such as Swing and Google Guice. However, AOP lacks theoretical foundations to clarify this new idea. 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 propose to see an aspect as a morphism between morphisms—that is as a program that transforms the execution of a program. To make this connection precise, we develop an advised λ-calculus that provides an internal language for 2-categories and show how it can be used as a base for the definition of the weaving mechanism of a realistic functional AOP language, called MinAML. Finally, we advocate for a formalization of more complex AOP languages (eg. with references or exceptions) using the notion of enriched Lawvere theories.

Key-words: category theory, aspect-oriented programming, lambda-calculi

* nicolas.tabareau@inria.fr

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

La programmation par aspects: un langage pour les 2-catégories

Résumé : 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, as it is the case for the related concept of code injection, in the guise of frameworks such as Swing and Google Guice. However, AOP lacks theoretical foundations to clarify this new idea. 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 propose to see an aspect as a morphism between morphisms—that is as a program that transforms the execution of a program. To make this connection precise, we develop an advised λ-calculus that provides an internal language for 2-categories and show how it can be used as a base for the definition of the weaving mechanism of a realistic functional AOP language, called MinAML. Finally, we advocate for a formalization of more complex AOP languages (eg. with references or exceptions) using the notion of enriched Lawvere theories.

Mots-clés : category theory, aspect-oriented programming, lambda-calculi

1 Introduction

Aspect-Oriented Programming Aspect-Oriented Programming (AOP) [8] promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns. Indeed, AOP provides the facility to intercept the flow of control in an application and perform new computations. In this approach, computation at certain execution points, called *join points*, may be intercepted by a particular condition, called *pointcut*, and modified by a piece of code, called *advice*, which is triggered only when the runtime context at a join point meets the conditions specified by a pointcut. Using aspects, modularity and adaptability of software systems can be enhanced. In the AOP terminology, the algorithm that controls which aspects can be executed at each join point is called a *weaving* algorithm.

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 [6, 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. 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 complicated 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. Namely, 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, we propose to see an aspect as a 2-cell, that is as a morphism between morphisms. In the programming point of view, this means that an aspect can be seen as a program which transforms the execution of 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.

The definition of an internal language for cartesian closed 2-category will be the keystone of this paper, the basis to give a precise meaning to the possible interactions of a single aspect with the rest of the code.

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. In this paradigm, 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 [10, 11, 17]. Soon later, Robert Seely proposed a 2-categorical interpretation of the λ-calculus [18] where β-reduction constructs 2-cells between terms and their β-reduced version. 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 [4] where he developed a 2-dimensional λ-calculus that corresponds to the free 2-category with lax exponentials.

Recall that a 2-category $\mathcal{C}$ is basically a category in which the class $\mathcal{C}(A, B)$ of morphisms between any 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*.

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:

- What is the good notion of variables at a 2-dimensional level?
- What is the extended notion of β-reduction?
- How to describe vertical and horizontal composition of a 2-category in the language of typed λ-calculi?

Once this effort to develop an internal language for cartesian closed 2-categories has been done, 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.

Let us anticipate on the description of the λ_2-calculus to give an example straightaway. 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. This aspect can be seen as a piece of advice whose pointcut intercepts

the square root function and proceeds with the absolute value of the original argument of the function. 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 as 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 the aspect α above will generate all the possible combinations of this aspect with 2-dimensional primitives of the language and other constant aspects.

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 above, 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.

At the end of this article, we explain how this algebraic account of AOP can drive the definition of aspects in more powerful languages extended with references, exceptions or any programming primitives that are well-understood in category theory. This could be done by using a 2-categorical version of computational monads introduced by Eugenio Moggi [14]—and used for example in Haskell—to extend the λ_2-calculus smoothly. This 2-categorical extension can be seen as a particular case of the recent work of Martin Hyland, Gordon Plotkin and John Power on enriched Lawvere theories [5].

Note that the work of Kovalyov [9] 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.

About higher order algebras We have identified the higher order notion provided by 2-categories as a suitable setting where programs are interpreted by 1-cells and aspects (or more generally program transformations) are interpreted by 2-cells. At this stage, one could wonder whether it would have been more fruitful to work with other higher order notions in category theory like bicategories or double categories. Both are generalization of 2-categories, the former where the horizontal composition is not strictly associative, the latter where one can distinguish between horizontal and vertical morphisms. We believe that the refinement proposed by bicategories or double categories is not

necessary to interpret AOP programs as application is always associative and there is no meaningful distinction between horizontal and vertical programs of a given type.

On the other hand, one could wonder whether a simpler setting such as order-enriched categories would not be sufficient to interpret AOP language and define a weaving algorithm using rewriting theory. It is indeed the case if one consider only "pure" aspects that have no effect on the program. In this case, an aspect is just a program transformation and thus two aspects between the same programs will always be equal. But when aspects can have side effect such as logging on the screen, this is not true anymore and the full power of 2-categories is required (see for example Section 5.2).

Note that most of AOP systems consider aspects that can intercept other aspects. But then, one needs to define some stratification in the flow of execution to control the weaving of aspects. This mechanism can be explained with a notion of execution levels [19] that prevents aspects of lower levels to intercept aspects of higher levels of execution. It would be interesting to consider this notion of aspects intercepting aspects through execution levels in the light of *n-categories* or weak ω-categories. A relation between Martin-Löf Intensional Type Theory and weak ω-categories has recently been settled [2, 13] and could be a good basis for the study of execution levels.

Plan of the paper We introduce (§2) the language of cartesian closed 2-category and define the notion of polynomial 2-category. After that, we define (§3) the λ_2-calculus, an extension of the λ-calculus with 2-dimensional primitives that will be the basis for the interpretation of aspects. We then show (§4) that the λ_2-calculus is an internal language for cartesian closed 2-category and use (§5) this language to give a formal semantics of a functional AOP language called MinAML. Finally, we conclude (§6) sketching how this work can serve to study more powerful AOP languages (with references or exceptions) using advanced work on enriched Lawvere theory.

2 Cartesian closed 2-Categories in a nutshell

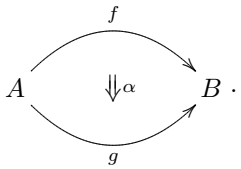
In this section, we briefly introduce cartesian closed 2-categories. We also present an extension of the notion of polynomials for 2-category, an extension that will be the base for the connection between the λ_2-calculus and cartesian closed 2-categories.

2.1 A glance at 2-categories

An abstract view on 2-categories is to see them as categories enriched over **Cat**, the cartesian category of categories (for more details about enriched category theory, see the monograph of Max Kelly [7]). Even if this point of view will become important at the end of this article, we prefer to give a more concrete definition. 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

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

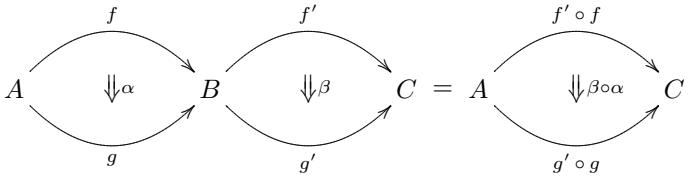
(or simply $\alpha : f \Rightarrow g$ when there is no confusion). A 2-cell $\alpha : (f \Rightarrow g) :: A \to B$ is generally diagrammatically represented as a 2-dimensional arrow between the 1-dimensional arrows f and g



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

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

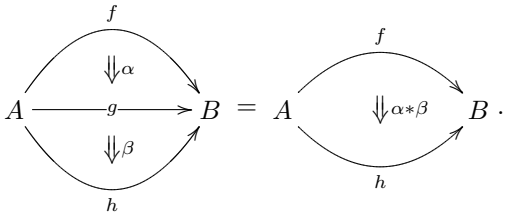
for the horizontal composite of two 2-cells $\alpha : f \Rightarrow g$ and $\beta : f' \Rightarrow g'$, represented diagrammatically as



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



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

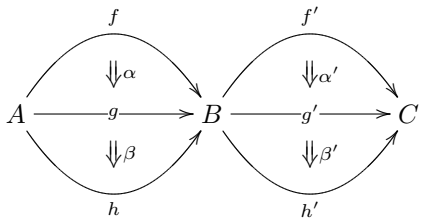
$$\mathbf{1}_f : f \Rightarrow f$$

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 diagram



are equal. From an AOP point of view, this property guarantees that applications of pieces of advice 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 pieces of advice at other points of the same program.

Just as a 2-category is a **Cat**-category, a *2-functor* consists of a functor enriched over **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.

A *2-natural transformation* is a **Cat**-natural transformation, i.e., a natural transformation between the underlying ordinary functors that also respects 2-cells.

2.2 Cartesian 2-categories

A 2-category $\mathcal{C}$ is said to be cartesian when the diagonal 2-functor $\Delta_n : \mathcal{C} \to \mathcal{C}^n$ has right 2-adjoints for all n. More concretely in a cartesian 2-category, every pair of objects A and B is equipped with two projection morphisms

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

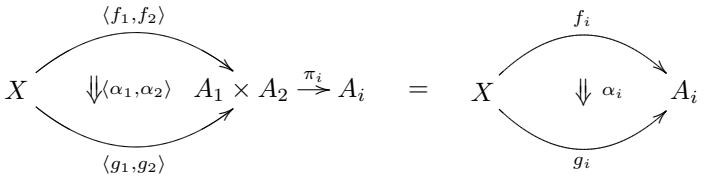
satisfying the following universal property: for every pair of 2-cells

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

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)



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.

Observe that the underlying category of a cartesian 2-category is also cartesian (instantiate every 2-cell in the universal property with the identity 2-cell) and that $\times$ can be extended to a 2-functor by

$$\alpha \times \beta = \langle \pi_1 \circ \alpha, \pi_2 \circ \beta \rangle.$$

From a λ_2-calculus point of view, the object 1 is represented by the type **Unit** and the unique 1-cell of that type is the constant term **skip**—the 1-cell $\mathbf{skip}_A$ is interpreted by $\lambda x : A.\ \mathbf{skip}$. Then, the unique aspect on $\mathbf{skip}_A$ is the identity aspect.

2.3 Closure in a cartesian 2-category

A cartesian 2-category $\mathcal{C}$ is *closed* when the 2-functor $(-) \times A$ has a right 2-adjoint $(-)^A$ for all objects A of $\mathcal{C}$.

More concretely, a cartesian closed 2-category 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.

2.4 Polynomial cartesian closed 2-category

To state the connection between cartesian closed 2-category and the λ_2-calculus, we need define the 2-category $\mathcal{C}[X]$ of polynomials over an indeterminate 2-cell

$$X : (x \Rightarrow x') :: 1 \to A$$

and indeterminate arrows $x, x' : 1 \to A$ of a 2-category $\mathcal{C}$. Indeed, a polynomial 2-cell

$$\alpha(X) : (t(x) \Rightarrow t'(x')) :: 1 \to B$$

will be seen as an aspect with one free 2-variable X, and the arrows $t(x)$ and $t'(x')$ will be seen as terms with respective free 1-variables x and x'.

The objects of $\mathcal{C}[X]$ are the same as those of $\mathcal{C}$, the morphisms are formal expressions built from the morphism forming operations of $\mathcal{C}$ and either from the symbol x or the symbol x'; and the 2-cells are formal expressions built from the symbol $X : x \Rightarrow x'$ and the 2-cell forming operations of $\mathcal{C}$. We note H_X the canonical embedding of $\mathcal{C}$ into $\mathcal{C}[X]$ which is the identity on objects, morphisms

and 2-cells. Just as it is the case for cartesian closed categories [10, 11], this 2-category of polynomials is cartesian closed as soon as $\mathcal{C}$ is cartesian closed, and furthermore $\mathcal{C}[X]$ satisfies the following universal property.

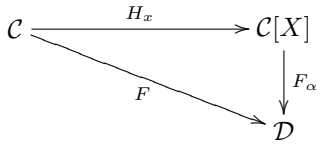
Proposition 1 *Given a cartesian closed 2-category $\mathcal{C}$ and an indeterminate 2-cell*

$$X : (x \Rightarrow x') :: 1 \to A$$

of $\mathcal{C}$, let $F : \mathcal{C} \to \mathcal{D}$ be a cartesian closed 2-functor into another cartesian closed 2-category $\mathcal{D}$ and $\alpha : a \Rightarrow b : 1 \to F(A)$ be a 2-cell of $\mathcal{D}$. Then there exists a unique cartesian closed 2-functor (ie. a 2-functor that preserves product and closure up to isomorphisms)

$$F_\alpha : \mathcal{C}[X] \to \mathcal{D}$$

with $F_\alpha(X) = \alpha$ and such that the diagram



commutes.

Applying this universal property that the identity 2-functor on $\mathcal{C}$ leads to a normal form theorem called *functional completeness*.

Corollary 1 (Functional completeness) *For every polynomial 2-cell*

$$\alpha(X) : f(x) \Rightarrow f'(x') : 1 \to B$$

in an indeterminate $X : (x \Rightarrow x') :: 1 \to A$, there exists a unique 2-cell

$$\beta : (g \Rightarrow g') :: 1 \to B^A$$

such that

$$\mathrm{eval} \circ (X \times \beta) = \alpha(X)$$

in $\mathcal{C}[X]$.

It is also possible to form a 2-category $\mathcal{C}[X_1, \ldots, X_n]$ of polynomials by adjoining a finite set of indeterminate 2-cells $X_i : (x_i \Rightarrow x_i') :: 1 \to A_i$ where the variables x_i or x_i' have to be distinct. Using product, one may show that

$$\mathcal{C}[X_1, \ldots, X_n] \equiv \mathcal{C}[Z]$$

for an indeterminate $Z : (z \Rightarrow z') :: 1 \to A_1 \times \cdots \times A_n$.

3 The λ_2-calculus

3.1 Types, terms and aspects

The grammar of the λ_2-calculus generated by a set of sort names $\mathcal{S}$ is presented in Figure 1. The sets of types and terms is closed under the traditional λ-calculus operations. For the second dimension, we construct a set of aspects

types	A	::=	$S \mid \mathbf{Unit} \mid A \times B \mid A \to B$
terms	t	::=	$f \mid x \mid \mathbf{skip} \mid \lambda x.\ t \mid t(t) \mid \langle t, t\rangle \mid \pi_i(t)$
aspects	α	::=	$a \mid X \mid \mathtt{asp}.\, t \mapsto t' \mid \alpha * \alpha \mid \alpha \circ \alpha \mid \langle \alpha, \alpha\rangle \mid \lambda X.\ \alpha$

Figure 1: The grammar of λ_2-calculus

which transform terms into other terms. An aspect α that transforms the term t of type A into the term t' of type A will be noted

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

Note that this can be thought of as a stratified instance of a dependent type theory, where 2-types can include terms. But the main difference between Martin-Löf type theory lies in the absence of a conversion rule—if $t = u$ and $\alpha : (t \Rightarrow t') :: A$, it is not in general the case that $\alpha : (u \Rightarrow t') :: A$.

For every type A, we suppose given a denumerable set of variables $x, \ldots$ that induces a denumerable set of 2-variables

$$X : (x \Rightarrow x) :: A$$

Note that we can have multiple variables of a given type but there exists only one 2-variable of a given 2-type. This simplifies the equational theory of the λ_2-calculus but 2-variables have to be fused when composed vertically (see Section 3.4). In [4], this issue is overcome differently by using de Bruijn indices. All the construction for pairing, abstraction and horizontal composition are extended to aspects and there is a notion of vertical composition $\alpha * \beta$ which means that the transformations performed by α and β are applied successively. We use the word "free" and "bound" in the usual sense for a 2-dimensional variable X in an aspect α. The main aspect forming operation

$$\mathtt{asp}.\, t \mapsto t' : (t \Rightarrow t') :: A$$

defines an aspect that transforms a closed term t of type A into another closed term t' of type A. It is crucial in the construction that the two terms are closed. Indeed, we do not accept aspect of the form

$$\mathtt{asp}.\, x \mapsto y : (x \Rightarrow y) :: A$$

where x and y are variables. Such an aspect would transform any term of type A into any term of type A.

We require that the class of aspects is closed under all aspect-forming operations except for the aspect $\mathtt{asp}.\, t \mapsto t'$ which must always exist only for identity aspects on constant terms, that is when $t' = t$ is a constant term.

Note that there may be additional types (S), constant terms (f) and constant aspects (a) in the language. As for the `asp.` constructor, constant aspects must be defined on closed typed terms.

3.2 Typing rules

The typing rules of the λ_2-calculus are given in Figure 2. Terms are typed in the presence of a context Γ that stipulates the type of variables while aspects

$$\textsc{Variable}\ \frac{}{\Gamma, x : A \vdash x : A} \qquad \textsc{Abstraction}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : A \to B}$$

$$\textsc{Application}\ \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t(u) : B}$$

$$\textsc{Bottom}\ \frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{Unit}} \qquad \textsc{Pairing}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B}$$

$$\textsc{Projection}\ \frac{}{\Gamma \vdash \pi_i^{A_1, A_2} : A_1 \times A_2 \to A_i}$$

$$\text{2-}\textsc{Variable}\ \frac{}{\Delta, X : (x \Rightarrow x) :: A \vdash X : (x \Rightarrow x) :: A}$$

$$\textsc{Aspect}\ \frac{\vdash t : A \qquad \vdash t' : A}{\Delta \vdash \mathtt{asp}.\, t \mapsto t' : (t \Rightarrow t') :: A}$$

$$\text{2-}\textsc{Pairing}\ \frac{\Delta \vdash \alpha : (t \Rightarrow t') :: A \qquad \Delta \vdash \beta : (u \Rightarrow u') :: B}{\Delta \vdash \langle \alpha, \beta \rangle : (\langle t, u \rangle \Rightarrow \langle t', u' \rangle) :: A \times B}$$

$$\text{2-}\textsc{Abstraction}\ \frac{\Delta, X : (x \Rightarrow x) :: A \vdash \alpha : (t \Rightarrow t') :: B}{\Delta \vdash \lambda X.\ \alpha : (\lambda x.\ t \Rightarrow \lambda x.\ t') :: A \to B}$$

$$\text{2-}\textsc{Application}\ \frac{\Delta \vdash \beta : (t \Rightarrow t') :: A \to B \qquad \Delta \vdash \alpha : (u \Rightarrow u') :: A}{\Delta \vdash \beta \circ \alpha : (t(u) \Rightarrow t'(u')) :: B}$$

$$\textsc{Vertical-composition}\ \frac{\Delta \vdash \alpha : (t_1 \Rightarrow t_2) :: A \qquad \Delta \vdash \beta : (t_2 \Rightarrow t_3) :: A}{\Delta \vdash \alpha * \beta : (t_1 \Rightarrow t_3) :: A}$$

Figure 2: Typing rules of the λ_2-calculus

are typed in the presence of a context Δ that stipulates the type of 2-variables. The rules for terms are the standard rules for the λ-calculus.

Rule 2-Abstraction and Rule 2-Pairing are the higher order version of closure and product in the calculus. Rule 2-Application and Rule Vertical-composition are the reminiscence of the corresponding 2-categorical compositions.

Observe that Rule Vertical-composition expects the same term t_2 at the common boundary of α and β. We would like to emphasize that we do not implicitly assume here that the equational theory is decidable. Indeed, the absence of conversion principle implies that only one 2-type can be assigned to an aspect in a given environment. So to decide if two aspects can be composed vertically, it is sufficient to check the syntactical equality of the term t_2 at the boundary.

Rule 2-VARIABLE introduces a 2-variable in the context Δ. Rule ASPECT checks that an aspect $\mathtt{asp}.\, t \mapsto t'$ transforms a closed typed term t into another closed term t' of the same type.

Additional constant terms and aspects of the language are given with their specific typing rules.

3.3 Equations between terms and aspects

The equality relation $\doteq$ between terms or between aspects of the same type and with the same free variables is defined as the least congruence relation derived from the equations below (we assume that all aspects used in the equation below are well-typed). Equality on terms is standard and is defined as in [10, 11, 17]. When two aspects

$$\alpha : (t \Rightarrow t') :: A \quad \text{and} \quad \beta : (u \Rightarrow u') :: A$$

are equal, we automatically know that

$$t \doteq u \quad \text{and} \quad t' \doteq u'.$$

The converse is not true in general. In the example above, we have no guarantee that α can be given the type $(u \Rightarrow u') :: A$. This is because we can not apply equality on terms independently from equality on aspects, that is we do not have a conversion principle..

1. Specific axioms for products
   (a) **[terminal object]**
   $$\alpha \doteq \mathtt{asp}.\, \mathbf{skip} \mapsto \mathbf{skip}$$
   for all $\alpha : (t \Rightarrow t') :: \mathbf{Unit}$
   (b) **[projections]**
   $$\pi_i \circ \langle \alpha_1, \alpha_2 \rangle \doteq \alpha_i$$
   (c) **[surjective pairing]**
   $$\langle \pi_1 \circ \alpha, \pi_2 \circ \alpha \rangle \doteq \alpha$$
   Here, π_i stands for the identity aspect $\mathtt{asp}.\, \pi_i \mapsto \pi_i$.
2. Specific axioms for lambda-calculus
   (a) **[α-conversion]**
   $$\lambda X.\ \alpha[X] \doteq \lambda Y.\ \alpha[Y]$$
   (b) **[β-rule]**
   $$(\lambda X.\ \alpha) \circ \beta \doteq \overline{\alpha}[\beta/X]$$
   (c) **[η-rule]**
   $$\alpha \doteq \lambda X.\ (\alpha \circ X)$$
   The notation $\alpha[\beta/X]$ denotes the aspect α where every occurrence of X has been replaced by β and $\overline{\alpha}$ denotes the canonical form of α (see Section 3.4)
3. Specific axioms on the vertical composition of aspects

(a) **[Identity for vertical composition]**

$$\begin{array}{rcll} X * X & \doteq & X & \\ (\mathtt{asp}.\, f \mapsto f) * \alpha & \doteq & \alpha & f \text{ constant} \\ \alpha * (\mathtt{asp}.\, f \mapsto f) & \doteq & \alpha & f \text{ constant} \end{array}$$

(b) **[Associativity for vertical composition]**

$$(\alpha * \beta) * \gamma \quad \doteq \quad \alpha * (\beta * \gamma)$$

(c) **[Distributivity over other operations]**

$$\begin{array}{rcl} (\beta \circ \alpha) * (\beta' \circ \alpha') & \doteq & (\beta * \beta') \circ (\alpha * \alpha') \\ (\lambda X.\ \alpha) * (\lambda X.\ \beta) & \doteq & \lambda X.\ \alpha * \beta \\ \langle \alpha, \beta \rangle * \langle \alpha', \beta' \rangle & \doteq & \langle \alpha * \alpha', \beta * \beta' \rangle \end{array}$$

Note that β-reduction is less usual as it involves the computation of a canonical form $\overline{\alpha}$ before the application of the substitution. The canonical form is here to make sure that 2-dimensional variables are fused (using the identity rule). Indeed, the vertical composite

$$(\lambda X.\ \alpha \circ X) * (\lambda X.\ \beta \circ X)$$

must be equal to the abstraction

$$\lambda X.\ (\alpha * \beta) \circ X.$$

(see the next section for more details).

The rules above constitute an extension of the rules for the traditional λ-calculus plus a management of 2-dimensional constructions. Note that specific axioms for horizontal composition (eg. associativity or identity) are not required as they can be deduced from the definition of substitution. There may be additional equations in the definition of the language.

3.4 Substitution and canonical form

As explained above, the definition of β-reduction requires to fuse the 2-dimensional variables. Indeed, because of vertical composition, the syntactical substitution of a 2-variable by an aspect does not preserve equality and is even not well defined on an arbitrary aspect. For example, consider the two equal aspects

$$\lambda X.\ X * X \quad \doteq \quad \lambda X.\ X$$

While the later aspect is just the identity, the substitution is not well-defined in general for the former aspect—it only makes sense when substituting by an aspect of the form $\alpha : (t \Rightarrow t) :: A$. This indicates that before the application of the substitution, we need to compute the *canonical form* of the aspect.

Given an aspect α, the aspect $\overline{\alpha}$ is defined as the normal form of the rewriting system obtained by orienting equations in (3.a) and (3.c) from left to right. Intuitively, this rewriting system puts the vertical composition inside the aspect syntax tree and eliminates redundant 2-variables $X * X$.

This rewriting system is confluent (there is no critical pair) and it is not difficult to show that it is terminating using a decreasing cost function on aspects $\kappa(\alpha)$ that emphasizes vertical compositions

- $\kappa(\mathtt{asp}.\, t \mapsto t') = \kappa(X) = 1$
- $\kappa(\alpha * \beta) = 2^{\kappa(\alpha)+\kappa(\beta)}$
- $\kappa(\beta \circ \alpha) = \kappa(\langle \alpha, \beta \rangle) = \kappa(\langle \alpha, \beta \rangle) = \kappa(\alpha) + \kappa(\beta)$
- $\kappa(\lambda X.\ \alpha) = \kappa(\alpha) + 1$

As an illustration, let us compute the canonical form of the previous example

$$\begin{array}{rcl} \overline{(\lambda X.\ \alpha \circ X) * (\lambda X.\ \beta \circ X)} & \doteq & \lambda X.\ \overline{(\alpha \circ X) * (\beta \circ X)} \\ & \doteq & \lambda X.\ \overline{(\alpha * \beta)} \circ \overline{(X * X)} \\ & \doteq & \lambda X.\ (\alpha * \beta) \circ X \end{array}$$

Then main interest of such a canonical form is that each 2-variable does not appear in a vertical composite.

Proposition 2 *In a well-typed aspect*

$$\overline{\alpha} : (t \Rightarrow t') :: A$$

with one free 2-variable X, every sub-aspect of the form

$$\alpha_1 * \alpha_2$$

does not contains X.

Proof When $\alpha_1 * \alpha_2$ is in canonical form, this means that the syntactical trees of α_1 and α_2 do no start with the same constructor. But we know that α_1 and α_2 agree on the term they have in common. As the λ_2-calculus does not support a conversion rule, this means that both aspects are closed and so do not contain X.

We can deduce from the previous proposition that substitution is well-defined on aspects in canonical form.

3.5 About strong normalization

The question of whether the λ_2-calculus calculus is strongly normalising amounts to define a strongly normalizing rewriting system by orienting each equality relation in a proper way.

As it is the case for the λ-calculus with equalities, there is no reason for a given λ_2-calculus to be strongly normalizing. This is because we have no guarantee that the added equations on extra constant terms or aspects can be turned into a strongly normalizing system.

But we can consider the question for the pure λ_2-calculus—that is the λ_2-calculus without additional constant terms, aspects or equalities. Let us orientate all equalities (except for α-conversion and η- expansion) of Section 3.3 from left to right. Note that the resulting rewriting system (that we note $\rightsquigarrow$) only satisfies a weak form of subject reduction :

Proposition 3 *If $\Delta \vdash \alpha : (t \Rightarrow t') :: A$ and $\alpha \rightsquigarrow \beta$, then their exists a sequence of reductions $\beta \rightsquigarrow^* \beta'$ such that $\Delta \vdash \beta' : (u \Rightarrow u') :: A$ and $t \doteq u$, $t' \doteq u'$.*

This weak version of subject reduction has two origins : (1) a local reduction can momentarily break the typing of vertical composition until a similar reduction is performed in the other branch of the composition; (2) the computation at the level of aspects (given by substitution) induces a computation at the level of terms and thus preserves the 2-types of aspects only up to equality between terms.

Church-Rosser Property The rewriting system without β-reduction is left-linear and confluent. The only critical pairs are given by harmless conflicts between associativity and other rules for vertical composition. Using Müller's Theorem [15], we can deduce that the rewriting system plus β-reduction is also confluent.

Strong normalization via reducibility candidate The technique of reducibility candidates as developed in [3] can adapted to the λ_2-calculus calculus. The notion of neutrality is extended accordingly: an aspect is called neutral if it is not of the form $\langle \alpha, \beta \rangle$, $\lambda X.\ \alpha$ or $\mathtt{asp}.\, t \mapsto t'$.

Proposition 4 *Equality in the pure λ_2-calculus is decidable.*

4 Cartesian closed 2-categories and the λ_2-calculus

The definition above leaves a lot of freedom. There are many λ_2-calculus. As it is the case for the traditional λ-calculus, one can think of *the* λ_2-calculus as the λ_2-calculus freely generated by a given set S of sort names, with no additional type, term, aspect or equation. But there are many more λ_2-calculi, as many as 2-categories.

4.1 Internal language of a Cartesian closed 2-category

Given a cartesian closed 2-category $\mathcal{C}$, we define the λ_2-calculus $\mathbf{L}(\mathcal{C})$ as follows:

1. types are objects of $\mathcal{C}$, **Unit** is the terminal object, $A \times B$ the cartesian product and $A \to B$ the exponentiation in $\mathcal{C}$,
2. terms with free variables
$$x_1 : A_1, \ldots, x_n : A_n$$
are morphisms of polynomial 2-cells in the cartesian closed 2-category $\mathcal{C}[X_1, \ldots, X_n]$, where
$$X_i : (x_i \Rightarrow x_i') :: 1 \to A_i$$
is a 2-dimensional indeterminate,
3. aspects with free variables
$$X_1 : (x_1 \Rightarrow x_1') :: A_1, \ldots, X_1 : (x_n \Rightarrow x_n') :: A_n$$
are polynomial 2-cells in $\mathcal{C}[X_1, \ldots, X_n]$, where

$$X_i : (x_i \Rightarrow x_i') :: 1 \to A_i$$

is a 2-dimensional indeterminate.

Lambda abstraction is given by functional completeness of Corollary 1. We define $\alpha \doteq \beta$ to hold if it holds as polynomial 2-cells.

It is not difficult to check that $\mathbf{L}(\mathcal{C})$ satisfies that equational theory of the λ_2-calculus.

4.2 Interpreting the λ_2-calculus in a Cartesian closed 2-category

Given a typed λ_2-calculus $\mathcal{L}$, we define the cartesian closed 2-category $\mathbf{C}(\mathcal{L})$ as follows:

1. objects of $\mathbf{C}(\mathcal{L})$ are the types of $\mathcal{L}$,
2. morphisms from A to B of $\mathbf{C}(\mathcal{L})$ are pairs
$$(x, t(x))$$
where $t(x) : B$ contains no other free variable than $x : A$. Two morphisms $(x, t(x))$ and $(y, t'(y))$ are equal when $\lambda x.\ t(x) \doteq \lambda y.\ t'(y)$ in $\mathcal{L}$,
3. 2-cells from $(x, t(x))$ to $(x', t'(x'))$ are aspects
$$(X, \alpha(X))$$
where
$$\alpha(X) : (t(x) \Rightarrow t'(x')) :: B.$$
contains no other free variable than $X : (x \Rightarrow x') :: A$. Two 2-cells $(X, \alpha(X))$ and $(X, \beta(X))$ are equal when
$$\alpha(X) \doteq \beta(X)$$
in $\mathcal{L}$.

We can define the interpretation of identities, composition, abstraction and pairing in the expected way.

Identity 2-cells We define identity 2-cells by

$$\mathbf{1}_{(x,t(x))} = (X, \mathbb{I}(t(x)))$$

where the 2-cell $\mathbb{I}(t)$ is constructed by induction on t :

- $\mathbb{I}(f) = \mathtt{asp}.\, f \mapsto f$ for any constant term f
- $\mathbb{I}(y) = Y : (y \Rightarrow y) :: A$
- $\mathbb{I}(\lambda y.\ u) = \lambda Y.\ \mathbb{I}(u)$
- $\mathbb{I}(u'(u)) = \mathbb{I}(u') \circ \mathbb{I}(u)$
- $\mathbb{I}(\langle u, u' \rangle) = \langle \mathbb{I}(u), \mathbb{I}(u') \rangle$

types	A	::=	$S \mid \mathbf{Unit} \mid A \times B \mid A \to B$
terms	t	::=	$x \mid f \mid \mathbf{skip} \mid \lambda x.\ t \mid t(t) \mid \langle t, t\rangle \mid \pi_i(t) \mid \texttt{proceed}(t)$
aspects	α	::=	$[\,] \mid [\texttt{around}\ f(x) = t] \cdot \alpha$
declarations	ds	::=	$[\,] \mid [\texttt{let}\ f = t] \cdot ds$
programs	p	::=	$ds \cdot \alpha \cdot t$

Figure 3: The grammar of the MinAML

Compositions The vertical composition is simply given by the vertical composition of aspects

$$(X, \alpha(X)) * (Y, \beta(Y)) = (X, \alpha(X) * \beta(X))$$

Horizontal composition is obtained by first computing the canonical form and then substituting

$$(Y, \beta(Y)) \circ (X, \alpha(X)) = (X, \overline{\beta}(\alpha(X)))$$

Pairing

$$\langle (X, \alpha(X)), (X, \beta(X))\rangle = (X, \langle \alpha(X), \beta(X)\rangle)$$

Abstraction The functor $\Lambda_{A,B,C}$ is given by

$$\Lambda_{A,B,C}(Z, \alpha(Z)) = (X, \lambda Y.\ \alpha(\langle X, Y\rangle))$$

Evaluation. The morphism $\mathrm{eval}_{A,B}$ from $A \times (A \to B)$ to B is given by

$$\mathrm{eval}_{A,B} = (X, \pi_2(X) \circ \pi_1(X))$$

We let the reader check that for any λ_2-calculus $\mathcal{L}$, $\mathbf{C}(\mathcal{L})$ is a cartesian closed 2-category. We can now state the property that makes the λ_2-calculus an internal language for cartesian closed 2-categories.

Proposition 5 *For any cartesian closed 2-category $\mathcal{C}$ and any λ_2-calculus $\mathcal{L}$,*

$$\mathbf{C}(\mathbf{L}(\mathcal{C})) \cong \mathcal{C} \quad \textit{and} \quad \mathbf{L}(\mathbf{C}(\mathcal{L})) \cong \mathcal{L}.$$

The first isomorphism presupposes the notion of morphisms between cartesian closed 2-categories, which is given by cartesian closed 2-functors. The second isomorphism presupposes the notion of morphisms between λ_2-calculi. This can be defined as for traditional λ-calculus with the notion of *translations*, ie. maps Φ that transport types to types, terms to terms (including mapping the ith variable of type A to the ith variable of type $\Phi(A)$), aspects to aspects (including mapping the ith variable of type $(x \Rightarrow x') :: A$ to the ith variable of type $(\Phi(x) \Rightarrow \Phi(x')) :: \Phi(A)$) and preserve all the type-, term- or aspect-forming operations and equations on the nose.

Note that it is possible to extend this isomorphism at a 2-categorical level by defining a notion corresponding to natural transformations between translations.

4.3 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 $\mathcal{L}$ where x is of type A, we will consider all the possible interactions of the (constant) aspects defined in $\mathcal{L}$ with $t(x)$ by considering the category $\mathbf{C}(\mathcal{L})(A, B)$. 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. Thus, the result of the weaving algorithm is given by the normal form of the image of $t(x)$ in that category.

More precisely, the set of woven terms is defined as

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

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 $\text{Woven}(t(x))$ 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. For example, when multiple pieces of advices can be applied at the same join point in AspectJ, precedence orders are (arbitrarily) defined, based on the order in which definitions of pieces of advice syntactically appear in the code. More algebraic approaches have been proposed (see eg. [12]).

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 Eric Tanter on execution levels is precisely a way to introduce a hierarchy in the application of aspects and thus to avoid basic circular definition [19]. Other lines of work have proposed to restricted the power of pieces of advice (for example using a typing system [1]) in order to guarantee that the execution of the program is not critically perturbed.

5 MinAML

This section gives the semantics of a concrete AOP language called MinAML by a translation to the λ_2-calculus. More precisely, given a program p, we will construct a λ_2-calculus λ_p, whose underlying 2-category defines a rewriting system from which we can deduce the definition of a weaving algorithm.

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 as 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. Indeed, unlike AspectJ which allows programmers to refer to any method

that appears anywhere in their program, even private methods of classes, the functions referred to by pieces of advice in the work of David Walker and colleagues must be in scope. 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

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

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

The second extension is the introduction of aspects with the constructor

$$\texttt{around}\, f(x) = t$$

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

When declaring pieces of advice, 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`. Let us slightly rewrite the history and say that the keyword `proceed` has been introduced to tackle the case where a pointcut can intercept more than one function. In that situation, the programmer may want to run the intercepted function without knowing its name, which can be using the keyword `proceed`. This is not possible in MinAML but we have kept this keyword as it also emphasizes that the function $\widetilde{f}$ executed by the piece of advice through `proceed` is not strictly identical to the intercepted function f. More precisely, the function $\widetilde{f}$ behaves as f but can no longer be intercepted by the aspect. This avoids trivial circular definitions in the application of aspects.

In the same way 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 3. A program p is constituted of a list of declarations ds, a list of aspects α and a term t. 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 Extension to effectful aspects

So far, an aspect of MinAML is always pure. As sketched in the introduction, in order to exploit the full power of the λ_2-calculus, we need to add effectful

$$\textsc{Variable}\ \frac{}{\Gamma, x : A; \Delta \vdash x : A} \qquad \textsc{Name}\ \frac{}{\Gamma; \Delta, f : A \vdash f : A}$$

$$\textsc{Bottom}\ \frac{}{\Gamma; \Delta \vdash \mathbf{skip} : \mathbf{Unit}}$$

$$\textsc{Abstraction}\ \frac{\Gamma, x : A; \Delta \vdash t : B}{\Gamma; \Delta \vdash \lambda x.t : A \to B} \qquad \textsc{Application}\ \frac{\Gamma; \Delta \vdash t : A \to B \qquad \Gamma; \Delta \vdash u : A}{\Gamma; \Delta \vdash t(u) : B}$$

$$\textsc{Pairing}\ \frac{\Gamma; \Delta \vdash t : A \qquad \Gamma; \Delta \vdash t' : B}{\Gamma; \Delta \vdash \langle t, t' \rangle : A \times B}$$

$$\textsc{Binding}\ \frac{; \Delta \vdash t : A \qquad ; \Delta, f : A \vdash p : B}{; \Delta \vdash \mathtt{let}\ f = t; p : B}$$

$$\textsc{Around}\ \frac{x : A; \Delta, f : A \Rightarrow A' \vdash t[f/\mathtt{proceed}] : A' \qquad ; \Delta, f : A \to A' \vdash p : B}{; \Delta, f : A \to A' \vdash \mathtt{around}\ f(x) = t; p : B}$$

Figure 4: Typing rules of MinAML

aspects. We choose here to simply add two constants in the language: a logging function

$$log : \mathbb{N}at \to \mathbb{N}at$$

whose purpose is to be intercepted by the logging aspect

$$Log\_asp : (log \Rightarrow \lambda x.\ x) :: \mathbb{N}at \to \mathbb{N}at.$$

that transforms $log(n)$ into n and prints n to the screen.

5.3 A simple example

Let us now express in this language the example developed in the introduction—of an aspect that ensures that all calls to the *sqrt* function are performed on non-negative values. To make the example richer, we also define an aspect that applies the function *log* (before the function *abs*) to the argument of *sqrt* so that the argument will then be printed out by the aspect *Log_asp*. The following program of MinAML (where we use some usual primitives on integers) defines such aspects and run *sqrt* on the negative value -4.

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

5.4 Typing

The typing rules for λ-terms presented in Figure 4 are standard. 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 ds, then a set of aspects α 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 also 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 `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, an aspect in MinAML is not be able to intercept a term with free variables in the same way as an aspect in λ_2-calculus cannot be defined between open terms.

5.5 A translation into the pure λ_2-calculus

We now present the translation of a typed program

$$p = ds \cdot \alpha \cdot t$$

into the λ_2-calculus. More precisely, we will define a λ_2-calculus $\mathcal{L}_p$ based on declarations present in ds and aspects present in α. The construction of $\mathcal{L}_p$ goes in two steps:

(1) we produce a list of aspects $[\![\alpha]\!]$ and a mapping γ from declaration names in ds to integers. As 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. This transformation can be written with an Ocaml-like function using the `fold_l` function.

```
let trans_asp(a,b) = match (a,b) with
   | ((γ,A),(around f(x) = t)) -> (γ[f ↦ γ(f) + 1],
         A++[asp. f_γ(f) ↦ λx. t[f_γ(f)+1/proceed]])
in (γ,[[α]]) = fold_l(trans_asp,(let γ x = 1,[]),α)
```

where $\gamma[f \mapsto \gamma(f) + 1]$ stands for the map γ whose value on f has been incremented by 1. That is, the ith aspect that intercept f, let say

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

will thus be translated into the aspect

$$\texttt{asp}.\ f_i \mapsto \lambda x.\ t[f_{i+1}/\texttt{proceed}]$$

that intercept f_i and proceeds with f_{i+1}. In this way, we construct a sequence of declaration names

$$f_1(= f), f_2, \ldots, f_{\gamma(f)}$$

that drives the list of aspects that can intercept the application of the function f.

(2) we define a list of aspects $[\![ds]\!]$ by translating each declaration

$$\texttt{let}\, f = t$$

into the aspect

$$\texttt{asp}.\, f_{\gamma(f)} \mapsto t.$$

This transformation can be written with an Ocaml-like function using the `map` function.

```
let trans_eq(d) = match d with
    | let f = t -> [asp. f_γ(f) ↦ t]
in [[ds]] = map trans_eq ds
```

The λ_2-calculus $\mathcal{L}_p$ is generated by the constant terms

$$\{f_i \mid f \in ds \text{ and } 1 \leq i \leq \gamma(f)\}$$

—that represent all declaration names introduced in the translation of aspects— and by the list of aspects $[\![\alpha]\!]$ and $[\![ds]\!]$. Let us present the effect of the translation on the program $\mathbb{P}$ above. The λ_2-calculus $\mathcal{L}_{\mathbb{P}}$ is generated by the constant terms

$$sqrt_1, sqrt_2, sqrt_3, abs_1$$

and by the four aspects

$$\begin{array}{ll} a_1: & \texttt{asp}.\, sqrt_1 \mapsto \lambda x.\ sqrt_2(\log(x)) \\ a_2: & \texttt{asp}.\, sqrt_2 \mapsto \lambda x.\ sqrt_3(abs_1(x)) \\ a_3: & \texttt{asp}.\, sqrt_3 \mapsto \lambda x.\ \sqrt{x} \\ a_4: & \texttt{asp}.\, abs_1 \mapsto \lambda x.\ |x| \end{array}$$

5.6 Weaving in MinAML

Once the λ_2-calculus $\mathcal{L}_p$ has been generated, the weaving algorithm is defined as in Section 4.3. Namely, we computed the normal form (if it exists) in the corresponding category—observe that, as usual, the ordering of pieces of advice guarantees that there is at most one normal form.

This gives us the interleaved program that we can then execute using the strong normalization of the pure λ_2-calculus.

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):

$$\begin{array}{lll} sqrt_1(-4) & \xrightarrow{a_1 \circ \mathbb{I}(-4)} & sqrt_2(\log(-4)) \\ & \xrightarrow{\mathbb{I}(sqrt_2) \circ Log\_asp \circ \mathbb{I}(-4)} & sqrt_2((\lambda x.\ x)(-4)) \\ & & = sqrt_2(-4) \\ & \xrightarrow{a_2 \circ \mathbb{I}(-4)} & sqrt_3(abs_1(-4)) \\ & \xrightarrow{a_3 \circ \mathbb{I}(abs_1(-4))} & \sqrt{abs_1(-4)} \\ & \xrightarrow{\mathbb{I}(\sqrt{-}) \circ a_4 \circ \mathbb{I}(-4)} & \sqrt{|-4|} = 2 \end{array}$$

Observe the particular kind of *parametricity* describes in the introduction. Indeed, a single definition of the aspect Log_asp generates all the possible combinations of that aspect with 2-dimensional primitives of the language, and in particular the aspect

$$\mathbb{I}(sqrt_2) \circ Log\_asp \circ \mathbb{I}(-4)$$

used in the computation of the weaving on $sqrt_1(-4)$.

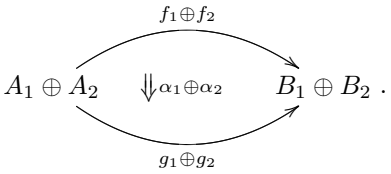
6 Extensions to more complex aspects

6.1 Adding conditional pointcuts

Conditionals are given in category theory by finite coproducts

$$A \oplus B.$$

A way to add conditionals in MinAML, and thus to be able to define conditional pointcuts, is thus to work in a cartesian closed 2-category equipped with finite coproducts. An aspect, whose pointcut is conditional on the values of the arguments of the intercepted function, will thus be described by a 2-cell



In that situation, the weaving mechanism will just be defined in the same way, and the choice will be resolved when we interpret morphisms that come from terms that reduce to values in the original language. In that case, there is no ambiguity in the branch that is taken during the execution. This static weaving can be seen as a partial evaluation that will be dynamically completed.

So the weaving of aspects is still static but contains a lot of choices. For example, the term

$$A_1 \oplus A_2 \xrightarrow{f_1 \oplus f_2} B_1 \oplus B_2$$

in the example above will be woven into

$$A_1 \oplus A_2 \xrightarrow{g_1 \oplus g_2} B_1 \oplus B_2$$

whereas the same term precomposed with the first injection (that resolves the choice between f_1 and f_2)

$$A_1 \xrightarrow{inj_1} A_1 \oplus A_2 \xrightarrow{f_1 \oplus f_2} B_1 \oplus B_2$$

will directly be woven into

$$A_1 \xrightarrow{g_1} B_1 \cdot$$

More generally, this is the way we understand *dynamic* weaving of aspects in the language of 2-categories: a *static* weaving that contains all the possibilities which are *dynamically* resolved at computation.

6.2 Extensions using enriched Lawvere theories

We conclude this article by sketching how to extend MinAML with references, exceptions or other notions of computation. The idea is to reuse the categorical interpretation of those notions in a 2-categorical setting in order to stick to our previous construction.

References or exceptions (among many other notions) are traditionally interpreted using the Kleisli construction over the suitable strong monad, called in that case a computational monad [14]. For example, the state monad T for references is defined on objects by

$$TA = S \to (S \times A).$$

The type S represents the memory in which references are registered. Then a term of type $A \to B$ in a λ-calculus with references is interpreted as a morphism of type $A \to B$ in the Kleisli category, that is a morphism of type

$$A \to TB \quad \cong \quad (S \times A) \to (S \times B).$$

Such a morphism explicitly manages the memory state associated to references. The key to extend MinAML with references is to define a 2-monad that extends the definition of the state monad on 2-categories. Then we can use the 2-dimensional version of the Kleisli construction to define the categorical interpretation of MinAML with references. We can do the same thing for exceptions and the associated exception monad.

Using the correspondence between strong monads and Lawvere theories, and the work of Martin Hyland, Gordon Plotkin and John Power on enriched Lawvere theories [16, 5], this indicates that we have to work with **Cat**-enriched Lawvere theories. Note that their motivation to extend the work of Eugenio Moggi to an enriched setting is the remark that in denotational semantics, the base category is not **Set** but rather the category ω-**Cpo** of ω-cpos and continuous functions. It appears that in our work on AOP, the base category is not **Set** but rather **Cat**.

This approach provides an algebraic way to formalize powerful AOP languages. We believe that **Cat**-enriched Lawvere theories are required for a clean definition of the complex mechanisms that show up in AOP.

7 Conclusion

The idea of the 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 (or more generally program transformations) can be seen as 2-cells. To make this analogy precise, we develop a language for 2-categories called the λ_2-calculus, as a 2-dimensional extension of the traditional λ-calculus, and show that it is an internal language for cartesian closed 2-categories. We also show that the pure λ_2-calculus is strongly normalizing.

We then 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 normal form in a rewriting system based on that 2-category. The well-foundedness of the weaving algorithm is thus given by the existence of a normal form in the corresponding rewriting system. We briefly sketch how to extend our categorical setting to interpret conditional pointcuts using finite coproducts.

At the end of the article, we discuss an algebraic way to extend the λ_2-calculus with various notions of computation using enriched Lawvere theory. This nice formulation of algebraic theories in an enriched setting enables to transpose the notion of computational monads of Eugenio Moggi at the level of 2-categories. We believe that this model-theoretic account of computation is necessary to understand the complex interaction between AOP mechanisms and traditional notions of computation.

References

[1] D. Dantas and D. Walker. Harmless advice. In *8th symposium on Principle of Programming Languages*, volume 41, page 396, 2006.

[2] R. Garner and B. Van den Ber. Types are weak omega-groupoids. Accepted for publication in the Proceedings of the London Mathematical Society.

[3] J. Girard, Y. Lafont, and P. Taylor. Proofs and types, volume 7 of Cambridge tracts in theoretical computer science, 1989.

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

[5] M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[6] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings of European Conference on Object Oriented Programming*, pages 54–73. Springer-Verlag, 2003.

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

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

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

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

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

[12] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, page 77. ACM, 2006.

[13] P. Lumsdaine. Weak ω-categories from intensional type theory. *Typed Lambda Calculi and Applications*, pages 172–187, 2009.

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

[15] F. Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.

[16] J. Power. Enriched Lawvere theories. *Theory and Application of Categories*, 6(7):83–93, 1999.

[17] P. Scott. Some aspects of categories in computer science. *Handbook of algebra*, 2:3–77, 2000.

[18] R. Seely. Modelling computations: a 2-categorical framework. In *2nd Logic in Computer Science*, pages 65–71, 1987.

[19] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, 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 International Conference Fonctional Programming*, 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.

Centre de recherche INRIA Rennes – Bretagne Atlantique
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399