Aspect Oriented Programming: a language for 2-categories

Nicolas Tabareau

Aspect Oriented Programming: a language for 2-categories

Nicolas Tabareau
INRIA
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. However, AOP lacks theoretical foundations to clarify this new idea. This paper proposes to put a bridge between AOP (and more generally program transformation) 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 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.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: semantics; F.3.2 [*Semantics of Programming Languages*]: Algebraic approaches to semantics

General Terms Languages, Theory, design

Contents

1. Introduction

Aspect Oriented Programming. Aspect-Oriented Programming (AOP) [4] 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 recent research efforts [2, 12, 13] have made significant progress on understanding some of the semantic issues involved, the algebraic explanation of aspect features has never rich the beauty and simplicity of the connection between the λ-calculus and cartesian closed categories. We believe that this 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 effects 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 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 sequence of normal forms 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 aspects 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 [5, 6, 9]. Soon later, Robert Seely proposed a 2-categorical interpretation of the λ-calculus [10] where β-reduction constructs 2-cells between terms and their β-reduced version. This perspectives is in line with the thought that 2-cells can be seen as rewriting rules between morphisms (or terms).

Recall that a 2-category $\mathcal{C}$ is basically a category in which the class $\mathcal{C}(A, B)$ of morphisms between the 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. But of course, not every 2-category can be viewed as a typed λ-calculus whose 2-cells corresponds to β-reduction.

In this paper, we propose to extend the typed λ-calculus with 2-dimensional primitives that enable to describe any 2-cell of a cartesian closed 2-categories. Those additional primitives construct a kind of 2-dimensional terms that we will (abusively) 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 rises interesting and difficult issues:

- What is the good notion of variable 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 ?
- What are the restrictions on how 2-dimensional variables – or more generally aspects with free variables – can be (horizontally or vertically) composed?

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 you have defined an aspect $\alpha : f \Rightarrow g$ that transforms the function f into the function g. The effect of α on the program

$$p = \lambda x.\, \lambda y.\, h(f(\langle x, y\rangle))$$

will be described by the aspect

$$\beta = \lambda X.\, \lambda Y.\, (\text{asp. } h \mapsto h) \circ \alpha \circ \langle X, Y\rangle$$

that transforms the program p into the program

$$p' = \lambda x'.\, \lambda y'.\, h(g(\langle x', y'\rangle)).$$

The aspect β is automatically generated from the aspect α and constructors of the λ_2-calculus.

Of course, existing AOP languages do not look like the λ_2-calculus but we show how programs of a simple functional language with aspects, introduced by David Walker and colleagues in [12] 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 sequence of normal forms 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 primitive that are well-understood in category theory. This could be done indeed by using a 2-categorical version of computational monads introduced by Eugenio Moggi [7] to extend smoothly the λ_2-calculus. 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 [1].

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 ver-

tical 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.

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 [11] that prevents aspects of lower levels to intercept aspects of higher levels of execution. It would be interested to consider this notion of aspect intercepting aspect through execution levels in the light of *n-categories*.

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 realistic 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

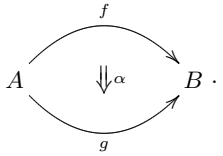
In this section, we briefly introduce cartesian closed 2-categories. We also present an extension of the notion of polynomials for 2-category, an extension that will be the base for the connection between the λ_2-calculus and cartesian closed 2-categories.

2.1 A glance at 2-categories

An abstract view on 2-categories is to see them as categories enriched over $\mathbf{Cat}$, the cartesian category of categories (for more details about enriched category theory, see the monograph of Max Kelly [3]). 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

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

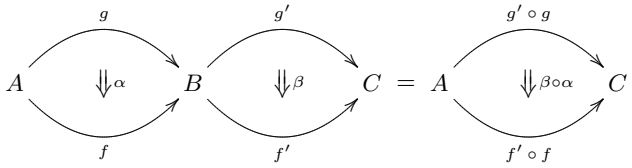
(or simply $\alpha : f \Rightarrow g$ when there is no confusion). A 2-cell $\alpha : (f \Rightarrow g) :: A \to B$ is generally diagrammatically represented as a 2-dimensional arrow between the 1-dimensional arrows f and g



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

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

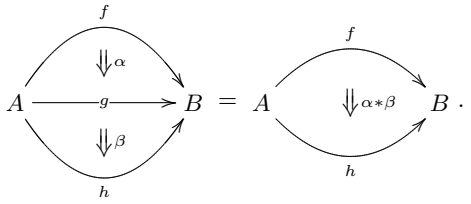
for the horizontal composite of two 2-cells $\alpha : f \Rightarrow g$ and $\beta : f' \Rightarrow g'$, represented diagrammatically as



and we write

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

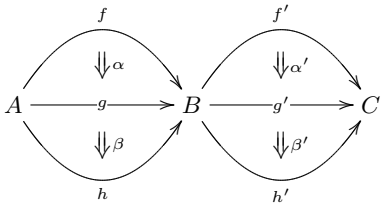
for the vertical composite of two 2-cells $\alpha : f \Rightarrow g$ and $\beta : g \Rightarrow h$, represented diagrammatically as



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 $\mathbf{1}_{\mathrm{id}_A}$). 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. 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 $\mathbf{1}_f$.

Just as a 2-category is a $\mathbf{Cat}$-category, a *2-functor* consists in 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.

A *2-natural transformation* is a $\mathbf{Cat}$-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

$$\pi_1 : A \times B \to A \qquad \pi_2 : A \times B \to B.$$

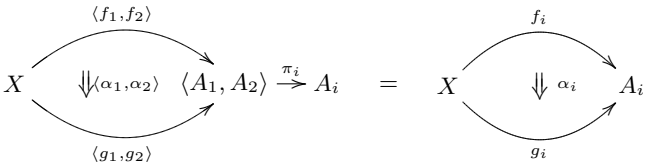
satisfying the following universal property: for every pair of 2-cells

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

there exists a unique 2-cells

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

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 $\perp_A : A \to 1$ and $\mathbf{1}_{\perp_A} : \perp_A \Rightarrow \perp_A$ is the unique 2-cell of that type.

Notice 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

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

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

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

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

2.4 Polynomial cartesian closed 2-category

Let us now 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 $\mathcal{C}$. 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 [5, 6], this 2-category of polynomials is cartesian closed as soon as $\mathcal{C}$ is cartesian closed and $\mathcal{C}[X]$ satisfies the following universal property.

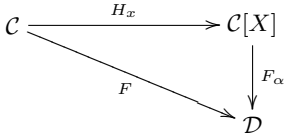
PROPOSITION 1. *Given a cartesian closed 2-category $\mathcal{C}$ and an indeterminate 2-cell*

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

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

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

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 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 it is crucial that the two terms t and t' have the same type.

We suppose given a denumerable set of variables $x, y, \ldots$ together with a denumerable set of 2-variables $X, Y, \ldots$ and assume that every 2-variable X has a fixed type

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

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

$$
\begin{array}{llll}
\text{types} & A & ::= & S \mid 1 \mid A \times B \mid A \to B \\
\text{terms} & t & ::= & x \mid \bot \mid \lambda x_A.\, t \mid t(t) \mid \langle t, t \rangle \mid \pi_i(t) \\
\text{aspects} & \alpha & ::= & X \mid \text{asp. } t \mapsto t' \mid \alpha * \alpha \mid \alpha \circ \alpha \mid \langle \alpha, \alpha \rangle \mid \lambda X_A^{x,x'}.\, \alpha
\end{array}
$$

Figure 1. The grammar of λ_2-calculus

"bound" in the usual sense for a 2-dimensional variable X in an aspect α. The main aspect forming operation

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

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

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

where x and y are variables. Such an aspect would transform (after β-reduction) any term of type A into any term of type A! We would like to stress already that this *will not be the case* for the definition of aspects in the realistic language of Section 5. Indeed, the definition of open pieces of advice will be encoded in the λ_2-calculus by the introduction of closed aspects (with some constants) and some equality relating those constants with other terms of the language. Anticipating slightly the rest of the definition, we would like also to point out that the term asp. $t \mapsto t'$ could be encoded by two constant terms c and c', a constant aspect $\alpha : c \Rightarrow c'$ and two equations relating constant terms to closed terms: $c \doteq t$ and $c' \doteq t'$. We have chosen to introduce this construction for two reasons:

1. The aspect
$$\text{asp. } t \mapsto t : (t \Rightarrow t) :: A$$
stands for the vertical identity aspect, which must exist for every closed term t, and thus the encoding above would require an introduction of infinitely many constant terms just to deal with identities.
2. This aspect forming operation is the heart of the translation of more realistic AOP language into the λ_2-calculus.

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

Note that there may be additional types, constant terms and constant aspects in the language.

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 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.

Rules 2-ABSTRACTION and 2-PAIRING are the 2-dimensional version of closure and product in the calculus. Rules HORIZONTAL-COMPOSITION and VERTICAL-COMPOSITION are the reminiscence of the corresponding 2-categorical compositions. Rule 2-VARIABLE introduces a 2-variable in the context Δ. Rule ASPECT checks that when a aspect asp. $t \mapsto t'$ transforms a closed term t into t', this two terms have the same type. It is important to insist on the fact that t and t' must be closed terms (see the paragraph above).

Note that the 1-dimensional typing rules APPLICATION, ABSTRACTION and PAIRING could be deduced from the 2-dimensional counterparts (using the identity aspect).

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

3.3 Equations between terms

The equality relation $\doteq$ between terms or between aspects of the same type and with the same free variables is defined as an equivalence relation. The rules defining this relation mimic equations that holds in a cartesian closed 2-category (a detailed definition can be found in Appendix A). For instance, we have a 2-dimensional version of the two specific axioms for $lambda$-calculus

1. **[β-rule]**
$$(\lambda X_A^{x,x'}.\, \alpha) \circ \beta \doteq \alpha[\beta/X]$$
2. **[η-rule]**
$$\lambda X_A^{x,x'}.\, (\alpha \circ X) \doteq \alpha$$

The notation $\alpha[\beta/X]$ denotes the aspect α where every occurrence of X has been replaced by β. In particular, when two aspects

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

are equals, this implicitly means that

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

Thus, looking at the 1-dimensional terms involved in the β-reduction above, one can deduce the traditional rule

$$(\lambda x.\, t)(u) \doteq t[u/x].$$

Note that the definition of the compatibility between vertical composition and λ-abstraction involves some extra work. This compatibility, representing the functoriality of the abstraction, requires to merge 2-dimensional variables in a proper way (see Appendix A for more details). We do not address here the problem of *empty types* using equations parametrized by the set of free variables [6, 9]. This simplification prevents the use of presheaf 2-categories for interpreting our language.

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

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

$$\textsc{Bottom}\ \frac{}{\Gamma \vdash \bot : 1} \qquad \textsc{Pairing}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B} \qquad \textsc{Projection}\ \frac{}{\Gamma \vdash \pi_i^{A_1, A_2} : A_1 \times A_2 \to A_i}$$

$$\textsc{2-Variable}\ \frac{}{\Delta, X : (x \Rightarrow x') :: A \vdash X : (x \Rightarrow x') :: A} \qquad \textsc{Aspect}\ \frac{\vdash t : A \qquad \vdash t' : A}{\Delta \vdash \text{asp. } t \mapsto t' : (t \Rightarrow t') :: A}$$

$$\textsc{2-Pairing}\ \frac{\Delta \vdash \alpha : (t \Rightarrow t') :: A \qquad \Delta \vdash \beta : (u \Rightarrow u') :: B}{\Delta \vdash \langle \alpha, \beta \rangle : (\langle t, u \rangle \Rightarrow \langle t', u' \rangle) :: A \times B} \qquad \textsc{2-Abstraction}\ \frac{\Delta, X : (x \Rightarrow x') :: A \vdash \alpha : (t \Rightarrow t') :: B}{\Delta \vdash \lambda X_A^{x, x'}.\, \alpha : (\lambda x_A.\, t \Rightarrow \lambda x'_A.\, t') :: A \to B}$$

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

Figure 2. Typing rules of the λ_2-calculus

or equation. But there are much 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-calculus $\mathbf{L}(\mathcal{C})$ as follows:

1. types are the objects of $\mathcal{C}$, 1 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 the top morphisms of 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,
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.

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

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 $(x, t'(x))$ are equal when $t(x) \doteq t'(x)$ 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 (details can be found in Appendix B).

PROPOSITION 2. *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 3. *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 a morphism between cartesian closed 2-categories, which is given by a cartesian closed 2-functor. 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.

types	A	::=	$S \mid 1 \mid A \times B \mid A \to B$
terms	t	::=	$x \mid f \mid \bot \mid \lambda x_A.\, t \mid t(t) \mid \langle t, t\rangle \mid \pi_i(t) \mid \texttt{proceed}(x)$
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 MinAML

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

5. MinAML

This section gives the semantics of a concrete AOP language called MinAML by a translation to a 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 [12] 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 introduced 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.

We discuss at the end of this section how to extend our categorical setting to interpret conditional pointcuts.

5.1 Syntax

MinAML is an extension of the λ-calculus with products in two steps. The first extension is usual as it is the introduction of 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 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`. Historically, 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 done with the keyword `proceed`. This is not possible in MinAML but we have kept this keyword as it emphasizes the fact that the function $\widetilde{f}$ executed by the 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. 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 A simple example

Let us anticipate slightly the end of this article and presents an example that will enable to illustrate the constructions defined below. Consider the following program $\mathbb{P}$ of MinAML (where we use some usual primitives on integers)

$$\begin{aligned}\mathbb{P} \;=\; & [\texttt{let}\ f = \lambda x.\, 2 * x;\\ & \ \texttt{let}\ g = \lambda x.\, x + 3;\\ & \ \texttt{let}\ h = \lambda x.\, g(f(x))] \cdot\\ & [\texttt{around}\ h(x) = \texttt{proceed}(x+1);\\ & \ \texttt{around}\ f(x) = \texttt{proceed}(\texttt{proceed}(x));\\ & \ \texttt{around}\ f(x) = g(x)] \cdot\\ & [h(5)]\end{aligned}$$

This program defines three functions f, g and h, three aspects and runs the term $h(5)$. The first aspect simply increments the argument of h before it is executed, the second aspect executes f twice and the third aspect substitutes the execution of f by the execution of g. Note that the order in which aspects are applied is important as third aspect would prevent the second aspect to be applied as it entirely replace the computation of f by a new computation, without a call to `proceed`. Note also that in this last aspect, the piece of advice $g(x)$ has one free variable g which must be an already defined declaration name. Given the typing system above, one can assign the type `Int` to $\mathbb{P}$. Furthermore, we will see that the execution of $\mathbb{P}$ gives the value 15.

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

$$\textsc{Bottom}\ \frac{}{\Gamma; \Delta \vdash \bot : 1} \qquad \textsc{Pairing}\ \frac{\Gamma; \Delta \vdash t : A \qquad \Gamma; \Delta \vdash t' : B}{\Gamma; \Delta \vdash \langle t, t' \rangle : A \times B} \qquad \textsc{Binding}\ \frac{; \Delta \vdash t : A \qquad \Gamma, \Delta, f : A \vdash p : B}{\Gamma; \Delta \vdash \texttt{let}\ f = t; p : B}$$

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

Figure 4. Typing rules of MinAML

5.3 Typing

The typing rules for λ-terms 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 declaration of a name 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 variable 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. Then the program $\texttt{around}\ f(x) = t; p$ has 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.

5.4 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 aspects present in α and a list of equalities $\mathcal{E}_p$ based on declarations present in ds. 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. Then, the function γ maps each declaration name f to the integer introduced by the last aspect in the list that intercepts f. The ith aspect that intercept f, let say

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

   will thus be translated into the aspect

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

   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. In what follows, we identify f_1 with f for every declaration name f.

2. we define a λ_2-calculus $\mathcal{L}_p$ generated by the constant terms

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

   that represent every declaration name introduced in the translation of aspects, and generated by the list of aspects $[\![\alpha]\!]$. Note that at this stage, we have not treated declarations in ds.

The construction of $\mathcal{E}_p$ is more direct as we simply transform each declaration

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

into the equality

$$f_{\gamma(f)} \doteq t_\sigma.$$

where t_σ is the term t after the application of the substitution σ defined on declaration names of ds as

$$\sigma(g_1) \ = \ g_{\gamma(f)} \quad \text{forall } g \in ds.$$

Before stated those definitions more formally, let us present the effect of the translation on the program $\mathbb{P}$ above. The λ_2-calculus calculus $\mathcal{L}_{\mathbb{P}}$ is generated by the constant terms

$$h_1, h_2, g_1, f_1, f_2, f_3$$

and by the three aspects

$$\begin{array}{l} \text{asp. } h_1 \mapsto \lambda x.\ h_2(x+1) \\ \text{asp. } f_1 \mapsto \lambda x.\ f_2(f_2(x)) \\ \text{asp. } f_2 \mapsto \lambda x.\ g_1(x). \end{array}$$

The list $\mathcal{E}_{\mathbb{P}}$ is given by the three equalities

$$\begin{array}{l} [h_2 \doteq \lambda x.\ g_1(f_3(x)); \\ \ g_1 \doteq \lambda x.\ x + 3; \\ \ f_3 \doteq \lambda x.\ 2 * x] \end{array}$$

Let us now give a formal definition of $\mathcal{L}_p$ and $\mathcal{E}_p$. We find convenient to describe the definition of $([\![\alpha]\!], \gamma)$ with an Ocaml-like function.

```
let rec trans(α,γ,A) =
  match α with
    | [] -> (γ,A)
    | (around f(x) = t) :: α') ->
        trans(α',γ[f ↦ γ(f) + 1],
         A++[asp.  f_{γ(f)} ↦ λx.  t[f_{γ(f)+1}/proceed]])

in ([[α]],γ) = trans(α,let gamma x = 1,[])
```

where $\gamma[f \mapsto \gamma(f) + 1]$ stands for the map γ whose value on f has been incremented by 1. Consider now the translation of the list of declarations ds into a list of equalities $\mathcal{E}_p$. As said before, we introduce one equation by declaration as follows

```
let rec Eq(ds) =
  match ds with
    | [] -> []
    | (let f = t) :: ds') ->
        Eq(ds') ++ [f_{γ(f)} ≐  t_σ]
in E_p = Eq(ds)
```

where t_σ is the same as in the informal explanation above. We introduce only one equation for $f_{\gamma(f)}$ because other constants of the sequence will not be present after the weaving of aspects. Remark that the order in which equations appear is inverted with the order of declarations.

5.5 Weaving algorithm as computation of normal forms

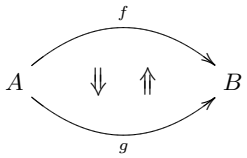
We now give the semantics of a program of MinAML

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

by defining a weaving algorithm in terms of a sequence of normal forms in a 2-category. The idea is to see the term t' corresponding to the term t after the weaving of aspects α as the normal form of t in the *underlying 2-category of* p, namely $\mathbf{C}(\mathcal{L}_p)$. Let us already notice that this normal form does not always exists. Aspects weaving may not terminate as it is possible to define loops with aspects. For instance, a program that contains the two aspects

$$\texttt{around}\ f(x) = g(x) \quad \text{and} \quad \texttt{around}\ g(x) = f(x)$$

will loop as soon as f or g is called. At a categorical level, this corresponds to the existence of a 2-cell from f to g and another from g to f



So the semantics of a term t will be defined using its normal form when it exists and else will be undefined.

Actually, the situation is even more complicated. If the term t contains a declaration name h that is itself defined as the composition of two declaration names f and g, one must weave aspects intercepting h first and then, if h is still present, substitute h by $g \circ f$ and weave aspects intercepting f and g. To emulate this process, we will rather compute a sequence of normal forms in the underlying 2-category of p, one after each application of an equality defined in $\mathcal{E}_p$.

Consider the 2-category $\mathbf{C}(\mathcal{L}_p)$, what we call above the underlying 2-category of p. To give a notion of normal form, we extract from $\mathbf{C}(\mathcal{L}_p)$ the rewriting system $\mathbf{R}(\mathcal{L}_p)$ whose objects are morphisms of $\mathbf{C}(\mathcal{L}_p)$ and whose reductions correspond to 2-cells that are different from identity 2-cells. The following proposition guarantees that there is at most one normal form of a morphism of $\mathbf{C}(\mathcal{L}_p)$ in $\mathbf{R}(\mathcal{L}_p)$.

PROPOSITION 4. *$\mathbf{R}(\mathcal{L}_p)$ is confluent.*

The confluence of $\mathbf{R}(\mathcal{L}_p)$ follows from the structure of a 2-category and the fact that there is no critical pair. Indeed, by construction, only one aspect can be applied to a given constant term f_i, and an aspect transforms only one constant term at a time.

As we have said, the termination of $\mathbf{R}(\mathcal{L}_p)$ depends on the structure of p. So in the sequel, when we say "the normal form of" we implicitly means "when it exists". So we take the liberty to talk about the normal form $t_{\mathbf{R}(\mathcal{L}_p)}$ of a term t in the rewriting system $\mathbf{R}(\mathcal{L}_p)$.

We now describe the weaving algorithm `weave(`t`,Eq)` in an Ocaml-like language. The idea is to compute the normal form of t in $\mathbf{R}(\mathcal{L}_p)$, substitute the last declaration name with its corresponding term and start again until all declaration names have been substituted

```
let rec weave(t,Eq) =
  match Eq with
    | [] -> t_{R(L_p)}
    | e :: Eq' -> subst e in t ;
                  weave(t_{R(L_p)},Eq')
in weave(t,E_p).
```

5.6 Weaving on a simple example

Let us explain 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) :

$$\begin{array}{lll}
h_1(5) & \longrightarrow & h_2(5+1) \quad (1)\\
& \longrightarrow & g_1(f_1(5+1)) \quad (2)\\
& \longrightarrow & g_1(f_2(f_2(5+1))) \quad (3)\\
& \longrightarrow & g_1(g_1(g_1(5+1)) \quad (4)\\
& \longrightarrow & (((5+1)+3)+3)+3 \quad (5)\\
& \longrightarrow & (((5+1)+3)+3)+3 \quad (6)\\
& \longrightarrow & 15 \quad (7)
\end{array}$$

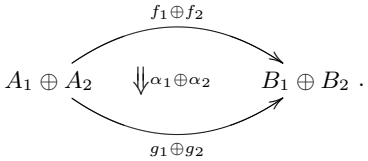
(1) computation of the normal form of $h(5)$ in $\mathbf{R}(\mathcal{L}_{\mathbb{P}})$ (this just means weaving with the unique aspect on h). (2) substitution of h_2 by $g_1 \circ f_1$. (3) weaving with the first aspect on f. (4) weaving with the second aspect on f, the result is a normal form in $\mathbf{R}(\mathcal{L}_{\mathbb{P}})$. (5) substitution of g by the function $\lambda x.\ x+3$ (plus some β-reduction). The new term is already in normal form. (6) substitution of f_3 by the function $\lambda x.\ 2 * x$, actually f_3 does not appear anymore in the term. (7) some final extra β-reduction.

5.7 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. So the aspects weaving 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

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

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. 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 [7]. 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 [1, 8], this indicates that we have to work with $\mathbf{Cat}$-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 $\mathbf{Set}$ but rather the category ω-$\mathbf{Cpo}$ of ω-cpos and continuous functions. It appears that in our work on AOP, the base category is not $\mathbf{Set}$ but rather $\mathbf{Cat}$.

This approach provides an algebraic way to formalize powerful AOP languages. We believe that working with $\mathbf{Cat}$-enriched Lawvere theories is the price to pay for a clean definition of the complex mechanisms appearing in AOP.

7. Conclusion

The idea of the paper is to approach AOP (and more generally 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 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 a sequence of computation of normal forms in a rewriting system based on that 2-category. The well-foundedness of a program (that is the convergence 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] M. Hyland, G. Plotkin, and J. Power. Combining effects: sum and tensor. *Theoretical Computer Science*, 357(1):70–99, 2006.

[2] 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.

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

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

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

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

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

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

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

[10] R. Seely. Modelling computations: a 2-categorical framework. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 65–71, 1987.

[11] 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.

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

[13] 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 (TOPLAS)*, 26(5): 890–910, 2004.

A. Equations between terms

The equality relation $\doteq$ between terms or between aspects of the same type and with the same free variables of the λ_2-calculus is defined as an equivalence relation closed under the rules below. When two aspects

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

are equals, this implicitly means that

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

Note that we only present equalities on aspects, equalities on terms can be deduced from the remark above. We also assume that all the aspects used in the rules below are well-typed:

1. $\doteq$ must be a congruence with respect to all term-forming operations. For example, one requires that

$$\frac{\alpha(X) \doteq \beta(X)}{\lambda X_A^{x,x'}.\, \alpha(X) \doteq \lambda Y_A^{y,y'}.\, \beta(Y)} \qquad \frac{\alpha_1 \doteq \beta_1 \qquad \alpha_2 \doteq \beta_2}{\langle \alpha_1, \alpha_2 \rangle \doteq \langle \beta_1, \beta_2 \rangle}$$

2. Specific axioms for products

   (a) **[terminal object]**

   $$\alpha \doteq \text{asp. } \bot \mapsto \bot \qquad \text{for all } \alpha : (t \Rightarrow t') :: 1$$

   In particular, this means that $t \doteq \bot$ for all $t : 1$

   (b) **[projections]**

   $$\langle \alpha_1, \alpha_2 \rangle \circ \pi_i \doteq \alpha_i$$

   (c) **[surjective pairing]**

   $$\langle \alpha \circ \pi_1, \alpha \circ \pi_2 \rangle \doteq \alpha$$

3. Specific axioms for lambda-calculus

   (a) **[β-rule]**

   $$(\lambda X_A^{x,x'}.\, \alpha) \circ \beta \doteq \alpha[\beta/X]$$

   (b) **[η-rule]**

   $$\lambda X_A^{x,x'}.\, (\alpha \circ X) \doteq \alpha$$

   The notation $\alpha[\beta/X]$ denotes the aspect α where every occurrence of X has been replaced by β

4. Specific axioms on the vertical composition of aspects

   (a) **[Id. and assoc. for vertical composition]**

   $$\begin{array}{rcl} (\text{asp. } t \mapsto t) * \alpha & \doteq & \alpha \\ \alpha * (\text{asp. } t' \mapsto t') & \doteq & \alpha \\ (\alpha * \beta) * \gamma & \doteq & \alpha * (\beta * \gamma) \end{array}$$

   (b) **[Commutation with other operations]**

   $$\begin{array}{rcl} (\beta \circ \alpha) * (\beta' \circ \alpha') & \doteq & (\beta * \beta') \circ (\alpha * \alpha') \\ (\lambda X_A^{x,x'}.\, \alpha) * (\lambda Y_A^{y,y'}.\, \beta) & \doteq & \lambda Z_A^{z,z'}.\, \overline{\alpha * \beta}^{X,Y,Z} \\ \langle \alpha, \beta \rangle * \langle \alpha', \beta' \rangle & \doteq & \langle \alpha * \alpha', \beta * \beta' \rangle \end{array}$$

   The second equation uses the normal form $\overline{\alpha}^{X,Y,Z}$ described in the next paragraph.

The rules above constitute an extension of the rules for the traditional λ-calculus plus a management of 2-dimensional constructions. Their may be additional equations in the definition of the language.

We do not address here the problem of *empty types* using an equations parametrized by the set of free variables [6, 9]. This simplification prevents the use of presheaf 2-categories for interpreting our language.

Functoriality of the abstraction. It turns that some work has to be done to define the compatibility of the vertical composition with the abstraction. Indeed, the vertical composite

$$(\lambda X_A^{x,x'}.\, \alpha \circ X) * (\lambda Y_A^{z,z'}.\, \beta \circ Y).$$

must be equal to the abstraction

$$\lambda Z_A^{z,z'}.\, (\alpha * \beta) \circ Z.$$

This requires to fuse the 2-dimensional variables.

Given an aspect α, the aspect $\overline{\alpha}^{X,Y,Z}$ is defined as the normal form of the rewriting system obtained by orienting equations in (4.a) and (4.b) from left to right plus the rewrite rule

$$X * Y \longrightarrow Z.$$

Let us unfold the definition for the example above

$$\begin{array}{rl} & (\lambda X_A^{x,x'}.\, \alpha \circ X) * (\lambda Y_A^{y,y'}.\, \beta \circ Y) \\ \doteq & \lambda Z_A^{z,z'}.\, \overline{(\alpha \circ X) * (\beta \circ Y)}^{X,Y,Z} \\ \doteq & \lambda Z_A^{z,z'}.\, \overline{(\alpha * \beta)}^{X,Y,Z} \circ \overline{(X * Y)}^{X,Y,Z} \\ \doteq & \lambda Z_A^{z,z'}.\, (\alpha * \beta) \circ Z \end{array}$$

B. Internal language

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 $(x, t'(x))$ are equal when $t(x) \doteq t'(x)$ 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

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

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

- $\mathbb{I}(f, \sigma) = \text{asp. } f \mapsto f$ for any closed term f
- $\mathbb{I}(y, \sigma) = \sigma(y)$ for any variable y
- $\mathbb{I}(\lambda y_A.\, u, \sigma) = \lambda Y_A^{y,y'}.\, \mathbb{I}(u, \sigma \cup (y \mapsto Y))$
- $\mathbb{I}(u'(u), \sigma) = \mathbb{I}(u', \sigma) \circ \mathbb{I}(u, \sigma)$
- $\mathbb{I}(\langle u, u' \rangle, \sigma) = \langle \mathbb{I}(u, \sigma), \mathbb{I}(u', \sigma) \rangle$

The substitution σ in the definition above relates variables to their corresponding 2-variables.

Compositions. Horizontal composition is simply given by substitution

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

The vertical composition is more involved as it uses the notion of normal form of an aspect defined above

$$(X, \alpha(X)) * (Y, \beta(Y)) = (Z, \overline{\alpha(X) * \beta(Y)}^{X,Y,Z}(Z)))$$

Pairing.

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

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

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

PROPOSITION 5. *For any λ_2-calculus $\mathcal{L}$, $\mathbf{C}(\mathcal{L})$ is a cartesian closed 2-category.*

Proof. We have to show that all equations valid in a cartesian closed 2-category are valid in $\mathbf{C}(\mathcal{L})$. We will only consider equations for identity 2-cells on the left and the interchange law as they concentrate all the technicality of the proof. Those two facts rely on the use of the normal form $\overline{\alpha}^{X,Y,Z}$ in the definition of the vertical composition.

Let us first show that identity 2-cells are indeed identities. Given a 2-cell $(Y, \alpha(Y))$ where

$$\alpha(Y) : (t(y) \Rightarrow t'(y')) :: B,$$

we have to show that

$$(X, \mathbb{I}(t(x), (x \mapsto X))) * (Y, \alpha(Y)) = (Z, \alpha(Z)).$$

Expanding the definition, this is equivalent to

$$\overline{\mathbb{I}(t(x), (x \mapsto X)) * \alpha(Y)}^{X,Y,Z} = \alpha(Z).$$

We now proceed by induction on $t(y)$, but before that, we need to generalize the induction hypothesis on an arbitrary substitution σ

$$\overline{\mathbb{I}(t(\vec{x}), \sigma) * \alpha(\vec{Y})}^{\vec{X},\vec{Y},\vec{Z}} = \alpha(\vec{Z}).$$

where $\vec{X} = (X_1, \ldots, X_n)$.

- **closed term.** $\mathbb{I}(f, \sigma) = \text{asp. } f \mapsto f$. We use the first equation of (A.4.a) relating identity and vertical composition.
- **variable.** $\mathbb{I}(y, \sigma) = \sigma(y)$. We use the fact that $X * Y = Z$ in the definition of a normal form.
- **abstraction.** $\mathbb{I}(\lambda w_A.\, u, \sigma) = \lambda W_A^{w,w'}.\, \mathbb{I}(u, \sigma \cup (w \mapsto W))$. We use the second equation of (A.4.b) and the generalized induction hypothesis.
- **application** $\mathbb{I}(u'(u), \sigma) = \mathbb{I}(u', \sigma) \circ \mathbb{I}(u, \sigma)$. We use the first equation of (A.4.b) and the induction hypothesis.
- **pairing** $\mathbb{I}(\langle u, u' \rangle, \sigma) = \langle \mathbb{I}(u, \sigma), \mathbb{I}(u', \sigma) \rangle$. We use the third equation of (A.4.b) and the induction hypothesis.

Let us now show that the interchange law is satisfied in the 2-category $\mathbf{C}(\mathcal{L})$. Consider four 2-cells $(X, \alpha(X))$, $(Y, \beta(Y))$, $(X', \alpha(X'))$ and $(Y', \beta'(Y'))$, we have to show that

$$\begin{array}{rl} & \Big((X', \alpha(X')) * (Y', \beta'(Y'))\Big) \circ \Big((X, \alpha(X)) * (Y, \beta(Y))\Big) \\ = & \Big((X', \alpha'(X')) \circ (X, \alpha(X))\Big) * \Big((Y', \beta'(Y')) \circ (Y, \beta(Y))\Big) \end{array}$$

Expanding the definition, this amounts to show that

$$\begin{array}{rl} & \overline{\alpha'(X') * \beta'(Y')}^{X',Y',Z'} \circ \overline{\alpha(X) * \beta(Y)}^{X,Y,Z} \\ = & \overline{(\alpha' \circ \alpha(X)) * (\beta' \circ \beta(Y))}^{X,Y,Z} \end{array}$$

By definition of the normal form (using the rewrite rule deduced from the first equation of (A.4.b)), we have

$$\begin{array}{rl} & \overline{(\alpha' \circ \alpha(X)) * (\beta' \circ \beta(Y))}^{X,Y,Z} \\ = & \overline{\alpha'(X) * \beta'(Y)}^{X,Y,Z} \circ \overline{\alpha(X) * \beta(Y)}^{X,Y,Z} \end{array}$$

which, up-to a change of variable, is just what we want.