Aspect Oriented Programming: a language for 2-categories

Nicolas Tabareau
INRIA
École des Mines de Nantes, France

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.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Algebraic approaches to semantics

General Terms

Languages, Theory, design

Keywords

AOP, 2-category

1. INTRODUCTION

Aspect-Oriented Programming. Aspect-Oriented Programming (AOP) [5] 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 [4, 15, 16] 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, an interaction 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 (2-CCC) 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 lan-

FOAL'11, March 21, 2011, Pernambuco, Brazil.

guage of Cartesian closed categories. Recall that Cartesian closed categories are categories equipped with a Cartesian product and such that each hom-set can naturally be seen as an object of the category (e.g. in the category **Set** of sets and functions, functions between two sets A and B form again a set). 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 [7, 8, 11]. Soon later, Robert Seely proposed a 2-categorical interpretation of the λ-calculus [12] 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 [2] 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 2-CCC. 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 [15] 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 [10]—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 [3].

Note that the work of Kovalyov [6] 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.

A full version of this article [13] is available.

2. THE λ_2-CALCULUS

2.1 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

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

Figure 1: The grammar of λ_2-calculus

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

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

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

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

All the constructions 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 aspects 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.

2.2 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.

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.

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. Equations between terms and between aspects, together with properties of the calculus are given in the long version [13].

3. 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 as stated by the following proposition (details can be found in [13].

Proposition 1. *The λ_2-calculus is the internal language of Cartesian closed 2-categories.*

Using the correspondence between the λ_2-calculus and 2-CCCs, 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 $\mathbf{C}(\mathcal{L})(A, B)$ of the 2-CCC $\mathbf{C}(\mathcal{L})$ associated to $\mathcal{L}$. 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.

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

Of course, such a normal form has no reason to be unique or even to exist.

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 advice 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. [9]).

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 [14]. Other lines of work have proposed to restrict 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.

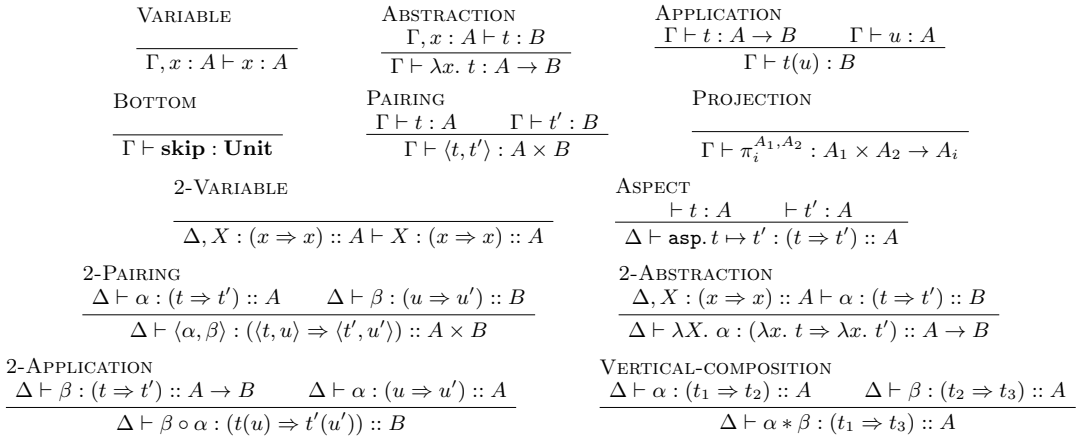

Figure 2: Typing rules of the λ_2-calculus

4. 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 [15] 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.

4.1 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

$$\mathtt{let}\ f = t.$$

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

The second extension is the introduction of aspects with the constructor

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

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`.

In the same way, when multiple aspects intercept the same function f, one must define an 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.

Typing rules of MinAML are given in the long version of this article [13].

4.2 Extension to effectful aspects

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

4.3 A simple example

Let us now express in this language the example developed in the introduction. 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.

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

4.4 Weaving on a simple example

MinAML can be translated into the λ_2-calculus (see [13] for details) . Thus, the weaving algorithm can be deduced from the weaving algorithm described in Section 3.

In this short version, we will just explain its behavior on the simple program $\mathbb{P}$. The computation can be described by the following sequence of reductions (where some extra β-reduction has been performed to make the reading easier):

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

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

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

$$id(sqrt_2) \circ Log\_asp \circ id(-4)$$

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

5. CONCLUSION

The idea of the paper is to approach AOP (and more generally type-preserving program transformation) from a category-theoretic perspective, 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 makes it possible to interpret a program of MinAML in a Cartesian closed 2-category and to define the weaving algorithm as the computation of a normal form in a rewriting system based on that 2-category. The well-foundedness of the weaving algorithm is thus given by the existence of a normal form in the corresponding rewriting system.

In the long version of this 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.

6. REFERENCES

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

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

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

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

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

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

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

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

[9] 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.

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

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

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

[13] N. Tabareau. Aspect oriented programming: a language for 2-categories (long version). Technical Report RR-7527, INRIA, 2011. http://hal.inria.fr/inria-00470400.

[14] É. Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th conference on AOSD*, pages 37–48, Rennes and Saint Malo, France, Mar. 2010. ACM Press.

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

[16] 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.