The Maquette Monad

Carlos Agon, Karim Haddad, Gonzalo Romero-García

The Maquette Monad *

Carlos Agon Karim Haddad Gonzalo Romero-García
Sorbonne Université, CNRS, IRCAM, STMS, Paris, France
{agonc,karim.haddad,romero}@ircam.fr

Abstract

This article defines the semantics of maquettes in the visual programming language OpenMusic (OM) using monads. A maquette can be seen as a sequencer of functions. Although maquettes have been widely used, their semantic have never been formalized. Formalizing maquettes has multiple benefits; primarily, we aim to provide a better understanding for composers through the use of a mathematical language rather than discursive semantics. In this work, we use a particular case of the state monad and show with examples how this monad is visualized in OpenMusic. The use of these advanced concepts in the field of music and their availability to composers aligns with our intention to bridge the gap between the theoretical and practical aspects of the intersection between mathematics and music.

CCS Concepts • **Software and its engineering** → **General programming languages**; • **Theory of computation** → *Program analysis*

Keywords computer music, functional programming, monads.

1. Introduction

A sequencer called *maquette* was introduced since the beginning of the OpenMusic (OM) programming language (Agon C. 1998). The main idea behind the maquette was to construct a sequencer not only for temporal objects but also to allow the inclusion of programs (functions) in time. Musically, this sequencer allows to combine the temporal arrangement of musical structures with their specification in the form of musical algorithms. For a description of how maquettes work, see (Bresson J. et al. 2011). For examples of musical pieces composed with maquettes, see (Agon C. et al. 2006).

Maquettes are highly efficient and widely used; however, unlike the rest of OM where a formal semantics is given for programs, maquettes were never formally defined and were always seen as a sequencer with a complex behavior difficult to understand. Thirty years after its computer implementation, this article provides a formal definition of what maquettes are by separating their calculative and constructive behaviors.

The semantic of the maquettes defined in this article is based on the notion of monads, and in particular, the state monad. The use of monads in music is not new; a pedagogical approach can be found in (Hudak P. et al. 2018). A more practical application, which considers the passage of time as a side effect taken in account by the Timed IO monad, is defined in (Janin D. 2020).

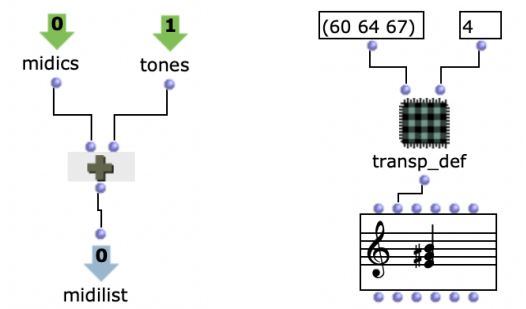
This article is organized as follows. Section 2 provides a quick overview of functional programming from the perspective of category theory, since the monad is a concept originating from this theory. In this section, we also provide a description of visual programs in OM from the perspective of functional programming. Additionally, we informally show how maquettes work and particularly its dual function as both a program and a sequencer.

In Section 3, we provide a definition of the monad from a computer science perspective, which will serve as the basis for defining the semantics of the maquettes. In particular, we will attempt to argue for the necessity of the monad to address the question: what does it mean to evaluate a program over time?

Section 4 properly defines the semantics of the maquette using the state monad. Section 5 presents some examples illustrating how the different components of the monad are visualized within the OpenMusic environment. In particular, we will illustrate how the inclusion of a state allows for certain peculiarities inherent to the maquette, which materialize the temporal and computational aspects of a musical piece.

Finally, in the conclusion, we analyze how the integration of advanced functional programming paradigms, such as monads, with musical composition tools offers new perspectives and methodologies for understanding and manipulating the temporal aspects of musical pieces.

* This research is supported by European Research Council ERC-ADG-883313 REACH, and Agence Nationale de la Recherche ANR-19-CE33-0010 MERCI.



(a) (b)

Figure 1. Two programs in OM.

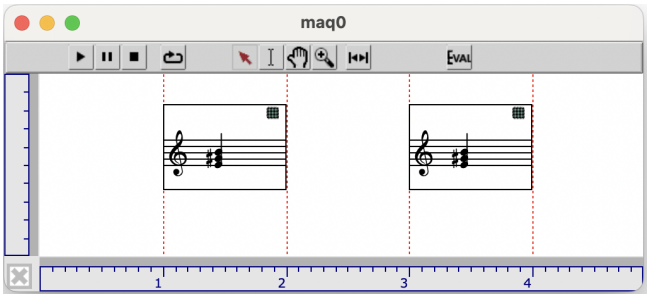


Figure 2. A maquette with two programs.

2. Preliminaries

In category theory, a category consists of a collection of objects and arrows between them, such that these arrows can be composed. Each object in a category has a unique arrow from itself to itself called the identity. For a precise definition of a category, see (S. MacLane 1991). Programs in OM are functions and they live in the "category of types". In the context of functional programming objects in this category are types, and the arrows are functions that transform elements of one type into elements of another type.

The program `transp_def` in Figure 1 (a) has the type $List\ Int \rightarrow Int \rightarrow List\ Int$. It tells us that this program has two inputs: a list of integers and an integer, and it produces a list of integers as a result. Graphically, the inputs and the output of the program are represented as boxes, with `midics` and `tones` for the inputs, and `midilist` for the output.

Programs can be composed with other programs, yielding new programs. Composition is at the core of functional programming. We denote the composition of two functions f and g by $g \circ f$. If f has type $a \rightarrow b$ (noted $f : a \rightarrow b$) and $g : b \rightarrow c$, then $g \circ f : a \rightarrow c$. In Figure 1 (b), the program `transp_def` is called with arguments (60 64 67) and 4, producing a list of numbers that are interpreted as MIDI pitches to create a chord. Informally, the program `trans_def` takes a list of MIDI notes $midics$ and a number of semitones $tones$ and produces a new list $midilist$ as a result of transposing $midics$ by n semitones.

The original idea behind maquettes is to position those programs and their eventual compositions on a timeline to construct a temporal musical structure from their evaluation. Figure 2 shows a maquette in which we have placed the program from Figure 1 (b) at two onsets of 1 and 3 seconds.

The issue with the maquette arises from the fact that when we incorporate functions into a temporal sequencer, it leads us to abandon the traditional semantics of functional programming. This change implies that our programs cease to be "pure" functions. Placing functions in a temporal context introduces two main side effects: time can influence the computation; computation can change the temporal positions of the programs.

Monads come to the rescue for separating the pure functional part of maquettes from the aforementioned side effects. The concept of monad originates from category theory, (M. Barr et al. 1966). Its use in functional programming is due to (E. Moggi. 1991).

3. Monads

The main idea of monads is to change the output type of all functions to a new type that sets the original type in a new context where extra functional aspects can be taken into acocunt. This change of type is made by the help of a functor.

A functor m from a category $\mathcal{C}$ to another category $\mathcal{D}$ takes objects x from $\mathcal{C}$ and produces object $m\ x$ in $\mathcal{D}$. Moreover, for every arrow $f : a \rightarrow b$, the functor produces a new arrow $m\ f : m\ a \rightarrow m\ b$. In the case of functional programming, beacuse the categories $\mathcal{C}$ and $\mathcal{D}$ are the same (i.e., the category of types) we use endofunctors. A classic example of an endofunctor in programming is the $List$ functor, (P. Wadler. 1989), which for every type a produces a new type $List\ a$ and for every function $f : a \rightarrow b$ produces a new function $fmap : a \rightarrow b \rightarrow List\ a \rightarrow List\ b$.

A monadic function is a function whose output type is modified to enrich the function with "extra-functional" actions. A functor m will transform all functions $f : a \rightarrow b$ into new functions $f' : a \rightarrow m\ b$. These new functions are called monadic functions. The problem with this transformation is that it breaks functional composition. If before we could compose two functions f and g we can no longer do this for the two correspondant monadic functions f' and g'. We can not compose $g' \circ f'$ because $f' : a \rightarrow m\ b$ and $g' : b \rightarrow m\ c$. The problem here is that g' expects a b but f' provides it with a $m\ b$. Therefore, we need to define a new composition operator called `bind` and denotated as `>>=`. The type of `>>=` is $m\ b \rightarrow b \rightarrow m\ c \rightarrow m\ c$. The role of `>>=` is to take a monadic value, that is, a value of type $m\ b$, a function $f : b \rightarrow m\ c$; to extract a value x of type b from the monadic value and apply the function f to x, in order to return a monadic value of type $m\ c$.

To construct monadic values, we define an operator called $return : a \rightarrow m\ a$. For example, in the case of the $List$

monad, if we define $return$ as `return x = [x]` we can create monadic values such as `return (3) = [3]`; `return (true) = [true]`, and so on.

DEFINITION 1. *A monad is defined by a triplet consisting of:*

- *a* `functor` m *to define new monadic types,*
- *a* `bind` *operator for the composition of monadic functions,*
- *a* `return` *operator for the construction of monadic values.*

To be a true monad, `return` and `bind` must satisfy the following three laws:

- `(return x) >>= f = f (x)`,
- `mv >>= return = mv`,
- `(mv >>= f) >>= g = mv >>=` λx.`f(x) >>= g`.

The following section defines the semantic of the maquettes by using a particular case of the `state monad`.

4. Maquette as a monad

In order to introduce programs in a maquette, we will use a particular kind of the `state monad`. The `state monad` allows reading and writing data which can be used and modified during the evaluation of a program. The minimal informations we need to put a program into a maquette are its *onset* and its *duration*. In our monad, which we call `Maq` the state is given by a pair (Int, Int) where the first element correspond to the *onset* and the second one to the *duration*.

The idea is to transform any function of type $a \rightarrow b$ into a function of type $(a \times (Int, Int)) \rightarrow (b, (Int, Int))$, it is, a function that, in addition to performing a calculation, can read an *onset* and a *duration* as inputs and return a result along with a new *onset* and a new *duration*, potentially modified. The type thus obtained is not a monadic type, but we can easily write it in the following way: $a \rightarrow (Int, Int) \rightarrow (b, (Int, Int))$. Thus, the functor that constructs our monadic functions, called Maq, can be defined as follows:

```
Maq  b = (Int,Int) → (b,(Int,Int)).
```

Maq transforms any type b into a functional type that, given a pair $(onset : Int, duration : Int)$ returns a type b along with a new pair $(onset : Int, duration : Int)$.

Once the functor Maq is defined, all that remains is to define the `return` and `bind` operators to obtain our monad.

For `return`, which has type $a \rightarrow Maq\ a$, we give the following definition:

```
return x = Maq λ(o,d).(x,(o,d))
```

For example, `(return 5) = Maq` λ`(o,d).(5,(o,d))`. To evaluate the function inside this monadic value, we use the operator `runState` of type $Maq\ a \rightarrow (Int, Int) \rightarrow (a, (Int, Int))$. `runState` takes a monadic value of the Maq monad and an initial pair `(onset, duration)`, and returns a pair containing a result of type a and a new pair `(onset, duration)`.

Thus, for example, `runState ((return 5), (0,1))` will return `(5,(0,1))`.

For `bind (>>=)`, which has type $Maq\ a \rightarrow (a \rightarrow Maq\ b) \rightarrow Maq\ b$, we give the following definition:

```
x >>= g =
   Maq λ(o,d).
    let (x',(o',d')) = runState x (o,d) in
      runState (g x') (o',d'')
```

Take for instance `x = (return 5)` and $g : Int \rightarrow Maq\ List\ Int$ defined as:

```
g n = Maq λ(o,d).([n*2],o+2,d+2)
```

And now let's execute the expression `runState (x >>= g, (2,4))` which is equivalent to executing the following program:

```
let(x',(o',d'))=runState(return 5)(2,4) in
      runState (g x') (o',d'')
```

The result of `runState (return 5) (2,4)` returns `x'=5; o'=2 et d'=4`. With these values

```
(g 5) = Maq λ(o,d).([5*2],o+2,d+2)
```

Finally, `runState (g 5) (2,4) = ([10], (4,6))` which is of type $(List\ Int, (Int, Int))$ as we expected.

We also define 2 functions that allow manipulating the state of monadic values in `Maq`:

- $get : Maq\ (Int, Int)$ which return the state value (o,d) being passed around,
- $put : (Int, Int) \rightarrow Maq\ ()$ to replace the current state value with a new (o,d) state.

The `Maq` monad is a special case of the state monad where the state is fixed to a pair (onset, duration). For this reason, we do not prove monad laws for `Maq`; a proof that state monad satisfies the three monad laws can be found in (B. O'Sullivan. et al. 1991).

5. Graphical Representation of the *Maquette* monad

In OM, a temporal program is a standard program extended with a default input called `state`. Temporal programs are the graphic representation of `Maq` monadic functions. The program in Figure 1 (b) has been extended in Figure 3 (a), where we can see the "state" box with three outputs. The first one return the state of the program, which is the graphic equivalent of the `get` operator. The second and third outputs respectively retrieve the onset and the duration of this state.

A maquette proposes two main relationships between the temporal program it contains: a causality relationship and a temporal relationship. The causality between two temporal programs is defined with respect to the order of evaluation.

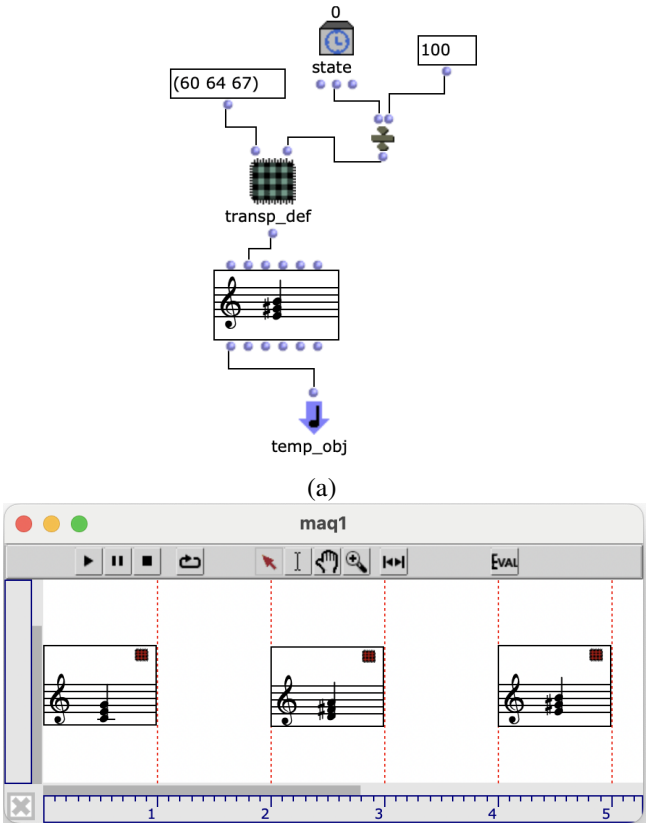


Figure 3. The onset changes the computation.

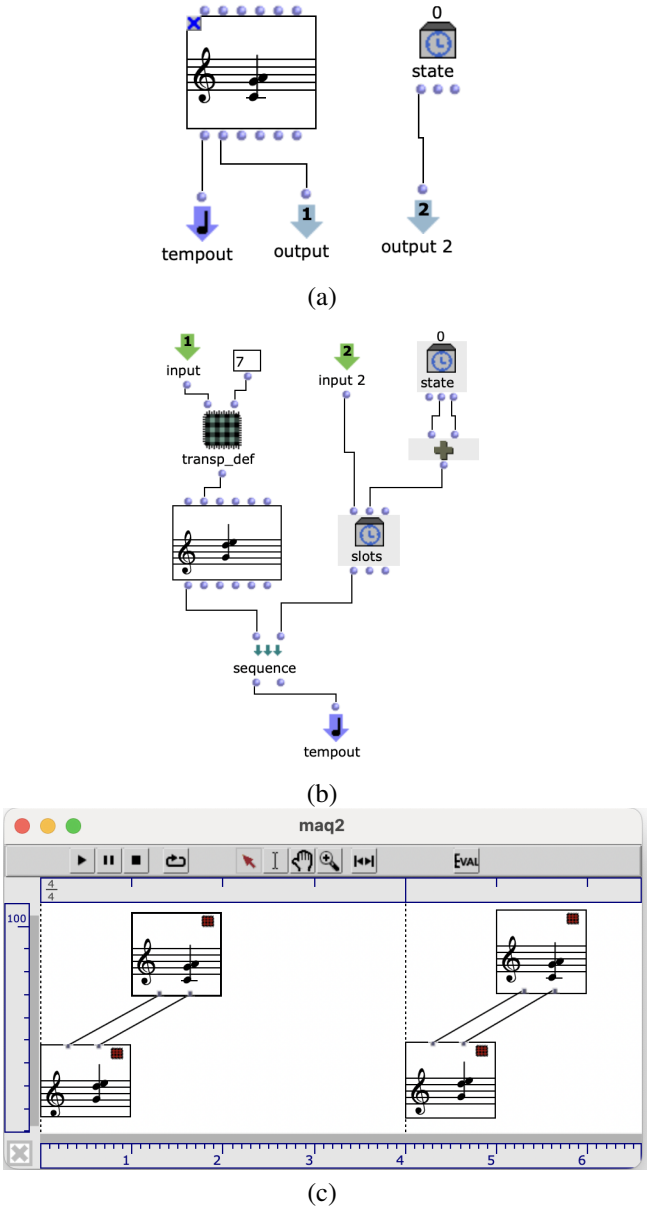


Figure 4. The computation changes the onset.

Similarly, the temporal line of the maquette allows the comparison of temporal program based on their onset. The following examples show possible combinations of these two relationships.

Figure 3 (b), shows the temporal program of Figure 3 (a) situated at three different onsets: 0, 2000, and 4000 milliseconds. Each time the program is evaluated (by using `runState`), a state is given as an argument containing a pair (onset,duration). Since the program has access to this state each time and because the program uses the onset to calculate the number of semitones by which the chord will be transposed, the three results will be different (i.e., transposed by 0, 2, and 4 semitones). In this example, we would say that the calculation result depends on the position of the temporal program. To quote (V. R. Pratt. 1996), one could say that "[temporal programs] bear time and change information".

Figure 4 (c) shows a maquette where two temporal programs are composed. The temporal program located at the top is shown in Figure 4 (a) (we will call it, program 1). This produces a chord but also sends as a result the list of MIDI notes of the chord as well as the state with which the temporal program is called.

The lower temporal program in Figure 4 (b) which we will call Program 2, receives the MIDI notes and creates a chord where notes are transposed by 7 semitones. Additionally, the state of the Program 1 is modified; its onset is set immediately after the onset plus the duration given by Program 2. It doesn't matter where the Program 2 is placed, after the evaluation, Program 1 will change its onset to be immediately after it. In this case, we will say that the calculation conditions the temporal organization. Or, quoting (V. R. Pratt. 1996), again,"states bear information and change time".

Another feature found in this last example consists of a kind of dichotomy between the two orders defined by time and composition. From the composition perspective, Program 1 takes place before Program 2. However, from the

temporal perspective, it is the opposite. This implies that the construction of one chord depends on another that is situated in the future. In other words, the second program, despite being evaluated later, actually runs earlier in time, creating a relationship where the outcome of one chord is conditioned by an event that has not yet occurred in the standard temporal flow. This inversion of orders opens up new possibilities for modeling how events can interrelate in a context where time and composition do not follow the same conventions.

6. Conclusions

Through the formalization of maquettes in OM with the help of monads, this article provides an example of the multiple possibilities of using functional programming paradigms as tools for musical composition. In particular, we aim to offer new perspectives and methodologies for understanding and manipulating the temporal aspects of musical pieces. The primary characteristic of a maquette comes from the possibility of combining computations and representations within the same object. Regarding computation, a maquette as the fonctional composition of temporal functions. However, unlike classical functional composition, in a maquette, the onset and the duration of these temporal programs, can play a determining role in the computation.

This formalization reflects our commitment to popularize access to advanced musical tools, thereby promoting a more inclusive musical landscape. Category theory is a discipline extensively studied by expert musicologists. Mazola's pioneering book,(G. Mazzola 2002), has given rise to many other publications and various applications focusing on theory and analysis. Few studies are based on the use of category theory for compositional purposes, although we can cite the works of (A. Popoff. et al. 2016). With this article, we wanted to show that the concept of monad, specific to category theory and known for its difficulty in understanding, can be explained through a generative application.

For this, we took a detour and first looked at how a monad is used in functional programming. In category theory, a monad is defined as a triplet (T, μ, η) where T is an endofunctor and μ and η are two natural transformations called `join` and `return`, see Figue 5.

Natural transformations in category theory find their equivalent in generic functions in functional programming; thus, μ and η play the roles of bind and return, respectively.

Despite this first abstraction reduction, the monad in functional programming remains an abstract concept difficult to grasp without resorting to a real application. We have attempted to explain in Section 3 the general utility of a monad in the context of functional programming, but we believe it is important to provide a concrete application, such as in the case of maquettes, which might further motivate composers to engage in this type of study.

We conclude this article by arguing that the monad, is above all, an abstraction of functional composition. That

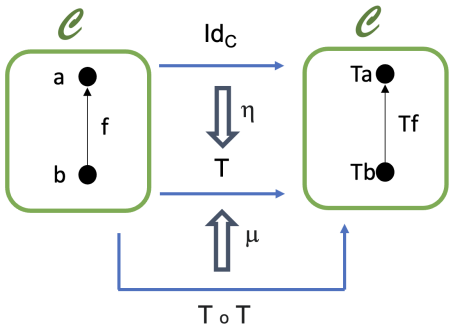


Figure 5. Monad diagram.

is, the monad makes the compositional operator ∘ variable. Understood in this way, the use of monads allows for the definition of spaces where functional composition can be arbitrarily defined (while respecting the laws, of course). This opens a door to the creation of novel relationships between musical components.

OM is open source; it can be downloaded at `https://openmusic-project.github.io`

References

C. Agon .OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur". PhD Paris 6 University, 1998.

C. Agon, Assayag G., Bresson J. The OM Composer's Book 1. Éditions Delatour France / Ircam-Centre Pompidou, 2006.

M. Barr, J. Beck. Acyclic models and triples. Proc. Conf. Categorical Algebra. Springer-Verlag, Berlin (1966), 336–343.

Bresson J., Agon C. Visual Programming and Music Score Generation with *OpenMusic*". *IEEE Symposium on Visual Languages and Human-Centric Computing*. Pittsburgh, 2011.

P. Hudak, D. Quick. The Haskell School of Music : From signals to Symphonies. Cambridge University Press, 2018.

D. Janin. A Timed IO monad. Practical Aspects of Declarative Languages (PADL), New Orleans, United States, 2020.

S. MacLane. Categories for the Working Mathematician. Springer-Verlag, New York, Graduate Texts in Mathematics, Vol. 5. 1971.

G. Mazzola. The topos of music. Birkhäuser Verlag ed. 2002.

E. Moggi. Notions of computation and monads. Information and Computation, 1991.

B. O'Sullivan, et al. Real World Haskell. O'Reilly Media, 2008.

A. Popoff , C. Agon , M. Andreatta , A. Ehresmann. From K-Nets to PK-Nets: A Categorical Approach. Perspectives of New Music. Volume 54, Number 2, Summer 2016.

V. R. Pratt. The Duality of time and information. Stanford University, 1996.

P. Wadler. Computational lambda calculus and monads. In IEEE Symposium on Logic in Computer Science 1989.