From out-of-time design to in-time production of temporal media

Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520

David Janin
LaBRI, Bordeaux INP,
Université de Bordeaux
F-33405 Talence

August 5, 2015

Abstract

The design a temporal media, a sequence of temporal media values such as notes, sounds, images, etc., is an out-of-time task. Fairly general out-of-time program constructs are available for such a purpose. For example, when writing a musical piece, a composer can traverse back and forth his creation. On the contrary, rendering a temporal media is an in-time task. The production of notes in a musical performance is bound to be coherent with the unceasing onward flow of time. It follows that some of the out-of-time programming constructs used for the creation of that pieces must have been re-ordered in order to produce the right media events in the right order and at the right time. In this paper, we propose a formal study of the interplay between these in-time and these out-of-time programing constructs. With an explicitly out-of-time design approach, we eventually show that simpler and more abstract declarative programming features become available, leaving to computers the tedious task of synchronizing and scheduling the media events to be produced in-time, upon demand.

1 Introduction

In many multimedia applications, one of the main task performed by computers is to render temporal media, that is, to produce and perform, in the right order and at the right time, media values: sounds, notes, images, etc. Such a production is an in-time process. It is necessarily performed in the irreversible flow of time. The computation of media values is bound to be coherent with the unceasing onward flow of time.

On the contrary, in the design process of these temporal media, the flow of time is partially reversible. A designer can easily traverse back and forth the

temporal media under to be created. It follows that, in such an out-of-time design process more programming features are available, leaving to computers the task of re-organizing, possibly just upon demand, the temporal media that must be produced.

As an example, a typical in-time program construct consists of specifying the production of values *after* a certain amount of time. This construct can be implemented by means of a timer. In this case, there is an immediate correspondance between the definition of a stream of values to be produced and its *in-time* production. In the out-of-time definition of a stream of value, another program construct consists of specifying that some event must occur *before* a certain amount of time. Even though refeering to the past, this backward dependence can still be implemented provided it does not violate some causality constraints. For example, another timer may be used to count down the remaining time available before the occurrence of a known planed events. However, in that later case, the correspondance between the definition of the stream of values and their effective production of these values is no longer straightforward. It requires to check its consistency with respect to the flow of time and, in the positive case, to retrieve the media values in the appropriate order and time.

In this paper, we propose a formal study of the interplay between such in-time and out-of-time programming features and we eventually show that a simple and efficient program transformation is available in order to convert out-of-time temporal media specification into in-time temporal media production. The tedious task of synchronizing and scheduling the media events to be produced in-time is thus left, upon demand, to the computer.

More precisely, we first define a simple set of out-of-time primitives, namely event productions and back and forth shifts of time, that can be combined one with the other. Then, we show how the resulting (zigzagging) sequence of primitives can be encoded so that an efficient on-the-fly normalization is available to re-order events in the way they have to be produced in-time. Doing so, we eventually recover the notion of tiled temporal media [5], a notion originating from the 80's in the field of computational music [1], that induces a rich algebraic structure. Under adequate assumption, temporal media form an inverse monoid.

Compared to [5], the main novelty of the present paper is that, by implementing out-of-time temporal media constructs and the related on-the fly in-time normalization process, we eventually provide a simple, elegant, and standalone implementation of truly polymorphic tiled temporal media. In [5], we had implicitly provided an encoding of out-of-time (tiled) temporal media into in-time temporal media. In this paper, we explicitly provide an embedding of finite in-time into out-of-time (tiled) temporal media.

Oddly enough to be mentioned, the work presented here essentially amounts to define and study a set of spatiotemporal primitives that can be combined one after the other for creating temporal media somehow much in the same way some spatial primitives are provided in Logo, for the turtle to create pictures.

The remarkable difference is that the pictures resulting from a spatial traversal of the turtle can be displayed as such. On the contrary, the temporal media resulting from the spatiotemporal traversal of the turtle must be normalized in order to be rendered. Then, when this normalization is defined as the in-time retrieval of media events to be rendered, we recover the classical correspondance between normalization and computation.

2 Out-of-time vs in-time temporal media

We give in this section the syntax of out-of-time temporal media definitions and we briefly review their associated in-time semantics described in terms of tiled temporal media. This leads us to the specification of the corner stone functions of our proposal: the head/tail normalization functions. Implementation and other semantics issues are then discussed in the remaining sections.

2.1 Out-of-time syntax

We first need to define a simple set of out-of-time primitives that can be used for designing temporal media: timed sequences of media values, be they sounds, notes, images, animation commands, etc.

The set of primitives we consider is defined as the primitive *event* e that describes the instantaneous event e at the current date, the basic primitive *delay* d that described the (positive or negative) shift of d units of time for the current position (forward or backward) in time. These primitives can then be composed by means of an (infix) sequential composition product $z_1 \,\%\, z_2$ that combined, one "after" the other, any out-of-time specifications z_1 and z_2. An exemple of the resulting zigzags is depicted Figure 1.

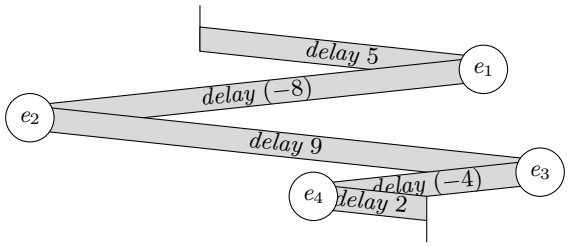

Figure 1: The out-of-time (zigzag) specification z of a temporal media.

In this figure, we have depicted the sequential composition product of the primitives *delay* 5, *event* e_1, *delay* (-8), *event* e_2, *delay* 9, *event* e_3, *delay* (-4), *event* e_4 and *delay* 2.

As far as semantics is concerned, we assume that the composition product is associative. This allows for depicting any out-of-time specification of a temporal

media as a sequence of events related by the forward and backward zigzag in time created by the positive and negative delays.

For the sake of simplicity, we only handle media events that are supposed to be instantaneous. As we shall see later in the text, this takes into account all common situations where media values, lasting for a certain amount of time, can be decomposed into two events, a *media on* event and a *media off* event.

2.2 In-time semantics

Now, we aim at rendering in-time the media events that have been described out-of-time.

In the above out-of-time description of streams of events, we observe that two events may appear syntactically close while, in the flow of time, while they are far one from the other in time.

For exemple, in Figure 1, the events e_2 and e_3 are syntactically close in the product defining the out-of-time specification while, in the flow of time, the event e_2 should be played the first and the event e_3 should be played the last, hence they are separated by all other events.

In some sense, in out-of-time descriptions of a temporal media, media events can be intricate. Executing or rendering such a temporal media thus necessitates to retrieve the events to be launched in an order coherent with the flow of time. This re-ordering defines the in-time semantics *linearize z* of an out-of-time specification z.

More precisely, linearizing an out-of-time description of a temporal media just amounts to order the events (or media values) that appear in a zigzag description. The resulting structure is a tiled temporal media [5] : a list of timed media values (a temporal media in the sense of [3, 4]) that is extended with two synchronization marks that memorized the position in time of the former beginning (for *Pre*) and end (for *Post*) of the out-of-time zigzag specification.

An exemple of a linearization is depicted in Figure 2. It corresponds to the tile *linearize z* obtained by linearizing the out-of-time definition z depicted in Figure 1.

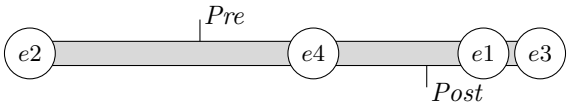

Figure 2: The in-time (linearized) semantics *linearize z* of the out-of-time temporal media specification z.

It occurs that a simple semantical mapping, that maps every of out-of-time specification z to its linearization *linearize z*, induces quite a rich algebraic structure: the algebra of tiled temporal media [5] that is reviewed below under the new point of view provided by out-of-time temporal media.

2.3 Induced tiled temporal media algebra

Let us consider again the zigzag depicted in Figure 1. Assume additionally that the *delay* 9 primitive is cut into two successive delays, *delay* 2 and *delay* 7. Then, as depicted in Figure 3, this cut defines two successive zigzags z_1 and z_2 that can be combined sequentially in order to form the zigzag $z = z_1 \,\%\, z_2$. Then, it can be proved (see [5]) that tiled temporal media can be equipped with

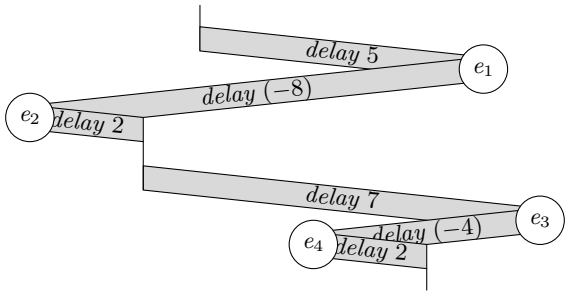

Figure 3: The sequential composition $z = z_1 \,\%\, z_2$ of two out-of-time (zigzag) temporal media z_1 and z_2.

a product, still denoted by %, such that the following property is satisfied :

$$(\mathit{linearize}\ z_1) \,\%\, (\mathit{linearize}\ z_1) = \mathit{linearize}\ (z_1 \,\%\, z_2)$$

In other words, the set of linearized zigzags can be equipped with a product, called the tiled product in [5], such that the mapping *linearize* is a morphism with respect to zigzag sequential composition.

An exemple of a tiled product is depicted in Figure 4. In this figure, the tiled product is detailed as the succession of two elementary steps: the synchronization (or alignment) step and the fusion (or reduction) step.

The first step, the *synchronization step*, amounts to making coincide the *Post* synchronization mark of the first component with the *Pre* synchronization of the second component. Then, the second step, the *fusion step*, amounts to merge the underlying streams of media values.

Although not appearing in the example depicted above, several events may occur at the same time. This may result from a linearization process. This may also just result from the composition of two events. At this stage of our presentation, let us just mentioned that simultaneous events are assumed to be totally ordered and, in the fusion step, they are collected into an ordered multiset.

In some sense, these multisets describe the instantaneous spatial structure that results from the back and forth traversal of the time dimension. All the example depicted above are kept simple so that they only induce singleton (spatial) structures.

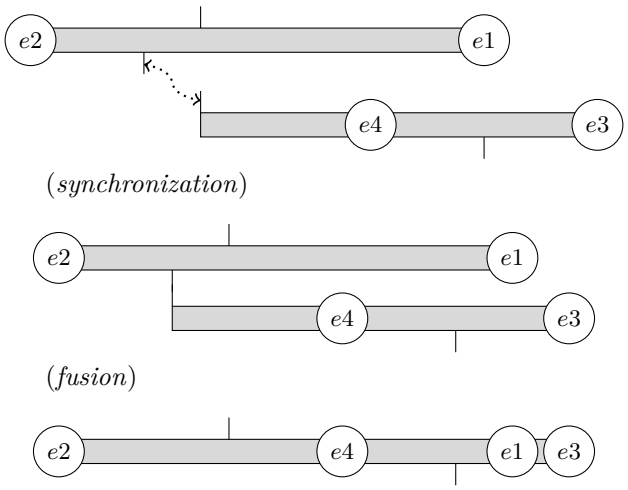

Figure 4: The induced tiled product *linearize* $(z_1 \,\%\, z_2)$ of their in-time (linearized) temporal media *linearize* z_1 and *linearize* z_2.

2.4 The head/tail on-the-fly normalization

We have briefly sketched above the linearization semantics. Now, we aim at providing a slightly more effective view of that linearization mapping. It appears that it can be defined as an on-the-fly inductive normalization process depicted as a repeated head/tail decomposition.

More precisely, this normalization process essentially consists of retrieving the events stored in an out-of-time (zigzag) definition of timed events in an order that is coherent with the flow of time. Here, we show that this can be done by defining two functions *head* and *tail* that behave as follows. For every out-of-time specification z, *head* z contains the earliest events that appear in z and *tail* z contains the remaining events in such a way that the following equation is satisfied

$$z == (\mathit{head}\ z) \,\%\, (\mathit{tail}\ z)$$

where $==$ stands for the equivalence induced by the linearization mapping.

Of course, there are various possibilities for defining *head* and *tail* for achieving such a goal. In this paper, we choose to define *head* z and *tail* z in the following way.

In the case there are no event in z, we put $\mathit{head}\ z = z$ and $\mathit{tail}\ z = \mathit{delay}\ 0$. In the case there are some events in z, we put $\mathit{head}\ z = (\mathit{delay}\ d) \,\%\, (\mathit{event}\ e)$ where d is the relative time distance from the beginning of z to the earliest event in z and where e is that earliest event to be played.

For instance, the zigzag example provided in Figure 1 can be decomposed as depicted in Figure 5. Then, it is quite an easy observation that such a

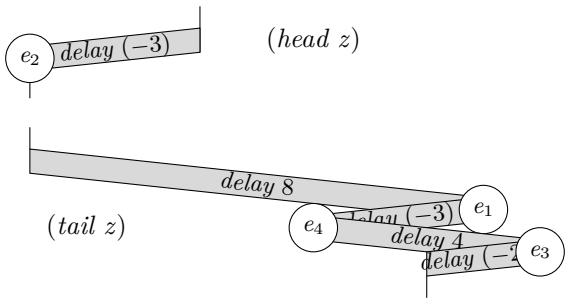

Figure 5: A one step *head* and *tail* normalization of the zigzag z depicted in Figure 1.

normalization can be repeated, taking

$$head\ (tail\ z),\ head\ (tail\ (tail\ z)),\ \text{etc.},$$

till all events have been retrieved.
The result of such a repeated decomposition is depicted in Figure 6 below.

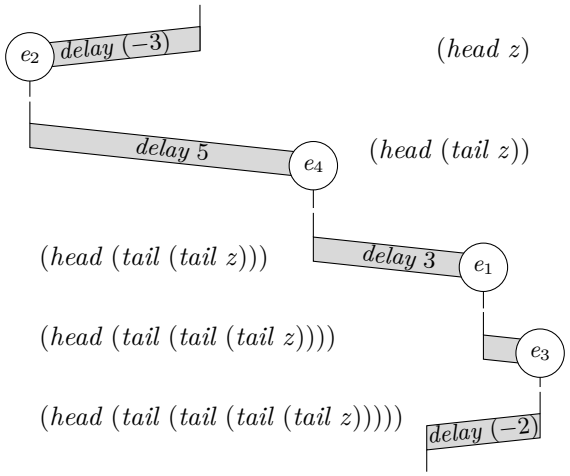

Figure 6: The resulting head/tail decomposition.

Then, it occurs that the repeated head/tail decomposition induces a normal form that may be used to unambiguously represent *linearize* z. Indeed, as depicted in Figure 7, such a normal form can be defined as the sequential composition of the heads of the form $head\ (tail^n z)$ until $tail^n(z)$ equals the unit $Delay\ 0$. In our ongoing example, this happens when $n = 5$.

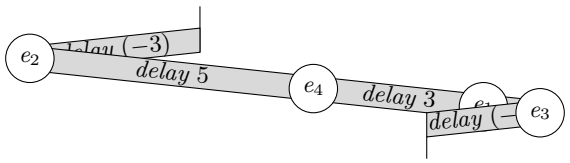

Figure 7: The induced normal form.

This suggests that the mapping *linearize* can be defined by the equation

$$\begin{aligned} \mathit{linearize}\ z \quad = \quad & \textbf{if}\ (\mathit{isUnit}\ z)\ \textbf{then}\ \mathit{delay}\ 0 \\ & \textbf{else}\ \mathit{head}\ t\ \%\ \mathit{linearize}\ (\mathit{tail}\ z) \end{aligned}$$

where *isUnit* z tests if z is equivalent to the unit temporal media *delay* 0 or not.

In other words, the in-time (linearized) semantics of out-of-time temporal media definition can be defined within zig-zag definitions. This opens the way for a standalone implementation of polymorphic temporal media. More precisely, as soon as the two functions *head* and *tail* are defined over out-of-time temporal media definitions, the equivalence induced by the linearization operator is definable. It follows that tiled temporal media can easily be defined as (equivalence classes) of out-of-time temporal media definitions. This is the purpose of the next section.

Observe again that, although this is not the case in our running exemple, it may be the case that several events are to be played at the same time. As already mentioned above for describing the fusion operation, we just assume that these simultaneous events are totally ordered. Then, as described below, they are collected into ordered multisets.

3 Lazy implementation in Haskell

In this section, we implement out-of-time temporal media and the related normalization functions *head* and *tail*. As observed above, this provides an implementation of the polymorphic tiled temporal media defined in [5].

The proposed implementation is lazy in the sense that, when computing the head and the tail of (the syntactic representation of) a zigzag z it never traverses more than what is strictly needed to retrieve the events to be collected into *head* z.

3.1 Preliminary remark on the tail function

Observe that, while the function *head* is completely specified in the previous section, the zigzag representation of *tail* is less clear. The example depicted

above, from Figure 5 to Figure 7, clearly shows that *tail z* can be defined in many ways.

From the point of view of semantics, this is not an issue since the *linearization* function is eventually defined in terms of heads. However, from the implementation point of view, there is an obvious efficiency issue since the *linearization* function is defined by iterated calls of the function *tail*.

In the proposed implementation, observing that *head* and *tail* functions are necessarily computed by doing partially traversals of syntactic representation of zigzags, we make these traversals optimal by precomputing on nodes of zigzag syntactic trees certain time information. These informations, implemented by the functions *dur* and *firstD*, are called synchronization profiles. They are defined below.

3.2 Synchronization profiles

Following [6] and [5], a first basic elements of the synchronization profile of a zig zag (or a tile) is the distance from its beginning (the *pre* synchronization mark of a tile) and its end (the *Post* synchronization of a tile). This distance is computed by the function *dur* that takes a tile (or a zigzag) as input and produces a *Duration* with

type *Duration* = *Rational*

This function is depicted in Figure 8 below.

As it should already be clear in view of the specification of the *head* function, we also need to know if there is an event in zigzags. Moreover, in the positive case, it is the position of the first events that should be known. This leads us

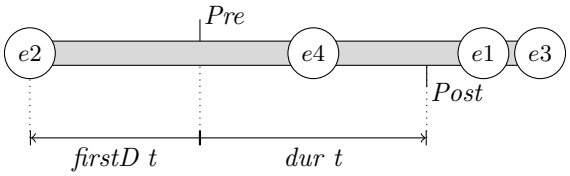

Figure 8: The synchronization profile of *linearize z*.

to the definition of the function *firstD* that takes a tile (or a zigzag) as input and produces a *Date* with

type *Date* = *Maybe Rational*

When there is no event in a tile, the function *firstD* returns *Nothing*. Otherwise, the function *firstD* returns *Just d*, where *d* is the distance from the *Pre* synchronization marks (equivalently the beginning of the zigzag) to the first events. This function is also depicted in Figure 8.

3.3 Tile syntax

The syntax of tile is then defined as follows.

$$\begin{aligned}&\textbf{data } \mathit{Tile}\ a = \mathit{Event}\ (\mathit{MSet}\ a)\\ &\qquad\qquad\ \ |\ \mathit{Delay}\ \mathit{Duration}\\ &\qquad\qquad\ \ |\ \mathit{Twist}\ \mathit{Duration}\ \mathit{Duration}\ (\mathit{Tile}\ a)\ (\mathit{Tile}\ a)\end{aligned}$$

where *MSet a* stands for the type of multisets of elements of type a.

While the purpose of *Event* and *Delay* data constructs is clear, the construct *Twist*, which code product of tiles, needs some explanation.

Firstly, the name twist, comes from the fact that an expression of the form $\mathit{Twist}\ f\ d\ t_1\ t_2$ encodes the product of two tiles, the tile t_1 and the tile $\mathit{inv}\ t_2$. As it shall become clear below, this twisted representation of products allows for encoding a syntactically involutive inverse operator.

Secondly, the argument d denotes the duration of the resulting tile, and the argument f denotes a the position of the first event. Pre-computing these values will prevent useless traversal of the underlying syntactic tree. The fact that f is assume to be of type *Duration* also means that we will guarantee that *Twist* constructs are always denoting tiles that, resulting from a product, contains one event at least.

Following these conventions, the functions *dur* and *firstD* are simply implemented over such syntactic constructs by:

$$\begin{aligned}&\mathit{dur}\ (\mathit{Event}\ \_) = 0\\ &\mathit{dur}\ (\mathit{Delay}\ d) = d\\ &\mathit{dur}\ (\mathit{Twist}\ \_\ d\ \_\ \_) = d\\ &\mathit{firstD}\ (\mathit{Event}\ \_) = \mathit{Just}\ 0\\ &\mathit{firstD}\ (\mathit{Delay}\ d) = \mathit{Nothing}\\ &\mathit{firstD}\ (\mathit{Twist}\ e\ \_\ \_\ \_) = \mathit{Just}\ e\end{aligned}$$

3.4 Tile generators

We have defined above the syntax of tiles. Now we define the tile generators *delay*, *event* and %, enriched with an explicit unit tile *unit*.

$$\begin{aligned}&\mathit{unit} = \mathit{Delay}\ 0\\ &\mathit{event}\ a = \mathit{Event}\ (\mathit{singleM}\ a)\\ &\mathit{delay}\ d = \mathit{Delay}\ d\\ &(\%)\ t_1\ t_2 = \textbf{case}\ (t_1, t_2)\ \textbf{of}\\ &\quad (\_, \mathit{Delay}\ 0) \to t_1\\ &\quad (\mathit{Delay}\ 0, \_) \to t_2\\ &\quad \mathit{otherwise}\quad \to\\ &\qquad \textbf{let}\ d_1 = \mathit{dur}\ t_1\\ &\qquad\quad\ \ d_2 = \mathit{dur}\ t_2\\ &\qquad\quad\ \ d\ \ = (d_1 + d_2)\end{aligned}$$

```
    f1 = firstD t1
    f2 = firstD t2
    f  = minD f1 (shiftD f2 d1)
in case (f) of
     Nothing → Delay d
     Just x  → Twist x d t1 (inv t2)
```

In this code, we use the function *minD*, that compute the minimm of two dates. It is defined by

$$\begin{aligned}
&minD\ Nothing\ d = d\\
&minD\ d\ Nothing = d\\
&minD\ (Just\ d_1)\ (Just\ d_2) = Just\ (min\ d_1\ d_2)
\end{aligned}$$

We also use the function *shiftD* that shifts a date by some duration. It is defined by

$$\begin{aligned}
&shiftD\ Nothing\ d = Nothing\\
&shiftD\ (Just\ d_1)\ d = Just\ (d_1 + d)
\end{aligned}$$

By construction, no *Event* construct may occur with no event at all (the empty list). Similarly, the product is defined in such a way that, as requested above, no *Twist* construct may be used over tiles without defined events.

3.5 Inverse, reset and co-reset

The inverse mapping *inv* is defined by

$$\begin{aligned}
&inv\ (Event\ e) = Event\ e\\
&inv\ (Delay\ d) = Delay\ (-d)\\
&inv\ (Twist\ y\ d\ t_1\ t_2) = Twist\ (y - d)\ (-d)\ t_2\ t_1
\end{aligned}$$

As expected, this implementation of the inverse function is syntactically involutive in the sense that, for every finite tile z, the representation of $inv\ (inv\ z)$ equals the representation of the tile z.

The related reset and co-reset functions are also defined by

$$\begin{aligned}
&re\ t = t\ \%\ (delay\ (-(dur\ t)))\\
&co\ t = (delay\ (-(dur\ t)))\ \%\ t
\end{aligned}$$

Although reset and co-reset are defined directly as above, it is known that they may be defined via the inverse function since, under adequate assumption [5], we have

$$\begin{aligned}
re\ t &== t\ \%\ (inv\ t)\\
co\ t &== (inv\ t)\ \%\ t
\end{aligned}$$

3.6 Normalization function

The head/tail normalization function is computed in two steps. In the first step, the function *norm* make a single (partial) traversal of its argument to compute all elements needed, in the second step, for computing *head* and *tail*.

The type of the function *norm* is defined by

$$norm :: (Ord\ a) \Rightarrow Tile\ a \rightarrow (Duration, MSet\ a, Tile\ a)$$

When $(d, e, tt) = norm\ t$, the duration d is the distance to the earliest events (or zero if there are none), the multiset e contains all these first events, and the tile tt is the remaining tail of the input tile t. This function, defined below, does not meant to be exported.

$$\begin{array}{l}
norm\ (Event\ e) = (0, e, Delay\ 0) \\
norm\ (Delay\ d) = (0, emptyM, Delay\ d) \\
norm\ (Twist\ f\ d\ t_1\ t_2) = \\
\quad \textbf{let}\ f_1 = firstD\ t_1 \\
\qquad f_2 = firstD\ ((Delay\ (dur\ t_1))\ \%\ (inv\ t_2)) \\
\quad \textbf{in case}\ (compareD\ f_1\ f_2)\ \textbf{of} \\
\qquad LT\ \rightarrow \textbf{let}\ (d_1, e_1, tt_1) = norm\ t_1 \\
\qquad\qquad\quad \textbf{in}\ \ (d_1, e_1, tt_1\ \%\ (inv\ t_2)) \\
\qquad GT \rightarrow \textbf{let}\ (d_2, e_2, tt) = norm\ (inv\ (Twist\ f\ d\ t_1\ t_2)) \\
\qquad\qquad\quad \textbf{in}\ \ (d + d_2, e_2, tt\ \%\ (Delay\ d)) \\
\qquad EQ \rightarrow \textbf{let}\ (d_1, e_1, tt_1) = norm\ t_1 \\
\qquad\qquad\qquad\quad (d_2, e_2, tt_2) = norm\ (inv\ t_2) \\
\qquad\qquad\quad \textbf{in}\ \ (d_1, unionM\ e_1\ e_2, \\
\qquad\qquad\qquad\qquad tt_1\ \%\ (Delay\ d_2)\ \%\ tt_2)
\end{array}$$

I n this code, we use the *unionM* that computes unions of multisets. We also use the function *compareD* that compare dates if a way coherent with the function *minD* defined above. It is defined by

$$\begin{array}{l}
compareD\ f_1\ f_2 = \textbf{case}\ (f_1, f_2)\ \textbf{of} \\
\quad (Nothing, Nothing) \rightarrow EQ \\
\quad (_, Nothing) \rightarrow LT \\
\quad (Nothing, _) \rightarrow GT \\
\quad (Just\ x1, Just\ x2) \rightarrow compare\ x1\ x2
\end{array}$$

The case LT above corresponds to the case that the first event of the underlying product is located in the tile t_1. The definition of *compareD* ensures that the tile t_1 contains an event.

The case GT defined above corresponds to the case that the first event of the underlying product is located in the tile t_2. Again, the definition of *compareD* ensures that the tile t_2 contains an event. This case is solved by duality, exploiting the syntactical encoding of inverses.

Lastly, the case *EQ* corresponds to the case that the first events of the underlying product are located both in the tile t_1 and in the tile t_2. The assumption that a *Twist* construct cannot be used when both underlying tiles are empty ensures that both tiles t_1 and t_2 are indeed non empty.

3.7 Head, Tail and induced equality

With the function *norm* given above, the *head* and *tail* functions are then simply defined as follows.

$$
\begin{array}{l}
ht\ t = \textbf{let}\ (d, e, tt) = norm\ t \\
\qquad \textbf{in if}\ (isEmptyM\ e) \\
\qquad\quad \textbf{then}\ (Delay\ (dur\ t), Delay\ 0) \\
\qquad\quad \textbf{else}\ ((Delay\ d)\ \%\ (Event\ e), tt) \\
head\ t = fst\ (ht\ t) \\
tail\ t = snd\ (ht\ t)
\end{array}
$$

where *isEmptyM* is the function that check if its argument is the empty multiset.

As observed in the previous section, the *linearization* function can be defined by means of *head* and *tail*. The semantical equivalence it induces can directly (and lazily) be defined as follows:

$$
\begin{array}{l}
\textbf{instance}\ (Eq\ a, Ord\ a) \Rightarrow Eq\ (Tile\ a)\ \textbf{where} \\
\quad (==)\ t_1\ t_2 = \textbf{case}\ (t_1, t_2)\ \textbf{of} \\
\qquad (Delay\ d_1, Delay\ d_2) \rightarrow (d_1 == d_2) \\
\qquad otherwise \rightarrow \\
\qquad\quad \textbf{let}\ (dd1, ee1, tt_1) = (norm\ t_1) \\
\qquad\qquad\ \ (dd2, ee2, tt_2) = (norm\ t_2) \\
\qquad\quad \textbf{in}\ ((dd1 == dd2) \\
\qquad\qquad\quad \wedge\ (equalM\ ee1\ ee2) \\
\qquad\qquad\quad \wedge\ (tt_1 == tt_2))
\end{array}
$$

where the function *equalM* is the equality of finite multi-sets.

It can be shown that the complexity of computing the (semantical) equality $z_1 == z_2$ of two out-of-time definition of temporal media z_1 and z_2 equals the sum of their syntactic size multiply by the depth of the underlying syntactical tree. In other words, provided this depth is kept small, our encoding of the linearization mapping is quasi-linear in the size of its arguments.

4 Conclusion

In the experiment described in the above pages, we thus show that allowing out-of-time programming constructs such as back and forth time shifts, eventually leads to a confortable language for specifying temporal media. Still, the specified

media events can be automatically reordered in an efficient way for the in-time rendering of these temporal media.

Quite strikingly, the complete implementation is achieved in less than a hundred lines of Haskell code. This comes in particular from the twisted encoding of the products that allows to reason by duality. This also comes from the robustness of the underlying mathematical objects.

It can easily be shown that classical (in-time) temporal media [3] can be embedded and thus re-encoded into the (out-of-time) tile formalism that is presented here. Even though of a rather theoretical nature, further experiments, with audio or midi temporal media are currently done in order to validate this approach in practice.

The on-the-fly head/tail decomposition that we have proposed and implemented here may sound familiar to the programmers known FRP [2]. Although going out of the scope of the presented experiments, it is believed that tiles can be used for multi-scale descriptions of temporal media. Indeed, more abstract temporal media can be defined as sequences more concrete tiled temporal media. The fact tiled temporal media may overlap should facilitate such a hierarchical design approach.

Beyond temporal media, our experiments examine the distinction one can make between program design, that may go back and forth in the space of causal dependencies, and program execution, that must respect causal dependencies. To which extent our approach can be extended to such a much broader scope is left to further investigations.

References

[1] P. Desain and H. Honing. LOCO: a composition microworld in Logo. *Computer Music Journal*, 12(3):30–42, 1988.

[2] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 25–36. ACM, 2009.

[3] P. Hudak. An algebraic theory of polymorphic temporal media. In *Proceedings of PADL'04: 6th International Workshop on Practical Aspects of Declarative Languages*, pages 1–15. Springer Verlag LNCS 3057, June 2004.

[4] P. Hudak. A sound and complete axiomatization of polymorphic temporal media. Technical Report RR-1259, Department of Computer Science, Yale University, 2008.

[5] P. Hudak and D. Janin. Tiled polymorphic temporal media. In *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*, pages 49–60. ACM Press, 2014.

[6] D. Janin, F. Berthaut, M. DeSainte-Catherine, Y. Orlarey, and S. Salvati. The T-calculus : towards a structured programming of (musical) time and space. In *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*, pages 23–34, Boston, USA, 2013. ACM Press.

The code of this experiment is available at

`http://www.labri.fr/~janin/Code/code.hs`