Structured reactive programming with polymorphic temporal tiles

Simon Archipoff, David Janin

Structured reactive programming with polymorphic temporal tiles

Simon Archipoff & David Janin
CNRS LaBRI, Inria Bordeaux,
Bordeaux INP, Bordeaux University
F-33405 Talence
archipoff|janin@labri.fr

July 29, 2016

Abstract

In functional reactive programming (FRP), system inputs and outputs are generally modeled as functions over continuous time (behaviors) whose future values are governed by sudden changes (events). In this approach, discrete events are embedded into piece-wise continuous behaviors.

In the field of reactive music system programming, we develop an orthogonal approach that seems to better fit our need. Much like piano keys can be played and combined both in sequence and in parallel, we model system inputs and outputs as spatio temporal combinations of what we call temporal values: continuous functions over time whose domain lays between two events: a start and a stop event.

Various high level data types and program constructs can then be derived from such a model. They are shown to satisfy robust algebraic and category theoretic properties. Altogether, this eventually provides a simple, robust and elegant programming front-end, temporal tile programming, for reading, memorizing, stretching, combining and transforming flows of inputs into flows of outputs.

Although at its infancy, the resulting approach has been experimentally validated for reactive and real-time music system programming.

Contents

1 Introduction

In this article we consider the problem of programming timed reactive systems that are stimulated by structured inputs such as, for instance, flows of notes that may be produced by a piano keyboard.

With a view towards application in music system programming, we must combine both data (or space) programming, that is, the values of the notes are to be computed, and control (or time) programming, that is, the moments these notes are to be played. This drastically contrasts with standard approaches developed since the mid 70s, where, on the contrary, programming languages tend to make a clear distinction between data and control.

Model-driven approaches. In terms of methodology, we aim at developing an abstract model for reactive system behaviors from which we can derive a domain specific language (DSL) suitable for programming such a kind of systems.

The main advantages of such a model driven approach[1] are that the resulting DSL is:

robust: each application programming interface (API) derives from models of programs and behaviors,

sound: programs semantics are based of well-defined mathematical model that can be analyzed,

testable: primitive functions satisfy various invariant equations that can be tested via automatic test generation tools such as *QuickCheck* [4],

pervasive: each application programming interface may evolve following enduser needs, but the underlying semantical model does not change over time.

Of course, as secondary goals, such a DSL is also expected to be *simple*. This means that the underlying model should also reflect user intuition. Ideally, it can be taught to and understood by end-user: in our case, musicians and composers.

We should insist on the fact that, following a model-driven approach, the underlying model can be derived into various APIs, each of those being targeted towards a given group of endusers: from children to engineers, via artists or even just amateurs. Even more, the algebraic properties and category theoretic properties of the underlying model and the related functions may also lead to various kind of visual representations: the basis of efficient and well-defined GUIs.

In such a research context, general functional programming languages and, especially, pure typed functional programming languages such as Haskell [13], offer, via the definition of an embedded domain specific language (DSL) [9], one of the most rapid and robust way to put in practice and experiment the underlying abstract model candidates.

[1] by opposition to, say, an enduser-needs based approach

A signal-based approach: FRP. Functional Reactive Programming (FRP) is an example of such a model-driven approach. It provides a model of timed reactive system behavior [6].

In this model, the flow of notes produced by a piano keyboard is viewed as a piecewise continuous function

$$behav : D \to \mathcal{P}(N) \tag{1}$$

from a given set D of *time stamps*[2] into the powerset $\mathcal{P}(N)$ of the set of possible notes N. At every time stamp t, the value $behav(t)$ represents the set of *active* notes at that time.

Pressing a key n at a given time t_1 creates an *On* n event that adds the note n to the set $behav(t_1)$ of active notes at time t_1. At a later time t_2, releasing the key n creates an *Off* n event that removes the note n from the set $behav(t_2)$ of active note at time t_2. In this approach, discrete events are embedded into piecewise continuous behaviors.

Assuming that notes are indeed active over intervals of the form $[t_1, t_2[$, we obtain a faithful model of piano behaviors.

However, at the application programming interface level, with a view towards application in computer music, one may question the relevance of such a model. Shall a composer use such a model for describing the flow of notes created by a piano and the transformations he may wish to apply on this flow of notes?

Basic musical elements such as played notes and their durations, generally explicit in music application software,[3] get a bit lost in such an approach. They are not encoded as primitive objects.

An algebra-based approach: PTM. Of course, at the programming level, music modeling has already been investigated. For instance, polymorphic temporal media (PTM) is an expresive algebra [10, 11] for modeling both the spatial and the temporal aspects of music pieces.

In such an approach, each played note[4] is a primitive object. At run time, it is defined by a start date, a value, and a duration, from which a stop date can be derived.

In other words, as already observed by Hudak [10], a flow of notes produced by a piano keyboard can be modeled as a function

$$behav' : D \to \mathcal{P}(N \times D) \tag{2}$$

At every time stamp t, the value $behav'(t) \subseteq N \times D$ is a set of pairs of the form $(n, d) \in N \times D$ where n is a note that starts at time t for some duration d.

In contrast, with the above approach that is depicted in Equation (1), the resulting behavior $behav'$ is a discrete function; we have $behav'(t) = \emptyset$ for most time-stamp t. Indeed, altogether, the time stamps where notes are started form a discrete set over the time scale represented by D. In this approach, continuous behaviors are embedded between discrete pairs of events.

[2] or, equivalently, a set of *duration* such a rational numbers of seconds elapsed since some origin of time

[3] via music score or piano roll visualizations interfaces

[4] or the related continuous signal heard when a note is played

Such an alternative does provide a handy application programming interface as illustrated by Paul Hudak's textbook on music programming [12]. However, to the best of our knowledge, it has not yet been used for programming reactive systems.

The main reason for this is rather simple. When a program may react upon the reception of an *On n* event, that is, upon the start of a note, the duration of that note is yet unknown. It depends on the reception of the *Off n* event that is yet not received. It follows that in most (not to say all) available programming languages, the duration of that note is not a primitive object. It must be computed explicitly by recording all received *On n* events and by waiting for the corresponding *Off n*. Durations can then be computed.

Our contribution. The main contribution of our paper is to provide a reactive encoding of Paul Hudak's notion of temporal media algebra suitable for reactive programing. More precisely, we eventually design a domain specific language (DSL) where durations are modeled as primitive objects.

This DSL is embedded in Haskell. Most of its features are directly translated into some Haskell functions and data types henceforth following a shallow embedding style [8]. However, since duration may be unknown values before being later updated, some duration comparisons, the associated conditionals and, more generally, some function applications, may need to be postponed to the last moment for durations to be known when computed. This implies that, part of the encoding of our DSL is also syntactical, with explicit syntactic application constructs, henceforth also following a deep embedding style [8]. Still, durations *are* handled as primitive objects in the sense that these updates are handled automatically via an adequate reactive and timed execution kernel.

In other words, our paper essentially describes: (1) a runtime that allows to convert and execute on-the-fly functions over flows of notes and (2) an application programming interface (API) based on a slight algebraic enrichment of flows of notes.

The resulting domain specific language is called the T-calculus[5], or temporal tile programming. It has already been successfully experimented for reactive and real-time music system programming[6].

Organization of the paper. We first define in Section 2, the notion of timed queue list (*QList*), a data type that implements flows of notes, or, more generally, the semantics of polymorphic temporal media as depicted in Equation (2). Their constructors, their getters and basic properties of queue lists such as functor properties are detailed in this section.

The reactive kernel is presented in Section 3. It essentially consists in converting every function over timed queue lists into reactive function over lists over timestamped events. This shows how queue lists can actually be used for specifying and programming timed reactive systems. The underlying runtime is implemented in Haskell over the UISF library [23].

More advanced category theoretic properties: categorical product, co-product and exponents, of functions over queue lists are presented in Section 4. Each

[5]see `http://poset.labri.fr/tcalculus/`

[6]see `http://poset.labri.fr/interpolations/`

of these properties leads to the definition of various functions that act on these functions. Their usage are illustrated by simple (reactive) system programs.

Last, extending queue lists with synchronization marks leads to the definition of the temporal tiles algebra. Equipped with tile addition (Section 5) and tile multiplication (Section 6), the resulting algebra, that also inherits from the categorical properties of queue lists, constitute the high level font-end of our proposal: temporal tile programming.

Comparison with existing work are summarized in the last section before giving a brief description of what remains to be done as a conclusion.

2 The QList data type

The *QList* data type, whose values are simply called queue lists, is presented in this section. It encodes flows of temporal values as in Equation (2).

2.1 Definition

Given a duration type d assumed to be an instance of the type class *Num*, given a value type v and an additional[7] input value type iv, queue lists are defined in Haskell by:

$$\begin{aligned}\textbf{data}\ \mathit{QList}\ d\ \mathit{iv}\ v = {} & \mathit{QList}\ [(d, v)]\ d\ (\mathit{QList}\ d\ \mathit{iv}\ v)\\ & \mid \mathit{QEnd}\end{aligned}$$

The data constructor *QList* takes as first parameter a list of temporal values represented by pairs of durations and values. Theoretically, a value of type v can be a time dependent function even though, in most examples, they are constant values. The second parameter is the duration between the beginning of the temporal values of the first parameter, and the very next beginning of values (encoded by the third parameter, recursively). The data constructor *QEnd* describes the empty queue list.

Example. An example of a finite queue list is depicted in Figure 1. In this

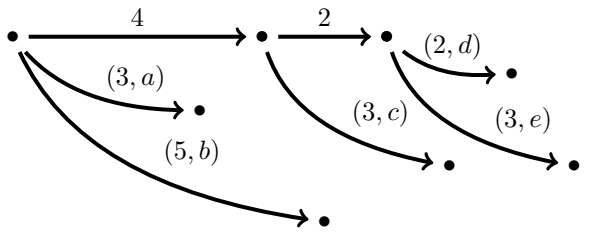

Figure 1: A QList

example, at time 0 from the origin of the queue list starts a value a that last for a duration 3, in parallel with an other value b that last for a duration 5. At time 4 starts an third value c that last for duration 3. Then, after an additional

[7] only used for type constraint propagation as detailed in the next section

delay of duration 2, the temporal values $(2, d)$ and $(3, e)$ occur. From the origin of the queue lists, these temporal values starts at time $4 + 2 = 6$.

2.2 Queue lists semantics

In queue lists, lists of temporal values that start at the same time are called *bundles* of temporal values. As far as semantics is concerned, these bundles are understood as sets. This assumption have two important consequences.

First, the order into which temporal values occur in lists is irrelevant. This assumption is called the *parallel commutativity* law.

Second, since sets have no multiplicity, every temporal values occurs at most once. This assumption is called the *parallel idempotency* law.

The semantical equivalence over queue lists, denoted by $==$, is thus defined by the fact that, at every position in time, bundles of temporal values starting at that time are equal. This way, we easily recover the semantical notion of flow of temporal values that has been sketched in Equation (2).

Crucial invariants that should always be satisfied by queue lists are the following. All durations are greater than or equal to zero. Bundles of temporal values are necessarily non empty unless starting at the origin of the queue list. We also assume that all durations between bundles of temporal values are strictly greater than zero. This means that successive bundles of notes can be numbered without any ambiguity. Last, the duration to the empty queue list *QEnd* is also assumed to be zero. These invariants provide, up to the set-wise interpretation of lists in queue lists, a canonical representation of classes of equivalent queue lists.

2.3 Queue lists constructors

The main constructors for queue lists, in addition to the empty queue list, are the following functions:

$$
\begin{aligned}
&\mathit{fromAtomsQ} :: \mathit{Num}\ d \Rightarrow [(d, v)] \to \mathit{QList}\ d\ \mathit{iv}\ v \\
&\mathit{fromAtomsQ}\ \mathit{la} = \mathit{QList}\ \mathit{la}\ 0\ \mathit{QEnd} \\
&\mathit{shiftQ} :: (\mathit{Num}\ d, \mathit{Eq}\ d) \Rightarrow d \to \mathit{QList}\ d\ \mathit{iv}\ v \to \mathit{QList}\ d\ \mathit{iv}\ v \\
&\mathit{shiftQ}\ 0\ q = q \\
&\mathit{shiftQ}\ d\ (\mathit{QList}\ [\,]\ d_1\ q) = \mathit{QList}\ [\,]\ (d + d_1)\ q \\
&\mathit{shiftQ}\ d\ q = \mathit{QList}\ [\,]\ d\ q \\
&\mathit{mergeQ} :: (\mathit{Ord}\ d, \mathit{Num}\ d) \Rightarrow \\
&\qquad \mathit{QList}\ d\ \mathit{iv}\ v \to \mathit{QList}\ d\ \mathit{iv}\ v \to \mathit{QList}\ d\ \mathit{iv}\ v \\
&\mathit{mergeQ}\ \mathit{QEnd}\ q_2 \quad = q_2 \\
&\mathit{mergeQ}\ q_1 \quad \mathit{QEnd} = q_1 \\
&\mathit{mergeQ}\ (\mathit{QList}\ \mathit{dv1}\ d_1\ q_1)\ (\mathit{QList}\ \mathit{dv2}\ d_2\ q_2) = \\
&\quad \textbf{let}\ q = \mathit{QList}\ (\mathit{dv1} +\!\!+ \mathit{dv2}) \\
&\quad \textbf{in case}\ \mathit{compare}\ d_1\ d_2\ \textbf{of} \\
&\qquad \mathit{EQ} \to q\ d_1\ (\mathit{mergeQ}\ q_1\ q_2) \\
&\qquad \mathit{LT} \to q\ d_1\ (\mathit{mergeQ}\ q_1\ (\mathit{QList}\ [\,]\ (d_2 - d_1)\ q_2)) \\
&\qquad \mathit{GT} \to q\ d_2\ (\mathit{mergeQ}\ q_2\ (\mathit{QList}\ [\,]\ (d_1 - d_2)\ q_1))
\end{aligned}
$$

The *fromAtomsQ* function creates a *QList* from a single value and its duration. The *shiftQ* function shifts a queue list by some (positive or null) duration in

time. The *mergeQ* function merges two queue lists such that each operand becomes sort of sub queue list of the resulting merge. Observe that *mergeQ* is a parallel composition function in the sense that, as a result of merge, temporal values may occur in parallel.

Example. With these constructors, the beginning of the example depicted in Figure 1 that contains the first three temporal values can be defined by

$$x = mergeQ\ (fromAtomsQ\ [(3, a), (5, b)])\\ (shiftQ\ 4\ (fromAtomsQ\ [(3, c)]))$$

Remark. The code provided above has been simplified. For instance, applying *shiftQ* over a negative duration creates a runtime error. Also, handling unknown duration as described in the next section makes this code significantly more complex. We refer the interested reader to the source code available on the internet for more details.

Invariants. Thanks to the parallel commutativity law, the function *mergeQ* is commutative, that is,

$$mergeQ\ q_1\ q_2 == mergeQ\ q_2\ q_1$$

Thanks to the parallel idempotency law, the function *mergeQ* is also idempotent, that is,

$$mergeQ\ q\ q == q$$

In particular, as expected with our set-wise interpretation of bundles of temporal values, two temporal values that occur at the same time and that are equal, that is, they have the same duration and the same value, are always (implicitly) merged into a single one.

2.4 Queue lists getters

The getters of *QList* are defined much in the same way *head* and *tail* are defined over lists. More precisely, the analogous of *head* is defined as a pair of functions *atomsQ* and *delayToTailQ* respectively describing the bundle of temporal values starting at the beginning of the queue list and the delay from that beginning to the next bundle of temporal values.

More precisely, these functions can be implemented as follows.

$$atomsQ :: QList\ d\ iv\ v \to [(d, v)]\\ atomsQ\ (QList\ l\ \_\ \_) = l\\ atomsQ\ (QEnd) = [\,]$$

$$delayToTailQ :: Num\ d \Rightarrow QList\ d\ iv\ v \to d\\ delayToTailQ\ (QList\ \_\ d\ \_) = d\\ delayToTailQ\ (QEnd) = 0$$

$$tailQ :: QList\ d\ iv\ v \to QList\ d\ iv\ v\\ tailQ\ (QList\ \_\ \_\ q) = q\\ tailQ\ (QEnd) = QEnd$$

As expected, over a non empty queu lists, the *atomsQ* function returns the initial bundle of temporal values, that is, the list of pairs (d, v) that start at the beginning of the queue list. In the case the remainder of the queue list is non empty, the *delayToTailQ* function returns the duration between the beginning of the queue list and the beginning of next bundle of temporal values. Finally, the *tailQ* function returns the queue list defined from that next bundle, thus pruning out the bundle of temporal values that start at time 0 (*atomsQ*) and the delay to the next bundle (*delayToTailQ*).

Invariant. For every queue list q, given the queue list $qa = fromAtomsQ \circ atomsQ\ q$, representing the bundle of temporal values starting at 0, given $dt = delayToTailQ\ q$ the delay to the tail of q and given $tq = tailQ\ q$ the tail of q, the invariant $q == mergeQ\ qa\ (shiftQ\ d\ tq)$ is always satisfied.

2.5 Functor like properties

Queue lists of type *QList d iv v* can be seen as containers of durations of type d and values of type v. As such, functor like properties allows defining lifts of functions.

The first one, the function *fmapQ*, allows to lift a function over values to a function over queue lists. It can be defined by

$$
\begin{aligned}
&fmapQ :: (v_1 \to v_2) \to QList\ d\ iv\ v_1 \to QList\ d\ iv\ v_2\\
&fmapQ\ f\ (QEnd) = QEnd\\
&fmapQ\ f\ (QList\ l\ d\ q)\\
&\quad = QList\ (map\ (\lambda(d, v) \to (d, f\ v))\ l)\ d\ (fmapQ\ f\ q)
\end{aligned}
$$

This allows to define a *Functor* instance by

$$
\begin{aligned}
&\textbf{instance}\ Functor\ (QList\ d\ iv)\ \textbf{where}\\
&\quad fmap = fmapQ
\end{aligned}
$$

The second one, the function *fmapDQ*, lifts a function over duration into a function over queue lists. It can be define by

$$
\begin{aligned}
&fmapDQ :: (d \to d) \to QList\ d\ iv\ v \to QList\ d\ iv\ v\\
&fmapDQ\ f\ (QEnd) = QEnd\\
&fmapDQ\ f\ (QList\ l\ d\ q)\\
&\quad = QList\ (map\ (\lambda(d, v) \to (f\ d, v))\ l)\\
&\qquad\qquad (f\ d)\ (fmapDQ\ f\ q)
\end{aligned}
$$

On purpose, its usage is limited to duration type preserving functions. Indeed, in the reactive and real-time execution context we aim at achieving, a change of time scale is not easy to define and handle.

An immediate application of the duration map *fmapDQ* is to define the *stretchQ* function, that multiplies every duration in a queue list by a given factor. It can be define by

$$
\begin{aligned}
&stretchQ :: Num\ d \Rightarrow d \to QList\ d\ iv\ v \to QList\ d\ iv\ v\\
&stretchQ\ d\ q = fmapDQ\ (\lambda x \to d * x)\ q
\end{aligned}
$$

In musical terms, all rests and notes in the queue list are stretched by a factor d.

Remark. The code above only applies to the case when d is positive. The case when d is negative creates a runtime error, and, since duration invariants must be preserved in queue lists, when d equals zero, all bundles must be merged instead at the beginning of the queue lists, all the temporal values being set to zero.

3 Real-Time/reactive kernel

Defined as bundles of temporal values separated by strictly positive delays, queue lists, or rather functions over queue lists, should be enough for modeling reactive system behavior. However, in a reactive execution, the duration of a temporal value is not known until the reception of its end. Similarly, the delay between two successive bundles of temporal values is not known until the reception of the second bundle of temporal values.

This difficulty is solved in our implementation by allowing explicitly unknown duration values, every application of a function to a partly unknown argument being frozen so that this argument can be updated till the last moment. Doing so, we recover a notion similar to the improving values used in some implementations of FRP [7].

3.1 From queue lists to lists of events

The proposed back end, that is, the execution layer of our DSL, is based on well-bracketed events. The event data type is defined by

$$\textbf{data}\ \mathit{Event}\ v = \mathit{Start} \mid \mathit{On}\ \mathit{VID}\ v \mid \mathit{Off}\ \ \mathit{VID} \mid \mathit{Stop}$$

with an ad hoc value identification type (VID) for matching every *Off* event to its corresponding *On* event. The parameter v bring by each *On* event is the value associated with the corresponding pair of events. *Start* and *Stop* events are added as global start and end markers.

Given the *Event* v type, one can define two functions

$\mathit{qListToEvents} :: \mathit{QList}\ \mathit{iv}\ d\ v \to [(d, \mathit{Event}\ v)]$

$\mathit{eventsToQList} :: [(d, \mathit{Event}\ v)] \to \mathit{QList}\ \mathit{iv}\ d\ v$

where the type $[(d, \mathit{Event}\ v)]$ denotes streams of timestamped events, that verify

$$\mathit{eventsToQList} \circ \mathit{qlistToEvents} == \mathit{id}$$

and

$$\mathit{qlistToEvents} \circ \mathit{eventsToQList} == \mathit{id}$$

in the case every stream starts with a *Start* event and ends with a *Stop* event. In other words, queue lists can be converted back and forth into events.

Our goal in this section is to convert every function over queue lists to a function over time-stamped stream of events that can be executed on-the-fly. In other words, we aim at defining

$\mathit{runFunction} :: (\mathit{QList}\ \mathit{iv}\ d\ \mathit{iv} \to \mathit{QList}\ \mathit{iv}\ d\ v)$
$\qquad \to ([(d, \mathit{Event}\ \mathit{iv})] \to [(d, \mathit{Event}\ v)])$

that verifies the following property

$\mathit{eventsToQlist}\ (\mathit{runFunction}\ f\ (\mathit{qListToEvents}\ q)) == f\ q$

for all queue list. Then, the runtime of our language will just consist in writing and reading the events passing through "arrowized" functions of the form *runFunction f* with *f* describing the system behavior at the queue lists level.

3.2 Reactive/real-time structured input

We present here the first component of our reactive kernel: the definition of an explicit input that can be built upon reception of start and stop event, and its translation into queue list.

This input is built on-the-fly according to the following data-type:

$$\begin{aligned}&\textbf{data } \mathit{InQList}\ d\ iv = \mathit{InQList}\ [(d, iv)]\ d\ (\mathit{InQList}\ d\ iv)\\ &\qquad\qquad\qquad\qquad \mid \mathit{InQUndef} \mid \mathit{InQEnd}\end{aligned}$$

In this encoding, *InQUndef* describes the part of the input that is yet unknown. Such an input structure is converted into a queue list by the function:

$$\begin{aligned}&\mathit{inputToQ} :: \mathit{InQList}\ d\ iv \to \mathit{QList}\ d\ iv\ iv\\ &\mathit{inputToQ}\ (\mathit{InQList}\ ial\ d\ iq)\\ &\quad = \mathit{QList}\ al\ d\ (\mathit{QRec}\ \mathit{inputToQ}\ iq)\\ &\mathit{inputToQ}\ (\mathit{InQEnd}) = \mathit{QEnd}\\ &\mathit{inputToQ}\ (\mathit{InQUndef}) = \mathit{error}\ \texttt{"Causality error"}\end{aligned}$$

One can observe that a call to (*inputToQ InQUndef*) creates a runtime error. Indeed, in our implementation, such a call denotes a computation that requires part of the input is yet not available. This corresponds to a runtime *causality error*.

Additionally, we introduce here a new polymorphic queue list data-type constructor *QRec* of type:

$$\begin{aligned}&\mathit{QRec} :: (\mathit{Updatable}\ p\ d\ iv) \Rightarrow\\ &\qquad (p \to \mathit{QList}\ d\ iv\ v) \to p \to \mathit{QList}\ d\ iv\ v\end{aligned}$$

which essentially amounts to freeze function application.

More precisely, in an expression of the form *QRec f p*, the application *f p* is frozen until the moment it will be needed. The class constraints *Updatable p d iv*, presented and detailed below, ensures that the argument of type *p* can be updated until that last moment.

Remark. A large part of our DSL is implemented in a shallow way: program constructs are directly translated into Haskell functions (see [8]). However, the constructor *QRec* and the associated class *Updatable* mix such a shallow embedding with a deeper one. Indeed, by freezing function application, the *QRec* construct clearly gives a syntactic flavor to the queue lists data type.

3.3 Freezable and updatable

The class type *Updatable p d iv* mentioned above is simply defined by:

$$\begin{aligned}&\textbf{class } \mathit{Updatable}\ p\ d\ iv\ \textbf{where}\\ &\quad \mathit{update} :: \mathit{UpdateData}\ d\ iv \to p \to p\end{aligned}$$

where the type *UpdateData d iv* is defined by:

```
type UpdateData d iv
   = (d → d, InQList d iv → InQList d iv)
```

This type specifies which kind of updates may be performed. The following instance shows how these updates are applicable. The first argument updates durations, the second argument is essentially used to replace the undefined input *InQUndef* by a more defined one.

```
instance Updatable (InQList d iv) d iv where
  update (f, nq) (InQList al d q)
     = InQList (fmap (λ(d1, v1) → (f d1, v1)) al)
       (f d) (update (f, nq) q)
  update (_, _) InQEnd = InQEnd
  update (_, nq) InQUndef = nq InQUndef
```

The role of the queue list constructor *QRec* becomes especially clear in the following code for queue lists updates.

```
instance Updatable (QList d iv v) d iv where
  update _ QEnd = QEnd
  update (f, nq) (QList al d q)
     = QList (update (f, nq) al) (f d)
             (update (f, nq) q)
  update (f, nq) (QRec g p) = QRec g (update (f, nq) p)
```

Of course, the class type *Updatable* is closed under product, sums and list type constructor as expected, thus allowing various types of updatable function arguments.

Remark. Observe the type *InQList* presented above is only defined in terms of one input value type *iv*. Then, the function *inputToQ* creates homogeneous queue lists of type *QList d iv iv*. Then, keeping such a type annotation all through functions acting over types allows us to remember the type of temporal values that may be updated, especially in the frozen construct defined by *QRec*.

3.4 Updatable duration

As already mentioned, the duration type that is needed for reactive execution should allow unknown durations. In our current implementation, these durations are simply encoded as a type *Dur d* of affine functions[8] built over a basic duration type *d* of the class *Num* and two sets of unknowns: one for input temporal values and another for delays between bundles of temporal values.

The arrival of a new bundle of temporal values generates a new input of the form

$$InQList\ [(X_{i,j}, v_{i,j})_{j\in J_i}]\ Y_i\ InQUndef$$

that will replace the former obsolete *InQUndef* occurences.

In this schema, i is the rank of such a new bundle, J_i is an ad hoc list of identifiers for the arrived values, $X_{i,j}$ is the unknown duration of the jth value

[8] though durations are instances of the *Num* and the *Fractional* class types, our (runtime and ad hoc) restriction to affine functions leads to a more tractable type than the rational functions that would arise from unrestricted duration combinations with unknowns

$v_{i,j}$ and Y_i is the unknown duration to the next bundle of temporal values. Off course, all variable indices are encoded by means of a single index type *ID*

Durations are then updated by means of two possible functions.

The *shiftUnknown dt* function, where *dt* is some elapsed duration, replaces every duration variables X by $X + dt$. This way, all unknowns are measured from the current timestamp *now*.

The *closeUnknown i* function, where $i :: ID$ is the index of an unknown variable, sets every occurence of the unknown variable X_i to zero. This allows updating unknown duration upon reception of the *Off* event of a given temporal value, or, for inter-bundle delays, the reception of a new bundle of such values.

Doing so, all unknowns are necessarily positive which allows a fairly simple traitement of the underlying affine functions.

3.5 Reactive states and loops

We can describe now the heart of our reactive kernel: the *on-the-fly* application of a function over queue lists to a time-stamped stream of events.

This kernel is based on the notion of reactive state, initialized by some function over queue lists and a state loop that converts timed streams of input events into timed streams of output event according to the specification provided by the function f.

The data type *QState* for encoding reactive state is defined by:

```
data QState d iv v
   = QState Int d (QList iv (Dur d) v) [(Dur d, v)]
```

with initializing function

```
initQState :: (QList iv (Dur d) iv → QList iv (Dur d) v)
              → QState d iv v
initQState f
   = QState 0 0 (QRec (f ∘ inputToQ) InQUndef) [ ]
```

The first parameter of *QState* is the rank of the last received bundles of temporal values, the second parameter of type d is the current time-stamp, the third parameter of type *QList* is the scheduled output, possibly built over unknown durations, and the last parameter of type $[(Dur\ d, v)]$ the list of temporal values that are currently active as output and waiting for their ends.

The reactive loop *updateStateOnEvent*, launched upon every new input events or elapsed timer on the output, can be informally described as the following sequence of tasks:

1. updates the internal queue list and the active temporal values according to the time elapsed since the last update,

2. upon reception of *Start*, *Stop* or *On* events, creates the new bundles of temporal values for replacing *InQUndef*, updates the input (and all frozen parameter) accordingly,

3. upon reception of *Off* events, sets the corresponding unknown duration to zero, both in the scheduled output queue list and in the list of active output temporal values,

4. possibly unfreezing some function applications which values are needed now, removes the atoms from the scheduled output that must be started now and load them into the active output list,

5. output the *On* and *Off* events accordingly, removing from the active output list every temporal values that is over,

6. and, in the case the *delayToNextT* value of the scheduled output queue list is a non zero constant duration, set up a timer for a wake-up according to that duration.

In other words, every function f over queue lists can be loaded into a *QState*, so that, together with the function *updateStateOnEvent*, it has been converted into a state machine defined over input and output streams of timestamped events. Such a state machine can then be run over the *UISF* reactive library [24, 23].

3.6 Taking and dropping queue lists

A typical example of function over queue lists that, to be run on-the-fly, necessitates to freeze its argument is the function $takeQ\ d$.

This function cuts all temporal values of a queue list that starts d units of time after its beginning. It can be defined by

$$
\begin{array}{l}
takeQ :: (Ord\ d, Num\ d) \Rightarrow \\
\qquad d \to (QList\ iv\ d\ v) \to (QList\ iv\ d\ v) \\
takeQ\ \_\ QEnd = QEnd \\
takeQ\ d\ (QList\ al\ d_1\ q_1) = \\
\quad \textbf{case}\ compare\ 0\ d\ \textbf{of} \\
\qquad GT \to QEnd \\
\qquad EQ \to atomsQ\ al \\
\qquad LT \to mergeQ\ (atomsQ\ al) \\
\qquad\qquad (shiftQ\ d_1\ (QRec\ (\lambda(d, q) \to takeQ\ d\ q) \\
\qquad\qquad\qquad\qquad (d - d_1, q_1)))
\end{array}
$$

In this code, one can observe that the recursive evaluation of *takeQ* is frozen in such a way that both the duration $d - d_1$ and and the queue list q_1 can be updated.

A function *DropQ* can be defined similarly in such a way that for every duration d, every queue lists q, we have

$$q == mergeQ\ (takeQ\ d\ q)\ (shiftQ\ d\ (dropQ\ d\ q))$$

Remark. Observe that the function $q \mapsto dropQ\ d\ q$ with strictly positive d cannot be run in a reactive system. Indeed, this function is expected to output every temporal value d units before they have arrived. Quite similarly, the function $q \mapsto stretchQ\ d\ q$ with $0 \leqslant d < 1$ cannot be run in a reactive way.

This illustrates the price we have to pay for the abstraction proposed here via function over queue lists. Some of these functions cannot be run in a reactive manner because they are temporally non causal, that is, there is an output value to be produced at a given time that depends on an input value that is yet not arrived.

The problem of analyzing causal vs non-causal properties of queue lists functions is clearly fundamental when programing reactive systems. However, its study, orthogonal to the purpose of the present paper and still to be completed, is postponed to further work.

4 Category theoretic properties

We have just seen that (causal) functions over queue lists define reactive system behavior. In this section, we aim at defining various operators that act on these functions. These operators can then be used for combining simple reactive system specifications into more complex ones.

For such a purpose, we review some categorical properties of functions over queue lists. More precisely, we consider the category defined, for some fixed types d and iv, by the types $QList\ iv\ d\ v$ as objects and by the function types $QList\ iv\ d\ v_1 \to QList\ iv\ d\ v_2$, as arrows. This category is simply called the category of queue lists, and the type $QList\ iv\ d\ v_1 \to QList\ iv\ d\ v_2$ is shortened into $FuncQL\ d\ iv\ v_1\ v_2$.

Following an approach well established in typed functional programming theory (see e.g. [2]), the properties of the category of queue lists induce numbers of function combinators that can be used for programming.

More precisely, we will see how categorical product and co-product allows to mix and separate queue lists at will. Then, combined with the functor property, we will see how categorical exponent allows us to apply on-the-fly transformations on parts of some input, these transformations being simply commanded by buttons that can be pressed and released on a GUI.

4.1 Categorical product and weak sum

Defining the type $Void$ as the empty type, it is an easy observation that $QList\ d\ iv\ Void$ is, up to equivalence of queue lists, only inhabited by the empty queue list $QEnd$. Then, one can check that the type $QList\ d\ iv\ Void$ is a terminal object in the category of queue lists: there is a unique arrow from every object into that one. The type $QList\ d\ iv\ Void$ is also weakly initial: there is at least on arrow from that object into any other.

The existence of a terminal object suggests to seek for a categorical product. The fact that the terminal object is also weakly initial suggests that such a categorical product could also be a weak categorical sum. This turned out to be true.

More precisely, let us now consider the type

$$QList\ d\ iv\ (Either\ v_1\ v_2)$$

where $Either\ v_1\ v_2$ is the *Haskell* type for the disjoint union of the types v_1 and v_2. Let $fromLeftQ$ and $fromRightQ$ be the functions defined by

$$\begin{aligned}
&fromLeftQ :: (Eq\ d, Num\ d) \Rightarrow \\
&\qquad FuncQL\ d\ iv\ (Either\ v_1\ v_2)\ v_1 \\
&fromLeftQ\ x \\
&\quad = \mathbf{let}\ lf = [(d, v) \mid (d, Left\ v) \leftarrow atomsQ\ x] \\
&\qquad \mathbf{in}\ mergeQ\ (map\ fromAtomsQ\ lf)\ \$
\end{aligned}$$

```
                shiftQ (delayToTail x)
                       (QRec (fromLeftQ ∘ tailQ) x)
fromRightQ :: (Eq d, Num d) ⇒
              FuncQL d iv (Either v1 v2) v2
fromRightQ x
  = let lf = [(d, v) | (d, Right v) ← atomsQ x]
    in mergeQ (map fromAtomsQ lf) $
                shiftQ (delayToTail x)
                       (QRec (fromRightQ ∘ tailQ) x)
```

Let also *toLeftQ* and *toRightQ* be the functions defined by

```
toLeftQ :: (Eq d, Num d) ⇒
           FuncQL d iv v1 (Either v1 v2)
toLeftQ = fmap Left
toRightQ :: (Eq d, Num d) ⇒
            FuncQL d iv v2 (Either v1 v2)
toRightQ = fmap Right
```

We easily observe that for every queue list q of type *QList d iv* (*Either* v_1 v_2) we have

$$q == mergeQ\ (toLeftQ \circ fromLeftQ\ q)\ (toRightQ \circ fromRightQ\ q)$$

This suggests that *QList d iv* (*Either* v_1 v_2) is both the product and the sum of *QList d iv* v_1 and *QList d iv* v_2 with functions *fromLeftQ* and *fromRightQ* as canonical projections, and functions *toLeftQ* and *toRightQ* as canonical injections. This is almost true.

Product. Given the function *factorProductQ* defined by

```
factorProductQ :: (Eq d, Num d) ⇒
  (FuncQL d iv v v1) → (FuncQL d iv v v2)
    → FuncQL d iv v (Either v1 v2)
factorProductQ f1 f2 t
  = mergeQ (toLeftQ (f1 t)) (toRightQ (f2 t))
```

we indeed have

$$f_1 == fromLeftQ \circ factorProductQ\ f_1\ f_2$$
$$f_2 == fromRightQ \circ factorProductQ\ f_1\ f_2$$

for every queue list functions f_1 and f_2. Since it can be shown that no other factorization exists, this ensures that *QList d iv* (*Either* v_1 v_2) with projections *fromLeftQ* and *fromRightQ* is the expected categorical product.

Weak sum. Given the function *factorSumQ* defined by

```
factorSumQ :: (Eq d, Num d) ⇒
  (FuncQL d iv v1 v) → (FuncQL d iv v2 v)
    → FuncQL d iv (Either v1 v2) v
```

$$
\begin{aligned}
&\mathit{factorSumQ}\ f_1\ f_2\ t \\
&\quad = \mathit{mergeQ}\ (f_1\ (\mathit{fromLeftQ}\ t))\ (f_2\ (\mathit{fromRightQ}\ t))
\end{aligned}
$$

we observe that we have

$$
\begin{aligned}
\mathit{factorSumQ}\ f_1\ f_2\ (\mathit{toLeftQ}\ t) &== f_1\ t \\
\mathit{factorSumQ}\ f_1\ f_2\ (\mathit{toRightQ}\ t) &== f_2\ t
\end{aligned}
$$

However, there exist other factorizations. By conditionally adding an arbitrary temporal value to $\mathit{factorSumQ}\ f_1\ f_2$ as soon as *both* left and right temporal values have appeared on the input queue list, we indeed obtain another function that satisfies the above equivalencies.

In other words, the type $\mathit{QList}\ d\ iv\ (\mathit{Either}\ v_1\ v_2)$ with injections *toLeftQ* and *toRightQ* is a *weak categorical sum* in the category of queue lists.

Altogether, following category theory terminology, we could say that the category of queue lists is a *weak semi-additive category* with *weak biproducts*. In turns out that many more function combinators than those described here can also be defined.

Application example. When building a reactive music system, one may need to handle multiple inputs and outputs. Categorical sums and products allow for doing this fairly easily.

For instance, the system we want to design may receive a musical input from a keyboard, with temporal values of type a, as well as a *On / Off* control input coming from some buttons in a GUI, with temporal values of type b. The related events can thus be merged into events built over the type $\mathit{Either}\ a\ b$. Then, as detailed in Section 3, these events will generate an input queue list of type $\mathit{QList}\ d\ iv\ (\mathit{Either}\ a\ b)$.

This means that, even though there are several input sources, the specification of a system reacting to these inputs can still be defined by means of a single function over queue lists. Indeed, thanks to the above projections, whenever needed, the mixed input can be projected either on its right component or on its left component.

Quite symmetrically, the existence of weak sums allows recombining, still within the same function over queue lists that specifies our reactive system, various produced queue lists that are merged into a single output. For instance, we can easily produce a queue list of type $\mathit{QList}\ d\ iv\ (\mathit{Either}\ c\ d)$. Thanks to our reactive kernel, this queue list is then translated in events of type $\mathit{Either}\ c\ d$. These mixed events are then easily routed to their adequate destinations; events of type c controlling, say, some music device and events of type c controlling, say, some visualization device.

In other words, categorical products and weak sums allows internalizing circuit like routing mechanisms within the definition of functions over queue lists. As a matter of fact, this has been realized in interpolation experiments[9].

4.2 Categorical exponent

The category of queue lists admits terminal objects and products. Is it also cartesian closed? The answer to that question is also positive provided the type

[9] `http://poset.labri.fr/interpolations/`

of duration admits a maximal element: the infinite duration *Top*.

Indeed, with function *applyQ* defined below, one can show that the type *QList d iv* (*FuncQL d iv* v_1 v_2) is the exponent object associated with functions from *QList d iv* v_1 to *QList d iv* v_2.

```
applyQ :: QList d iv (FuncQL d iv v1 v2)
            → QList d iv v1 → QList d iv v2
applyQ (QList [] _ QEnd) t = QEnd
applyQ f q
  = let  dfl = getAllFromAtomsQ f
         df = delayToTailQ f
         evalA [] _ = []
         evalA ((d1, f1) : l) q@(QList d ad tq)
           = (f1 (QList d ad (takeQ (d1 − ad) tq))) :
             evalA l q
    in mergeQ (evalA dfl q) $
         shiftQ df (QRec (λ(f, q) → applyT f q))
         (tailQ f, dropD df q)
```

with *dropD d* a function that cuts a QList by a duration d from its input.

Indeed, one can prove that, up to semantical equivalence, the property

$$applyQ\ (fromAtomsQ\ (Top, f))\ q = f\ q$$

holds for every queue list function f and every queue list t with adequate types.

Application example. From a theoretical point of view, such a property is certainly appealing though predictable in some sense.

With a view towards application, the interests of embedding queue list functions into queue lists is of a much higher interest. Indeed, the activation duration of a function embedded as a temporal value can also be finite. This means that applying a queue list of functions over an input queue list means that each function is applied to a certain portion of the input. In reactive music system programing, the application possibilities are infinite.

For instance, receiving as input a queue list of type *QList d iv* (*Either a b*), with an input type a describing some GUI buttons activation and deactivation, and an input type b describing some musical content, one can easily transform elements of type a into functions of type *FuncQL d iv b c*. Then, projections and application of the function *applyQ* defined above allows for producing an output of type *QList d iv c* which corresponds to the on-the-fly application on the musical input of the functions activated by the GUI's buttons. Again, such a possibility is used in our experiments[9].

5 The additive tile algebra

So far, we have defined the notion of queue lists to represent flows of temporal values. We have also shown that functions over queue lists can be compiled, thanks to our reactive kernel, into a reactive system runtime.

However, our presentation is yet not complete since we still lack of a convenient front end for endusers such as composers to defined functions over queue

lists. In this section and the next one we show how queue list can be embedded into a richer model: temporal tiles. These tiles are first informally described below before their encoding over queue lists being formally defined.

5.1 Primitive tiles from temporal values

Simply put, embedded in a tile, a temporal value (d, v) can be depicted as in Figure 2. In this picture, the value v is *rendered* from a start event, received

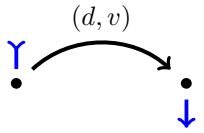

Figure 2: A primitive tile.

at its *input mark* (ϒ), to a stop event, produced at its *output mark* (↓). The duration d of the resulting tile x is denoted by $|x|$.

Writing $Tile\ d\ iv\ v$ for the type of tiles built over the duration type d and the value type v, the lift of temporal values into tiles is encoded as a function:

$$fromDurationAndValueT :: Num\ d \Rightarrow \\ d \to v \to Tile\ d\ iv\ v$$

5.2 Delay primitive tiles

Delays are defined as tiled representation of temporal values bearing no value at all but their duration. They play a crucial role in our proposal since they allow us to view durations as a particular case of temporal tiles. The tile representation of a delay is depicted in Figure 3. Again, the duration of the resulting tile is

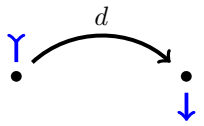

Figure 3: A delay.

the time elapsed from the start of the delay to its end.

The embedding of durations into temporal tiles is defined by the function:

$$fromDurationT :: Num\ d \Rightarrow d \to Tile\ d\ iv\ v$$

that transforms every duration d into a delay. With the duration function, for every duration d, we have

$$|fromDurationT\ d\ | == d$$

5.3 Primitive tile sum

The first operator defined over tiles is the tiled sum that combines two tiles by synchronizing the output mark of the first one with the input mark of the second one and by merging their temporal contents.

In the simplest case of primitive tiles, the sum can be depicted as in Figure 4. The duration $|x+y|$ of the resulting tile satisfies the equality $|x+y| == |x|+|y|$.

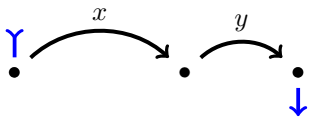

Figure 4: The sum $x + y$ of two primitive tiles x and y.

Such a property, taken as an invariant property, shall always be satisfied.

Observe that, in this example, the sum appears as a typical sequential composition operator. Having in mind that temporal values are rendered in time, the sum is not commutative. It is however associative, that is,

$$x + (y + z) == (x + y) + z$$

This means that the function *fromDurationT* from durations to delays acts as an additive morphism w.r.t. to tile. In particular, still denoting by 0 the zero duration delays, we have

$$x + 0 == x == 0 + x$$

that holds for arbitrary tile x. In other words, tiles equipped with sums form an additive monoid with the zero delay 0 as neutral element.

5.4 Negation and difference

Non classical algebraic properties appear with the definition of the negation of a tile defined by inverting the input mark and the output mark of that tile. In the simpler case of primitive tiles, the negation can be depicted as in Figure 5. The duration $|-x|$ of a negated tile is simply defined as the negation $-|x|$ of

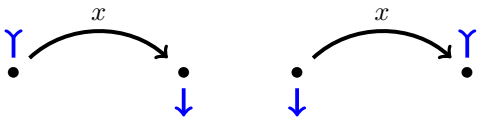

Figure 5: A primitive tile x (on the left) and its negation $-x$ (on the right).

its duration $|x|$. We should insist on the fact that the content of the negated tile has been left unchanged. There is no reverse construction that has been performed. It is just a matter of synchronization marks exchange.

The true interest of negation appears when defining the difference $x - y$ of two tiles by $x + (-y)$. This operation and its dual $-x + y$ over primitive tiles are depicted in Figure 6.

Figure 6: Differences $x-y$ (on the left) and $-x+y$ (on the right) of two primitive tiles x and y.

We easily check that $-(x + y) == -y + -x$, that is, the negation acts over tiles as an anti-morphism. Concerning duration, we have $|x - y| = |x| - |y|$ and $| - x + y| = |y| - |x|$. In particular, we have $|x - x| = 0 = | - x + x|$.

As it shall become clear in the sequel, we do not expect that $x - x == 0$ for arbitrary tiles. On the contrary, the tile algebra implements a preservation principle over temporal values in such a way that no temporal value that has been produced shall ever disappear...

Remark. While the sum of two primitive tiles can be interpreted as a sequential composition, their differences introduce some parallelism. In the difference $-x + y$, the two primitive tile are synchronized on their starts. Dually, in the difference $x - y$, the two primitive tiles are synchronized on their ends. This shows that, with negation, the sum is neither a sequential nor a parallel composition operator, it is both. This property constitutes the first main feature of the notion of temporal tile programming.

5.5 From queue lists to temporal tile

Combining primitive tiles and their negation eventually creates zigzags [15]. Rendering a zigzag amounts to normalizing its content into a single queue list. Such a tight relationship between tiles and queue lists is presented in this section.

Simply put, the type *Tile d iv v* is defined from the type *QList d iv v* by the following constructor:

data *Tile d iv v* = *Tile d d* (*QList d iv v*)

An expression of the form *Tile d ad q* can simply be depicted as in Figure 7. In this figure, we observe that the first duration value d denotes the distance

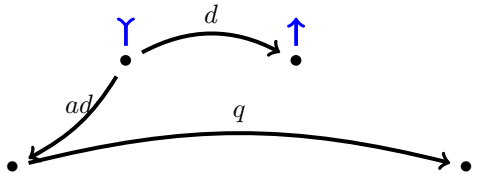

Figure 7: Embedding queue lists into tiles.

between the input and the output synchronization mark of the tile. The second duration value *ad* denotes the distance from the input of the tile to the beginning

of the actual content of the tile: a queue list. In this example, *ad* is negative hence the queue lists starts before the input root.

Functions *fromDurationT* and *fromDurationAndValueT* are simply encoded by

$$
\begin{aligned}
&\mathit{fromDurationT}\ d = \mathit{Tile}\ d\ 0\ \mathit{QEnd}\\
&\mathit{fromDurationAndValueT}\ d\ v\\
&\quad = \mathbf{case}\ \mathit{compare}\ 0\ d\ \mathbf{of}\\
&\qquad GT \to \mathit{Tile}\ d\ 0\ (\mathit{fromAtomsQ}\ [(-d, v)])\\
&\qquad \_ \to \mathit{Tile}\ d\ 0\ (\mathit{fromAtomsQ}\ [(d, v)])
\end{aligned}
$$

Then, tile sum and tile negation are simply encoded by:

$$
\begin{aligned}
&\mathit{sumT}\ (\mathit{Tile}\ d_1\ \mathit{ad1}\ q_1)\ (\mathit{Tile}\ d_2\ \mathit{ad2}\ q_2)\\
&= \mathbf{let}\ s = \mathit{ad1} - d_1 - \mathit{ad2}\\
&\quad d = d_1 + d_2\\
&\quad \mathbf{in\ case}\ \mathit{compare}\ 0\ s\ \mathbf{of}\\
&\qquad LT \to \mathit{Tile}\ d\ \mathit{ad1}\ (\mathit{mergeQ}\ q_1\ (\mathit{shiftQ}\ s\ q_2))\\
&\qquad EQ \to \mathit{Tile}\ d\ \mathit{ad1}\ (\mathit{mergeQ}\ q_1\ q_2)\\
&\qquad GT \to \mathit{Tile}\ d\ (d_1 + \mathit{ad2})\ (\mathit{mergeQ}\ (\mathit{shift}\ (-s)\ q_1)\ q_2)\\
&\mathit{negT}\ (\mathit{Tile}\ d\ \mathit{ad}\ q) = \mathit{Tile}\ (-d)\ (\mathit{ad} + d)\ q
\end{aligned}
$$

In other words, in the sum of two tiles, the queue lists representing the contents of these tile are always merged, their synchronization in time before this merge being automatically handled by tile synchronization marks: tiles input and output roots.

Remark. Again, we have slightly simplified the code above, assuming that duration are totally ordered. In presence of unknown durations, that is, whenever used in a reactive context, some durations may be incomparable. In this case, we produce a runtime error for it means that the implemented function is not causal.

5.6 Parallel commutativity and idempotency laws

Thanks to queue lists semantics, with bundles of temporal values understood as sets, tiles also inherit from the parallel commutativity and idempotency laws.

Applied to tiles, these laws are interpreted as follows. Parallel commutativity law: for all temporal tiles x and y we have

$$\text{if } |x| == |y| \text{ then } x - y == y - x$$

This essentially says that the order in which x and y are rendered when positioned in parallel does not matter. In particular, this means that there is no possible asymmetric covering (or alteration) of a temporal value by another one when played in parallel.

Idempotency law: for all temporal tiles x we assume that

$$x - x + x == x$$

Again, this axiom implies that rendering many times the same temporal value in parallel is equivalent with rendering it only once.

The consequences of these two axioms are numerous (see [14] and [15]). They imply in particular that, up to the induced equivalence, the tile $-x$ is the *semigroup inverse* of the tile x in the sense of inverse semigroup theory [19], that is, the tile $-x$ is the *unique* tile y such that $x+y+x == x$ and $y+x+y == y$.

In other words, provided both axioms are satisfied, every type *Tile d iv v* equipped with sum and negation is an inverse monoid [19] with the zero duration delay as neutral element.

5.7 Delays and natural partial order

It is an easy observation that the function *fromDurationT* from arbitrary durations to delays is not only on-to-one, but it defines an additive group morphism that preserves sums and negation. This suggest that, in our intended programming language, durations can just be treated as delays. More precisely, we define the function:

$$\begin{aligned} &\mathit{delayT} :: \mathit{Num}\ d \Rightarrow \mathit{Tile}\ d\ \mathit{iv}\ v \rightarrow \mathit{Tile}\ d\ \mathit{iv}\ v_1 \\ &\mathit{delayT}\ (\mathit{Tile}\ d\ \_\ \_) = \mathit{Tile}\ d\ 0\ \mathit{QEnd} \end{aligned}$$

As expected, the function *delayT* is a morphism from the inverse monoid of temporal tile into itself, that is,

$$\mathit{delayT}\ (x + y) == \mathit{delayT}\ x + \mathit{delayT}\ y$$

and

$$\mathit{delayT}\ (-x) == -\mathit{delayT}\ x$$

for every tiles x and y. Moreover, it is a projection since we have $\mathit{delayT} \circ \mathit{delayT} == \mathit{delayT}$.

Inverse semi-group theory says a little more about the relationship between a tile x and the delay defined by $\mathit{delayT}\ x$. More precisely, every inverse semigroup can be equipped with a partial order relation, called the *natural order*, defined by $x \leqslant y$ when $x == y - x - x$ or, equivalently, $x == x - x + y$. This partial order is stable under sum, that is, if $x \leqslant y$ then $z + x \leqslant z + y$ and $x + z \leqslant y + z$. Then it appears that, for every temporal tile x, $\mathit{delayT}\ x$ is the *maximum element above* x in the natural order.

6 The multiplicative tile algebra

In Section 5 above, we have defined the additive algebra of temporal tile that forms an additive inverse monoid. As already observed, the sum of two tiles is not a sequential composition operator nor a parallel composition operator, but both.

Combining stretches with more explicit parallel features deriving from the tile sum leads to the definition of a multiplicative algebra over tiles. As a result, tiles come equipped with sum, negation, product and division so that they eventually form an instance of the Haskell classes *Num* and *Fractional*.

6.1 Resets and coresets

The idempotency law provides a fairly simple way to move the output to the input (an operation we call a *reset*) or to move the input to the output (an operation that we call *coreset*). These two operations are depicted in Figure 8.

Figure 8: The reset $\mathrm{re}\,[x] == x - x$ and the coreset $\mathrm{co}\,[x] == -x + x$ of a tile x.

More generally, we define the functions reset and coreset over lists of tiles, by

$$\begin{aligned}
&\mathrm{re} :: (\mathit{Num}\ d) \Rightarrow [\,\mathit{Tile}\ d\ \mathit{iv}\ v] \to \mathit{Tile}\ d\ \mathit{iv}\ v\\
&\mathrm{re}\ [\,] = \mathit{fromDurationT}\ 0\\
&\mathrm{re}\ (x : \mathit{xs}) = x - x + \mathrm{re}\ \ \mathit{xs}\\
&\mathrm{co} :: (\mathit{Num}\ d) \Rightarrow [\,\mathit{Tile}\ d\ \mathit{iv}\ v] \to \mathit{Tile}\ d\ \mathit{iv}\ v\\
&\mathrm{co}\ [\,] = \mathit{fromDurationT}\ 0\\
&\mathrm{co}\ (x : \mathit{xs}) = -x + x + \mathrm{co}\ \ \mathit{xs}
\end{aligned}$$

6.2 Parallel fork and join

From the reset and coreset functions defined above, we can derive two explicit parallel operators defined by:

$$\begin{aligned}
&\mathit{parForkT}, \mathit{parJoinT} :: (\mathit{Num}\ d) \Rightarrow\\
&\qquad \mathit{Tile}\ d\ \mathit{iv}\ v \to \mathit{Tile}\ d\ \mathit{iv}\ v \to \mathit{Tile}\ d\ \mathit{iv}\ v\\
&\mathit{parForkT}\ x\ y = \mathrm{re}\,[x] + y\\
&\mathit{parJoinT}\ x\ y = x + \mathrm{co}\,[y]
\end{aligned}$$

which are depicted in Figure 9.

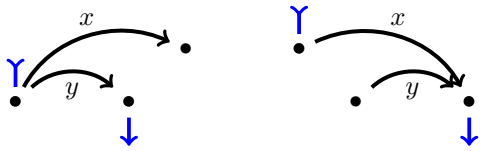

Figure 9: Parallel fork (on the left) and parallel join (on the right) of two primitive tiles x and y.

In general, these two operators are neither equivalent nor commutative. However, it is an easy observation that, in the case x and y have the same duration, then, thanks to parallel commutativity axiom, the parallel fork and the parallel join are equivalent commutative operators.

6.3 Stretching product

Though this departs from standard inverse semi-group theory, the previous observation allows us to lift the product over duration to a product over tiles. We can define a stretch operation:

$$stretchT :: d \to Tile\ d\ iv\ v \to Tile\ d\ iv\ v$$

that just amounts to stretch, that is, to multiply by a given factor, all durations in a tile. Handing in a distinct way the case of positive or negative duration, this function can be defined in such a way that, for every *positive* duration d:

$$stretchT\ d\ (-x) == -(stretchT\ d\ x)$$

and

$$stretchT\ d\ (x + y) == stretchT\ d\ x + stretch\ d\ y$$

in the case d is positive with $|stretchT\ d\ x| == d{*}|x|$. Then, we define the product of two tiles by

$$(*) :: (Num\ d) \Rightarrow Tile\ d\ iv\ v \to Tile\ d\ iv\ v \to Tile\ d\ iv\ v$$
$$(*)\ x\ y = parForkT\ (stretchT \mid x \mid y)\ (stretchT \mid y \mid x)$$

which turned out to be commutative. An example of product over primitive tile is depicted in Figure 10. In this figure, with a bit of type abuse, we write $d * x$

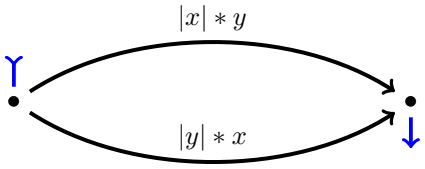

Figure 10: The product of two primitive tiles x and y.

the stretch of a temporal x by d units of time. Clearly, the product of tile offers an alternative for the definition of parallel compositions of tiles.

6.4 Num and Fractional instances

Function *fromDurationT* is a one-to-one morphism the additive group d defined by durations and the additive group formed by delays. It is a faithful embedding of the types *Integer* and *Rational* into the type of polymorphic temporal tiles $Tile\ d\ iv\ v$. Together with sum, negation and product, this leads to the instance

$$\textbf{instance}\ (Num\ d) \Rightarrow Num\ (Tile\ d\ iv\ v)$$

with ad hoc *abs* and *signum* functions respectively defined by $abs = const\ 1$ and $signum = id$.

Quite similarly, when $|x|$ is non zero, we can define the multiplicative inverse by

$$recipT :: (Fractional\ d) \Rightarrow Tile\ d\ iv\ v \to Tile\ d\ iv\ v$$
$$recipT\ x = stretchT\ (1/ \mid x \mid)\ (stretchT\ (1/ \mid x \mid)\ x)$$

which eventually leads the instance

$$\textbf{instance}\ (\mathit{Fractional}\ d) \Rightarrow \mathit{Fractional}\ (\mathit{Tile}\ d\ \mathit{iv}\ v)$$

Again, embedding of arbitrary rational constant into tiles is done via the function $\mathit{fromDurationT}$.

6.5 Concluding example

The instances of *Num* and *Fractional* offer us a convenient syntax over tiles. Most functions over queue lists such as atomsQ, $\mathit{delayToTailQ}$, tailQ or $\mathit{stretchQ}$, as well as functions deriving from category theoretic properties, have straightforward lift to function over tiles. Moreover, functions over tiles can easily be converted back into functions over queue lists.

It follows that functions over temporal tiles can also be used for specifying reactive system behaviors. The expressive power and the succinctness of the resulting DSL is illustrated by the following final example.

Based on sort of a *comonadic* property, we first define the function $\mathit{coJoinT}$ that lifts every input tile into a tile of tiles.

$$\begin{array}{l}
\mathit{coJoinT} :: \mathit{Tile}\ d\ \mathit{iv}\ v \to \mathit{Tile}\ d\ \mathit{iv}\ (\mathit{Tile}\ d\ \mathit{iv}\ v) \\
\mathit{coJoinT}\ (\mathit{Tile}\ d\ _\ \mathit{QEnd}) = \mathit{Tile}\ d\ 0\ \mathit{QEnd} \\
\mathit{coJoinT}\ t \\
\quad = \textbf{let}\ \mathit{liftedAtoms} = \mathit{fmap} \\
\qquad\qquad (\lambda(d, v) \to (\mathit{fromDurationAndValueT}\ 1 \\
\qquad\qquad\qquad\qquad (\mathit{fromDurationAndValueT}\ d\ v)) \\
\qquad\qquad (\mathit{atomsT}\ t) \\
\quad\ \textbf{in}\ \mathrm{re}\ \mathit{liftedAtoms} + \mathit{delayToTailT}\ t \\
\qquad + \mathit{recT}\ \mathit{coJoinT} \circ \mathit{tailT}\ t
\end{array}$$

Dually, based on sort of a *monadic* property, we also define the function joinT that turns back every input tile of tiles into a tile.

$$\begin{array}{l}
\mathit{joinT} :: \mathit{Tile}\ d\ \mathit{iv}\ (\mathit{Tile}\ d\ \mathit{iv}\ v) \to \mathit{Tile}\ d\ \mathit{iv}\ v \\
\mathit{joinT}\ (\mathit{Tile}\ d\ _\ \mathit{QEnd}) = \mathit{Tile}\ d\ 0\ \mathit{QEnd} \\
\mathit{joinT}\ t \\
\quad = \textbf{let}\ \mathit{tileAtoms} = \mathit{fmap} \\
\qquad\qquad (\lambda(d, \mathit{ta}) \to \mathit{stretchT}\ d\ \mathit{ta})\ (\mathit{atomsT}\ t) \\
\quad\ \textbf{in}\ \mathrm{re}\ \mathit{tileAtoms} + \mathit{delayToTailT}\ t \\
\qquad + \mathit{recT}\ \mathit{joinT} \circ \mathit{tailT}\ t
\end{array}$$

Then, we can define the function over tiles:

$$\begin{array}{l}
\mathit{finalEx} :: \mathit{Tile}\ d\ \mathit{iv}\ v \to \mathit{Tile}\ d\ \mathit{iv}\ v \\
\mathit{finalEx}\ t = \mathit{joinT} \circ \mathit{fmapT}\ (\lambda x \to x + 2 * x) \circ \mathit{coJoinT}\ t
\end{array}$$

whose behavior, when applied to a flow of notes embedded into an input temporal tile, is the following:

> every input note is passed through to the output and then repeated with a double duration.

Indeed, thanks to *functorial* properties, via $fmapT$, the function $\lambda x \rightarrow x + 2 * x$ copies and repeats every single (tile lifted) note just as expected.

To the best of our knowledge, no other existing programing language offers such a concise way to describe such a music system behavior.

7 Related work

The idea of using tiles as models for music goes back at least to the language LOCO [5] although the induced algebra were first observed by the second author [16, 3]. Some experiments of controlling music systems based on tile specifications have been made [17, 1]. Although the principle of tiled programming has already been sketched earlier [18] and experimented over Euterpea [14], the implementation presented here is the first fully featured polymorphic and reactive one implemented to far.

Our proposal share clear goals with functional reactive programming (FRP) approaches [6, 7]. It is reactive in the sense that programs may react on the start (or the stop) of input temporal values. It is also weakly real-time in the sense that durations of temporal values, automatically computed on-the-fly, are also embedded as first class citizens. More precisely, durations (or rather delays) are automatically interpreted as timers. Delays can thus be used to schedule outputs at known time.

However, by embedding (possibly continuous time dependent) temporal values between well-bracketed events, instead of embedding events into piecewise continuous behavior, the resulting types are significantly distinct. This is especially clear in view of the algebraic layer that, defined over queue lists, eventually leads to the definition of temporal tiles.

In terms of programming usability, the notion of tile modeling and tile programming is in its infancy. To the best of our knowledge, as illustrated by generalizations of our last example, no available real-time and reactive language yet offers the same degree of succinctness.

However, despite such an apparent success, we need many more usage experiments before achieving a fair evaluation of the benefits and the drawback of our model-driven proposal, especially compared to somewhat more classical and more robust event-based approaches (see e.g. [21, 20, 22]). Additional experiments also suggest that a language like Reactive ML [20] could offer a simpler embedding language thanks to its fairly generic multi-process reactive kernel.

8 Conclusion

We have defined the notion of queue lists, from which we have derived both a reactive kernel, translating back and forth queue lists into streams of events, and the temporal tiles front-end with rather robust algebraic properties. The resulting Domain Specific Language (DSL), we called the *T-calculus*[10], is embedded into Haskell.

The event-based layer has been built over the Euperpea/UISF libraries[12, 23] for concrete experiments in computational music. Non trivial music reactive

[10] see `http://poset.labri.fr/tcalculus/`

systems have been realized as programming examples[11]. However, we expect our DSL to be applicable to many other application fields, especially those involving temporal media [11].

Of course, the notion of temporal causality (yet quite implicit) arising from defining functions over temporal tiles needs to be understood much more in the depth. This is the purpose of an ongoing study.

Acknowledgement

This work is dedicated to the memory of Paul Hudak who showed us the way for bridging the gap between clean mathematical models and cool reactive music systems via pure typed functional programming.

References

[1] S. Archipoff. An efficient implementation of tiled polymorphic temporal media. In *Work. on Functional Art, Music, Modeling and Design (FARM)*, pages 25–34. ACM, 2015.

[2] A. Asperti and G. Longo. *Categories, types and structures - an introduction to category theory for the working computer scientist.* Foundations of computing. MIT Press, 1991.

[3] F. Berthaut, D. Janin, and B. Martin. Advanced synchronization of audio or symbolic musical patterns: an algebraic approach. *International Journal of Semantic Computing*, 6(4):409–427, 2012.

[4] K. Claessen and J. Hugues. QuickCheck: a lightweight tool for random testing of haskell programs. In *Int. Conf. Functional Programing (ICFP)*, 2000.

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

[6] C. Elliott and P. Hudak. Functional reactive animation. In *Int. Conf. Functional Programing (ICFP)*. ACM, 1997.

[7] C. M. Elliott. Push-pull functional reactive programming. In *Symp. on Haskell*, pages 25–36. ACM, 2009.

[8] J. Gibbons and N. Wu. Folding domain-specific languages: deep and shallow embeddings (functional pearl). In *Int. Conf. Functional Programing (ICFP)*, pages 339–347, 2014.

[9] P. Hudak. Keynote address - the promise of domain-specific languages. In *Proceedings of the Conference on Domain-Specific Languages (DSL)*, 1997.

[10] 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, 2004.

[11] see `http://poset.labri.fr/interpolations/`

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

[12] P. Hudak. *The Haskell School of Music : From signals to Symphonies*. Yale University, Department of Computer Science, 2013.

[13] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*. ACM Press, 2007.

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

[15] P. Hudak and D. Janin. From out-of-time design to in-time production of temporal media. Research report, LaBRI, Université de Bordeaux, 2015.

[16] D. Janin. Vers une modélisation combinatoire des structures rythmiques simples de la musique. *Revue Francophone d'Informatique Musicale (RFIM)*, 2, 2012.

[17] D. Janin, F. Berthaut, and M. Desainte-Catherine. Multi-scale design of interactive music systems : the libTuiles experiment. In *Sound and Music Comp. (SMC)*, 2013.

[18] 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 *Work. on Functional Art, Music, Modeling and Design (FARM)*, pages 23–34. ACM Press, 2013.

[19] M. V. Lawson. *Inverse Semigroups : The theory of partial symmetries*. World Scientific, 1998.

[20] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2005.

[21] H. Nilsson, J. Peterson, and P. Hudak. Functional hybrid modeling. In *Int. Symp. On Practical Aspects of Declarative Languages (PADL)*, pages 376–390, 2003.

[22] I. Perez and H. Nilsson. Bridging the GUI gap with reactive values and relations. In *Symp. on Haskell*, pages 47–58. ACM, 2015.

[23] D. Winograd-Cort. *Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming*. PhD thesis, Yale University, December 2015.

[24] D. Winograd-Cort and P. Hudak. Settable and non-interfering signal functions for FRP: how a first-order switch is more than enough. In *Int. Conf. Functional Programing (ICFP)*, pages 213–225. ACM, 2014.