The T-Calculus : towards a structured programing of (musical) time and space

David Janin, Florent Berthaud, Myriam Desainte-Catherine, Yann Orlarey, Sylvain Salvati

LaBRI, CNRS UMR 5800
Laboratoire Bordelais de Recherche en Informatique

Rapport de recherche RR-1466-13

The T-Calculus : towards a structured programing of (musical) time and space

June 19, 2013

David Janin[1], Florent Berthaut[1], Myriam DeSainte-Catherine[1],
Yann Orlarey[2], Sylvain Salvati[1,3]

[1] Université de Bordeaux
LaBRI UMR 5800
351, cours de la libération
F-33405 Talence, FRANCE

[2] GRAME
Centre Nat. de Création Musicale
11, Cours de Verdun
F-69002 Lyon, FRANCE

[3] Inria Bordeaux - Sud-Ouest
200, avenue de la Vieille Tour
F-33405 Talence, FRANCE

Contents

The T-Calculus : towards a structured programing of (musical) time and space

David Janin[1]*, Florent Berthaut[1], Myriam DeSainte-Catherine[1], Yann Orlarey[2], Sylvain Salvati[1,3]

[1] Université de Bordeaux
LaBRI UMR 5800
351, cours de la libération
F-33405 Talence, FRANCE

[2] GRAME
Centre Nat. de Création Musicale
11, Cours de Verdun
F-69002 Lyon, FRANCE

[3] Inria Bordeaux - Sud-Ouest
200, avenue de la Vieille Tour
F-33405 Talence, FRANCE
Corresp. author: janin@labri.fr

June 19, 2013

Abstract

In the field of music system programming, the T-calculus is a proposal for combining space modeling and time programming into a single programming feature: spatiotemporal tiled programming. Based on a solid algebraic model, it aims at decomposing every operation on musical objects into the sequence of a synchronization operation that describes how objects are positioned one with respect the other, and a fusion operation that describes how their values are then combined. A first version of that calculus is presented and studied in this paper.

1 Introduction

With the development of intermedia devices, there is a need for programming complex interactive musical systems. Numbers of tools, norms, languages and design principles, more or less specific to such a task, can be used. Lots of outstanding musical applications have already been and still are successfully developed.

*partially funded by the project INEDIT, ANR-12-CORD-009

The need for developing even more complex system is however still growing. The available tools, norms, languages and design principles can thus still be developed for that purpose.

Beyond the apparent entertaining nature of musical applications, the programming of a musical piece, be it out of time for musical composition, or in time for musical performance, is also at least as complex as programming any time-sensitive system in another application field. It follows that advances in music programming may also impact programming in other application fields.

Every problem encountered in music system programming deserves thus full attention and is worth being solved in the most versatile way.

1.1 The complex structure space of music

The structure of music is complex. Being polyphonic, it is structured in some musical space (P). Being rhythmic, it is structured in some musical time (T). It also combines various levels of abstraction (A), from low level audio signal processing to high level concert movements combination, upon which may depend the space and time scales. It can even be interactive (I), that is, subject to changes depending on the input of several independent (or loosely coupled) musical sources.

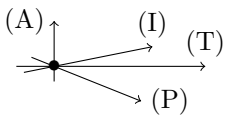

Figure 1: The 4D structure space of music

This observation leads to the development of application specific languages, dedicated to music programming, that are based on generic and well-defined programming language paradigms. For instance, the versatile DSP programming language *Faust* [6] is based on the synchronous programming paradigm of languages such as Lustre or Esterel.

Another example, the domain-specific language *Euterpea* [7] is embedded in the typed functional language *Haskell* [8].

These languages provide strong abstraction design principles that can be used efficiently when programming musical applications. Using such mathematically well-defined programming languages increases both the efficiency of the development process and the reliability of the developed application or system.

1.2 Modeling space and programming time

In this paper we aim at contributing to the development of these languages. We are more specifically concerned with the following problem: coping with the combined programming of musical time and space.

It is common observation that music writing involve both sequential composition (in time) and parallel composition (in space) of musical objects.

A classical programming of these two features will often amount to:

- ▷ *space modeling (data structure)* : creating a (finite) vector of musical objects to be played in parallel, with one local player per musical objects,
- ▷ *time programming (control flow)* : creating a (potentially infinite and evolving) list of musical objects to be played in sequence, with a single global player for that list.

From a strict logical point of view, this is however *not* a necessity.

A priori, from an abstract point of view, there is no need to distinguish between spatial dimensions that describe parallelism and time dimensions that describe execution flow. These are just dimensions that create together the spatiotemporal space into which music evolves.

A posteriori, some modeling experiments [10, 1] show that, at least in the simpler problem of synchronizing and mixing audio or musical sequences, there *are* some mathematically well-founded models [15, 16] that allow for the description of both spatial and temporal combinations of these sequences with a *single* operator, the tiled product, that encompasses both sequential and parallel composition operators.

This gives hints that, when aiming at programming music systems, that handle objects defined in some complex spatiotemporal space, the classical programming distinction between data structure (space) and control structure (time), though quite healthy in many programming context, is not mandatory.

1.3 Towards spatiotemporal modelo-programming

Our purpose is to examine to which extent and for what benefit the tiled product, defined in [15] for crystal structure analysis and tuned for application to music synchronization in [1], can be integrated in a programing language as a buit-in programming feature.

More precisely, every musical object is seen as a *partial* function $m : \mathcal{S} \to \mathcal{V}$ from some spatiotemporal space $\mathcal{S}$ into some set $\mathcal{V}$ of musical values. The domain $dom(m) \subseteq \mathcal{S}$ of such a musical object m describes the *spatiotemporal structure* of that object and, for every $x \in dom(m)$, the value $m(x) \in \mathcal{V}$ describes the local *musical state* of that object at position x.

Then, given two musical object $m_1 : \mathcal{S} \to \mathcal{V}_1$ and $m_2 : \mathcal{S} \to \mathcal{V}_2$, every operation on these musical objects that aims at defining a new musical object $F(m_1, m_2) : \mathcal{S} \to \mathcal{V}_3$ is decomposed as a sequence of two primitive operations:

- ▷ *synchronization:* defining the domain $dom(F(m_1, m_2))$ of the resulting musical object as some generic combination $dom(m_1) \oplus dom(m_2)$ that tells how the domains $dom(m_1)$ and $dom(m_2)$ are translated to be positioned one relative to the other with possible overlaps,

- ▷ *fusion:* defining how the new musical states are defined on each position of the resulting domain from the local musical states of the translated musical objects m_1 and m_2.

Doing so, we somehow aim at generalizing to spatiotemporal structures the notion of spatial programming that is already emerging in bioinformatics system modeling or musical structure analysis [20, 2, 3].

2 Strings and streams in music programming

In this section, we review some of the basic features of the strings and streams data types. Then, by taking some examples from music modeling, we show how strings and streams fail to satisfy some compositional properties one may expect when modeling music.

From now on, let $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{D}_2$, etc., be some alphabet types. That is, elements of a type $\mathcal{D}$ will compose the strings and streams we will handle.

2.1 Strings in music programming

Strings (or equivalent data types) are probably one of the basic data-structure one may use in music programming as shown for instance in the *libAudioStream* [19]. Since we propose below to extend this data type, let us first review our notation for strings.

By $\mathcal{D}$-strings we mean here any finite (possibly empty) list of elements of type $\mathcal{D}$. A $\mathcal{D}$-string is thus a mapping

$$m : [0, |m| - 1] \to \mathcal{D}$$

where $|m| \in \mathbb{N}$ is the length of the string. The main operation on strings is the concatenation product (denoted by $\cdot$). For every two $\mathcal{D}$-strings m_1 and m_2, the concatenation product

$$(m_1 \cdot m_2) : [0, |m_1| + |m_2| - 1] \to \mathcal{D}$$

of the two strings m_1 and m_2 is defined, for every $k \in [0, |m_1| + |m_2| - 1]$ by

$$(m_1 \cdot m_2)(k) = \begin{cases} m_1(k) & \text{if } 0 \leq k < |m_1| \\ m_2(k - |m_1|) & \text{if } |m_1| \leq k < |m_1| + |m_2| \end{cases}$$

Example 2.1 Let us now turn to music programming with strings. For that purpose, assume that we want to encode the following bebop tune, depicted in Figure 2, that have already been considered in [10]. A simple analysis of the musical structure of that song shows that it is composed with three repetition of a first melodic line, depicted in Figure 3, that is followed by a second melodic line, depicted in Figure 4. Then, given a data type $\mathcal{D}$ that simply corresponds

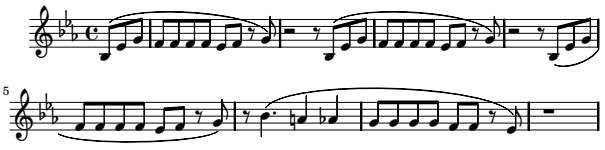

Figure 2: *My little suede shoes* (C. Parker, 1951)

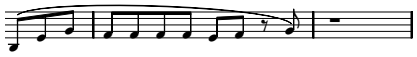

Figure 3: A first melodic line

notes and silences of duration one eight, we can encode these melodic lines as two $\mathcal{D}$-strings m_1 and m_2 with respective length $|m_1| = 11$ and $|m_2| = 15$.

Of course, for the second melodic line, we assume that there is some special symbol in $\mathcal{D}$ that merely means "keep on playing the same note" in order to encode the quarter notes and the dotted quarter note.

The entire melodic line that is depicted in Figure 2 is then encoded by the string

$$m = m_1 \cdot cs_5 \cdot m_1 \cdot cs_5 \cdot m_1 \cdot cs_1 \cdot m_2$$

where the constant strings cs_ks describe strings of silences of length k. The structure of such a concatenation can be depicted as in Figure 5.

An immediate remark is that, in this concatenation example of the first and second melodic lines, we have to insert silences of various length. Doing so, the general structure of the entire melodic line, stated as "three times melody m_1 followed by melody m_2" is a little lost.

Of course, one can directly include the silences in the encoding of the melodic line themselves, as actually written in Figures 3 and 4. But then, one need two encodings of the first melodic line in m. We need one encoding for its first two occurrences ($m_1 \cdot cs_5$) and another one for its last occurrence ($m_1 \cdot cs_1$).

Continuing our example, if we want to play twice that melody, then we have to construct the melodic line

$$mm = m \cdot cs_5 \cdot m$$

This encoding is definitely lacking musical sense. Indeed, no musician will ever pay attention that 5 eighth silences are needed here. As we shall see from Section 3.1, we can actually extend the string type to encode what musicians actually do: they synchronize on music bars.

In other words, all proposed encodings of the melodic lines miss part of the musical meaning of these melodic lines. They are not faithfully encoding the musical reality we aim at capturing. In some sense, *an expected compositionality property cannot be satisfied by the concatenation product on strings.* Depending on the melodic line to be concatenated, the amount of silences to be inserted is changing as illustrated by the examples.

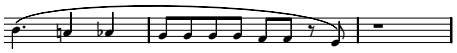

Figure 4: A second melodic line

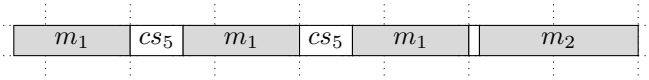

Figure 5: The structure of the complete melodic line

Remark 2.2 *In musical terms, as already observed in [10], both the first and the second melodies are built upon a notion of* anacrusis *that creates a distinction between the effective start of the melodic line: the beginning of its first note, from the logical start of the melodic line: the first bar of the examples.*

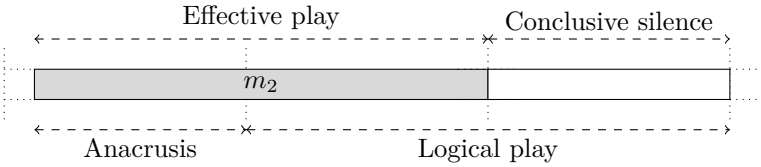

Figure 6: The musical structure of the second melodic line

The same examples can be used to show that similar phenomena occur at the end of melodic lines. Indeed, the logical ends of both melodic lines are actually given by the last bars depicted in Figure 3 and Figure 4 while the concrete ends occur before.

This is especially clear in the second melodic line where, as often in bebop style, the last note is played right before the bar in order to emphasis even more on the next strong beat, though silent, that concludes the underlying rhythm. The situation for the second melodic line is depicted in Figure 6.

2.2 Streams in music programming

Let us now consider streams in music. Streams are (potentially) infinite sequences of values often used to model audio signals as in *Faust* [6] or even arbitrarily abstract sequences as in *Haskell* [8]. Again, as we propose below to extend this data type, let us first review our notation for streams.

By $\mathcal{D}$-streams we mean here an infinite list of elements of some type $\mathcal{D}$. A $\mathcal{D}$-stream is thus a mapping $s : \mathbb{N} \to \mathcal{D}$. The main operation on streams is the parallel product. For every $\mathcal{D}_1$-stream s_1 and $\mathcal{D}_2$-string s_2, the parallel product

$$s_1|s_2 : \mathbb{N} \to \mathcal{D}_1 \times \mathcal{D}_2$$

of the two streams s_1 and s_2 is defined, for every $k \in \mathbb{N}$, by

$$(s_1|s_2)(k) = (s_1(k), s_2(k))$$

Observe that with such a definition we have, up to isomorphisms, $(s_1|s_2) = (s_2|s_1)$, i.e. the parallel product is commutative.

With both strings and streams, we may also assume that we have an additional operation (denoted by ::) that take one $\mathcal{D}$-string m and one $\mathcal{D}$-stream s as inputs and that produces the $\mathcal{D}$-stream $m :: s$ obtained by placing the string m in front of the stream s, i.e. the stream $m :: s$ is defined, for every $k \in \mathbb{N}$, by

$$(m :: s)(k) = \begin{cases} m(k) & \text{if } 0 \leq k < |m| \\ s(k - |m|) & \text{if } |m| \leq k \end{cases}$$

Example 2.3 (2.1 continued) Going back to our musical example, we assume that we have a constant $\mathcal{D}$-stream $null_{\mathcal{D}}$ that is defined as the silent stream that is, for every $k \in \mathbb{N}$, we have $null_{\mathcal{D}}(k) = 0_{\mathcal{D}}$ for some elementary silence $0_{\mathcal{D}}$. Then, the operation defined above allows for converting the entire melody m in Figure 2 into a $\mathcal{D}$-stream $s : \mathbb{N} \to \mathcal{D}$ by letting $s = m :: null_{\mathcal{D}}$.

Assuming now we want to play in parallel (or rather to describe the structure of such a play) the musical stream s with another stream of music $s' : \mathbb{N} \to \mathcal{D}$ encoding, say, some accompanying bass line. For simplicity, we assume that these two musical lines are encoded on the same time scale, say one (symbolic) time click per eighth note.

A priori, the parallel plays are defined as $p = s|s'$. However, for reasons similar to the string case examples, such a product is not satisfactory because it assumes that the two streams actually start at the same time. It may be the case, for instance, that the stream s' of the bass line starts right after the first bar. In that case, what we want to build is defined as the parallel streams

$$p = s|(cs_3 :: s')$$

the length of the silence cs_3 being defined by some comparison between the musical nature of s and s'. Such a situation is depicted in Figure 7.

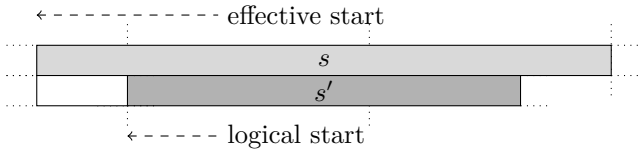

Figure 7: On time starts of parallel musical plays

Assuming instead that the bass starts one measure before, the play is rather defined as $p = (cs_5 :: s)|s'$.

Again, we fail to capture the musical nature of the melodic lines we aim at encoding. *Some expected compositionality law that would guarantee the preservation of the musical intention we have in our encoding of melodic lines is violated.* Indeed, with such an encoding the intended parallel composition depends on the properties of the composed streams s and s': it is not uniformly defined as one would expect with a more robust data-type *melody*.

Remark 2.4 *A programmer with some knowledge in music will immediately propose to solve both problems encountered in Section 2.1 and here in Section 2.2 by encoding, within note sequences, be them finite strings or infinite streams, the musical bars. Such a programmer would indeed be right. This exactly what we will do.*

However, we also aim at keeping only the features we really need in these musical bars – only a selection of the musical bars that are truly needed for temporal positioning – and we also aim at defining these remaining features as a built-in extension of the data types and not as a somehow adhoc encoding that may later fail to behave nicely.

It occurs that this can be done in an accurate way via the notion of tiled streams presented in the next section.

3 From strings and streams to tiled streams

The examples studied in the previous section give incentives for merging these two data types into a more general one: tiled streams with synchronized products, that is presented in this section.

3.1 Tiled streams for music programming

Let $\mathcal{D}$ be an element type. We assume that $\mathcal{D}$ is equipped with a distinguished element $0_{\mathcal{D}}$ that acts as a default undefined value.

A tiled $\mathcal{D}$-stream is a pair $\langle t, d\rangle$ where t is a bi-infinite mapping $t : \mathbb{Z} \to \mathcal{D}$ and $d \in \mathbb{N}$ is a natural number called the synchronization duration of t. In order to keep notation simple, we just write t for such a $\mathcal{D}$-stream and we write $d(t) \in \mathbb{N}$ for its synchronization duration.

Of course, we may assume that such tiled streams have a default value (silence in music) outside some effective realization interval. Such a tiled streams with finite (or at least left finite) effective interval is depicted in Figure 8. All examples pictured below assume such a finite realization interval.

In other words, following the analogy with musical notation, a tiled stream is a (bi-infinite) stream with to bars: the first one that indicates the logical beginning of the music, the second one that indicates the logical end of the music.

Synchronizing two tiled streams one "after" the other is then defined by means of the tiled product defined as follows.

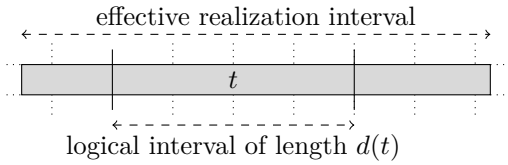

Figure 8: A typical tiled stream with anticipated start and late end

For every tiled $\mathcal{D}_1$-stream t_1 and $\mathcal{D}_2$-string t_2, the tiled product $t_1 ; t_2$ of the two tiled streams t_1 and t_2 is defined, for every $k \in \mathbb{N}$, by

$$(t_1 ; t_2)(k) = (t_1(k), t_2(k - d(t_1)))$$

with the synchronization duration $d(t_1 ; t_2)$ of the product defined by

$$d(t_1 ; t_2) = d(t_1) + d(t_2)$$

It is an easy exercise to check that, up to isomorphism, this product is associative.

The product of two tiled streams t_1 and t_2 is depicted in Figure 9.

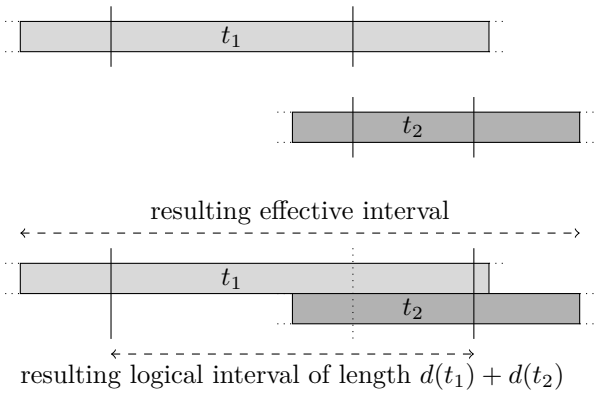

Figure 9: The tiled synchronization product of t_1 and t_2

3.2 From musical strings to musical tiled streams

Going back to Example 2.1, let m_1 and m_2 be the $\mathcal{D}$-string defined in Section 2.1.

We encode the first melodic line in Figure 3 by the tiled stream t_1 defined, for every $k \in \mathbb{Z}$, by

$$t_1(k) = \begin{cases} m_1(k+3) & \text{if } -3 \leq k < |m_1| - 3 \\ 0_{\mathcal{D}} & \text{(otherwise)} \end{cases}$$

with sync. duration $d(t_1) = 16$, and we encode the second melodic line in Figure 4 by the tiled stream t_2 defined, for every $k \in \mathbb{Z}$, by

$$t_2(k) = \begin{cases} m_2(k+7) & \text{if } -7 \leq k < |m_2| - 7 \\ 0_{\mathcal{D}} & \text{otherwise} \end{cases}$$

with sync. duration $d(t_2) = 16$. Then, defining the entire melodic line in Figure 2 just amount to perform the tiled products:

$$t = t_1 \,;t_1 \,;t_1 \,;t_2$$

that simply encode our initial musical analysis, i.e. three times the first melodic line followed by one time the second melodic line.

Remark 3.1 *Of course, the resulting model has four parallel voices. Indeed, we need to merge all these voices into a single one, possibly, in the general cases, by allowing chords instead of single notes. This will be done by the merge operation that has been mentioned in the introduction. Thus the encoding of our complete (tiled) melody is rather defined to be merge*$(t_1 \,;t_1 \,;t_1 \,;t_2)$ *or even, as detailled in Section 4.3 below (see Example 4.9), just defined to be* $t_1 + t_1 + t_1 + t_2$.

In other words, we have defined a model that solves the lack of compositionality raised by modeling music as strings as in Section 2.1.

3.3 From musical streams to musical tiled streams

We now aim at showing that the compositionality problem raised by modeling music as streams as in Section 2.2 can also be solved similarly. In order to do so, we need to extend a little further the tiled stream data type with two additional operators on tiled streams that amount to reset the synchronization length to zero.

More precisely:

Definition 3.2 (Tiled stream reset and co-reset) *For every tiled $\mathcal{D}$-stream t, we define the synchronization reset $R(t)$ and the synchronization co-reset $L(t)$ to be the tiled $\mathcal{D}$-streams defined, for every $k \in \mathbb{Z}$ by*

$$(R(t))(k) = t(k) \text{ and } (L(t))(k) = t(k - d(t))$$

with $d(R(t)) = d(L(t)) = 0$.

The reset and co-reset operators are depicted in Figure 10.

Going back to Example 2.1, given the tiled stream t defined in Section 2.2 that encode the melodic line in Figure 2, given another tiled stream t' that model some base line. Assuming that $t'(0)$ models the first eighth note or silence of the first bar of that bass line, then, the parallel play p of both the melodic line and the bass line is simply modeled by

$$p = R(t) \,; R(t')$$

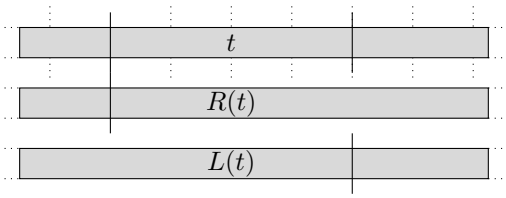

Figure 10: Reset $R(t)$ and co-reset $L(t)$ of a tiled stream t.

In other words, since the logical start of the lead melody and the bass line are encoded in their respective tiled stream encodings, the parallel product of the two just amount to synchronize these two starts. This is easily done as above.

Doing so, we have solved the compositionality problem raised by modeling as streams as in Section 2.2.

Remark 3.3 *More generally, with the proposed model of tiled streams, we eventually internalize in each model the synchronization information that is needed to position in time two melodic lines, be them put in sequence, as with strings, or in parallel as with streams. Doing so, the tiled product is a composition operators that has both features of a sequential product and of a parallel product. This observation is made formal in the next section.*

3.4 Strings and streams embeddings

From a more theoretical point of view, we show that both strings and streams can be embedded into tiled streams.

For strings, let φ be the mapping that maps every $\mathcal{D}$-string m to the tiled $\mathcal{D}$-stream $\varphi(m)$ defined, for every $k \in \mathbb{Z}$ by

$$(\varphi(m))(k) = \begin{cases} m(k) & \text{if } 0 \leq k < |m| \\ 0_{\mathcal{D}} & \text{otherwise} \end{cases}$$

with $d(\varphi(m)) = |m|$, we also have:

Lemma 3.4 (Strings embeddings) *The transformation φ from $\mathcal{D}$-strings to tiled $\mathcal{D}$-streams is a one-to-one homomorphism that maps string concatenation to (merged) tiled product, i.e. for every $\mathcal{D}$-strings m_1 and m_2 we have*

$$\varphi(m_1 \cdot m_2) = \mathit{merge}(\varphi(m_1)\,;\varphi(m_2))$$

From a theoretical point of view, given ψ the mapping that maps every $\mathcal{D}$-stream s to the tiled $\mathcal{D}$-stream $\psi(s)$ defined, for every $k \in \mathbb{Z}$ by

$$(\psi(m))(k) = \begin{cases} s(k) & \text{if } 0 \leq k \\ 0_{\mathcal{D}} & \text{otherwise} \end{cases}$$

with $d(\varphi(s)) = 0$, we also have:

Lemma 3.5 (Streams embeddings) *The mapping ψ from $\mathcal{D}$-streams to tiled $\mathcal{D}$-streams is a one-to-one homomorphism that maps stream parallel product to tiled product, i.e. for every $\mathcal{D}$-stream s_1 and s_2 we have*

$$\psi(s_1|s_2) = \psi(s_1)\,;\psi(s_2)$$

In particular, when restricted to tiled streams with zero synchronization length, the tiled stream product is, up to isomorphism, commutative.

We observe also that these embeddings of strings and streams into tiled streams also preserve mixed products stated below:

Lemma 3.6 (Mixed embeddings) *For every $\mathcal{D}$-string m and $\mathcal{D}$-stream s, we have*

$$\psi(m :: s) = R(merge(\varphi(m)\,;\psi(s)))$$

i.e. the mixed product of strings with streams is also preserved by the homomorphisms φ and ψ.

To make the picture complete, let us conclude this section by showing how the (stream like) parallel product defined on tiled stream with zero sync. length generalizes to arbitrary tiled streams. More precisely, combination of tiled product and resets lead to the definition of *fork* and *join* parallel products defined bellow.

Definition 3.7 (Fork and join derived operators) *For every two tiled streams t_1 and t_2 let $fork(t_1, t_2)$ and $join(t_1, t_2)$ be the products defined by:*

- ▷ *Parallel fork: $fork(t_1, t_2) = R(t_1)\,;t_2$, i.e. the logical intervals start at the same time,*
- ▷ *Parallel join: $join(t_1, t_2) = t_1\,;L(t_2)$, i.e. the logical intervals end at the same time.*

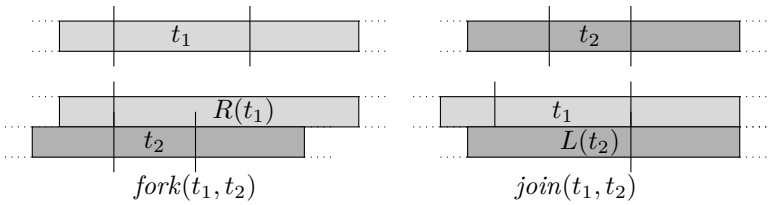

Figure 11: Derived *fork* and *join* operators.

These two derived products are illustrated in Figure 11. One can observe that the tiled stream $fork(t_1, t_2)$ may be distinct from the tiled stream $fork(t_2, t_1)$ even up to isomorphism since in the first case it inherits from the logical interval of t_2 while in the second case it inherits from the logical interval of t_1. A similar remark holds for $join(t_1, t_2)$ and $join(t_2, t_1)$.

4 Static (out of time) T-calculus

We present here a static version of the T-calculus, that is, the T-calculus with no I/O mechanisms. Effectivity issues (how to compute T-programs) are discussed in Section 5.

4.1 Syntax

Basic constants and types. We assume that there is some set of basic types such as booleans: `bool`, positive integers: `natural`, or sets of events: `eventSet(E)` for some (finite) predefined set of events `E`, etc.

These types are uniformly seen as *semiring* structures with (infix) associative sum `+`, e.g. union for event sets, (infix) associative product `*` that distributes over sum, e.g. intersection for event sets, with a zero `0` that is neutral for sum and absorbant for product, e.g. empty set for even sets, and a unit `1` that is neutral for product, e.g. the set `E` of all events for event sets.

They are associated with constants. In particular, we use the notation `{a,b,c}` for the set of three events `a`, `b` and `c` $\in$ `E`. To keep notations simple, we drop any subscript (or any other mark) that may refer to the type of these constants and operators though they are implicitly considered as unambiguously typed.

For convenience, we may also assume that these types are equipped with heterogeneous mappings such as, for instance, from pairs of event sets to booleans, event sets equality test `==` or inclusion $<$.

Programs. A T-calculus program p is just a term built by means of the program constructs described in Figure 12 where `c` is a constant, `x` is a variable, `f` is a function symbol and p_1, p_2, ..., p_i are syntactically simpler programs.

p ::=		– *primitive constructs* –
	`c`	(constant)
|	`x`	(variable)
|	`f(`$p_1, p_2, \cdots, p_n$`)`	(mapping)
|	`x=`p_1	(assignment)
|	`R(`p_1`)`	(sync. reset)
|	`L(`p_1`)`	(sync. co-reset)
		– *derived constructs* –
|	p_1 `op` p_2	(operator)
|	p_1 `;` p_2	(sync. product)

Figure 12: Static T-calculus syntax

The last two constructs, operators and synchronization product, derive from the others and will thus be treated as such.

Remark 4.1 *Since there is no notion of variable scope, we also assume that* for every program p, for every variable $\mathtt{x}$ occurring in p, there is at most one subprogram of p of the form $\mathtt{x} = p_{\mathtt{x}}$.

4.2 Types

Every T-calculus program will be interpreted as a tiled $\mathcal{D}$-stream also called α-stream when α is the type of elements of $\mathcal{D}$. As there shall be no surprise with the typing of the elements of tiled streams, we concentrate on the typing of tiled streams resulting from T-calculus programs. The typing relation is defined in Figure 13. More precisely, for every T-program p, we define the typing relation

▷ Constants:	$\dfrac{}{\Gamma \vdash \mathtt{c} : (1, \alpha_c)}$
▷ Variables:	$\dfrac{(\mathtt{x}, (d, \alpha)) \in \Gamma}{\Gamma \vdash \mathtt{x} : (d, \alpha)}$
▷ Mapping:	$\dfrac{\Gamma \vdash p_i : (d_i, \alpha_i) \quad (i \in [1, n])}{\Gamma \vdash \mathtt{f}(p_1, \cdots, p_n) : (d_1 + \cdots + d_n, \alpha)}$ with $\mathtt{f} : \alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \alpha$
▷ Assignment:	$\dfrac{\Gamma \vdash \mathtt{x} : (d, \alpha) \quad \Gamma \vdash p : (d, \alpha)}{\Gamma \vdash \mathtt{x} = p : (d, \alpha)}$
▷ Sync. reset :	$\dfrac{\Gamma \vdash p : (d, \alpha)}{\Gamma \vdash \mathtt{R}(p) : (0, \alpha)}$
▷ Sync. co-reset:	$\dfrac{\Gamma \vdash p : (d, \alpha)}{\Gamma \vdash \mathtt{L}(p) : (0, \alpha)}$

Figure 13: Static type rules

$$\Gamma \vdash p : (d, \alpha)$$

that means *the program p in environment Γ is a tiled α-stream with synchronization length d.*

The typing environment Γ tells what the types of the variables that occur in p are. It is represented as a set of pairs of the form $(\mathtt{x}, (d, \alpha))$. We also write α_c for the basic type of the constant $\mathtt{c}$. This implicitly means that every constant is written in such a way that its type is unambiguous. However, in all examples and rules given below, we keep the notation simple with constant 0 meaning the constant 0_α for any of the basic type α.

Additionally, we provide in Figure 14 the derived rules associated with the last two program constructs.

▷ Operator:	$\dfrac{\Gamma \vdash p_1 : (d_1, \alpha_1) \quad \Gamma \vdash p_2 : (d_2, \alpha_2)}{\Gamma \vdash p_1 \,\mathtt{op}\, p_2 : (d_1 + d_2, \alpha_3)}$ with $\mathtt{op} : \alpha_1 \times \alpha_2 \to \alpha_3$
▷ Sync. product:	$\dfrac{\Gamma \vdash p_1 : (d_1, \alpha_1) \quad \Gamma \vdash p_2 : (d_2, \alpha_1)}{\Gamma \vdash p_1 \,;\, p_2 : (d_1 + d_2, \alpha_1 \times \alpha_2)}$

Figure 14: Derived static type rules

Remark 4.2 *As expected, the typing rule for infix operator $p_1 \,\mathtt{op}\, p_2$ derives from the mapping rule in Figure 13. Quite new in our proposal, the type rule for the synchronization product $p_1 \,;\, p_2$ also derives from that same rule when applied to the identity mapping $\mathtt{id}_{\alpha\times\alpha} : \alpha \times \alpha \to \alpha \times \alpha$, one per type α, hence leading to the derived rule described in Figure 14.*

It shall be clear that for every program p, for every environment Γ, there is at most one type (d, α) such that $\Gamma \vdash p : (d, \alpha)$. Moreover:

Theorem 4.3 *It is decidable if there exists Γ and (d, α) such that $\Gamma \vdash p : (d, \alpha)$.*

Proof. Deciding of the existence of the basic type α is standard. It thus poses no difficulty. One may even imagine to allow polymorphic types as in languages like ML.

Deciding of the existence of sync. length typing easily reduce to the resolution, on positive integers, of a finite system of linear fixpoint equations with positive coefficients only. Indeed, any equation of the form

$$x_i = a_{i,i}x_i + b_i(\{x_J\}_{j\neq i})$$

can be replaced by a (syntactically) simpler equation according to one of the following cases: if $a = 0$ then it $x_i = b_i(\{x_J\}_{j\neq i})$, if $a = 1$ then it implies that $0 = b_i(\{x_J\}_{j\neq i})$ which leads to further simplifications, and if $a > 1$ then the system has no solution. □

Example 4.4 The program `x1 = c + R(x1)` can be typed with sync. length 1. The program `x2 = L(x2) + c + R(x2)` can be typed similarly.

On the contrary, neither the program `x3 = c + x3` nor the program `x4 = x4 + c + x4` can be typed for they would have an right- or bi-infinite synchronization interval.

4.3 Semantics

Let p be a program. Let $\mathcal{X}_p$ be the set of variables that occur in p. Let Γ be a type assignment of variables such that $\Gamma \vdash p : (d, \alpha)$.

A valuation $\mathcal{E}$ for p is a map that associates every variable $\mathtt{x} \in \mathcal{X}_p$ of p with a tiled stream $\mathcal{E}(\mathtt{x})$. It is coherent with Γ when, for every variable $\mathtt{x} \in \mathcal{X}_p$, if $(\mathtt{x}, (d_{\mathtt{x}}, \alpha_{\mathtt{x}})) \in \Gamma$ then the tiled stream $\mathcal{E}(\mathtt{x})$ is a tiled $\alpha_{\mathtt{x}}$-stream with synchronization length $d_{\mathtt{x}}$.

A semantic for the program p under the valuation $\mathcal{E}$, assumed to be coherent with Γ, is then a mapping $[\![]\!]_{\mathcal{E}}$ that maps every subprogram p' of the program p to a tiled stream $[\![p']\!]_{\mathcal{E}}$ according the rules in Figure 15 and that, *moreover*,

▷ Constants: $d([\![\mathtt{c}]\!]_{\mathcal{E}}) = 0$ and

$$[\![\mathtt{c}]\!]_{\mathcal{E}}(k) = \begin{cases} \mathtt{c} & \text{when } k = 0, \\ 0 & \text{when } k \neq 0, \end{cases}$$

▷ Variable: $[\![\mathtt{x}]\!]_{\mathcal{E}}(k) = \mathcal{E}(\mathtt{x})(k)$,

▷ Mapping: $d([\![\mathtt{f}(p_1, \cdots, p_n)]\!]_{\mathcal{E}} = \sum_{i\in[1,n]} d([\![p_i]\!]_{\mathcal{E}})$

$$\text{and } [\![\mathtt{f}(p_1, \cdots, p_n)]\!]_{\mathcal{E}}(k) = \mathtt{f}(v_1, \cdots, v_n)$$

with $v_i = [\![p_i]\!]_{\mathcal{E}} \left(k - \sum_{0\leq j<i} d([\![p_i]\!]_{\mathcal{E}}\right)$,

▷ Assignment: $d([\![\mathtt{x} = p_1]\!]_{\mathcal{E}}) = d([\![p_1]\!]_{\mathcal{E}})$ and

$$[\![\mathtt{x} = p_1]\!]_{\mathcal{E}}(k) = [\![p_1]\!]_{\mathcal{E}}(k)$$

▷ Sync. reset : $d([\![\mathtt{R}(p_1)]\!]_{\mathcal{E}}) = 0$ and

$$[\![\mathtt{R}(p_1)]\!]_{\mathcal{E}}(k) = [\![p_1]\!]_{\mathcal{E}}(k)$$

▷ Sync. co-reset: $d([\![\mathtt{L}(p_1)]\!]_{\mathcal{E}}) = 0$ and

$$[\![\mathtt{L}(p_2)]\!]_{\mathcal{E}}(k) = [\![p_2]\!]_{\mathcal{E}}(k)$$

for every $k \in \mathbb{Z}$.

Figure 15: Static semantics rules

satisfies the fixpoint property (Y) described in Figure 16.

Remark 4.5 *As already announced in the introduction, operations on tiled streams are indeed defined as synchronizations followed by fusions. This becomes especially clear in the (point wise) extension of a mapping* $\mathtt{f} : \alpha_1 \times \alpha_2 \to \alpha_3$ *to a program construct of the form* $\mathtt{f}(p_1, p_2)$ *on some tiled* α_1*-stream* p_1 *and some tiled* α_2*-stream* p_2*. Indeed, computing* $\mathtt{f}(p_1, p_2)$ *amounts to:*

(Y) *For every* $\mathtt{x} \in \mathcal{X}_p$ *we have* $\mathcal{E}(\mathtt{x}) = [\![p_{\mathtt{x}}]\!]_{\mathcal{E}}$.

Figure 16: Fixpoint soundness rule

- ▷ *Synchronization (in time): computing the synchronized product* $p_1 \,;\, p_2$ *on the tiled* α_1*- and* α_2*-streams passed as arguments,*
- ▷ *Fusion (in space): applying mapping* `f` *in a point wise fashion on the resulting tiled* $\alpha_1 \times \alpha_2$*-stream in order to build the expected* α_3*-stream.*

This feature already appeared in the typing rules in Figure 13. It has been made explicit in Figure 15.

The semantics of infix operators and synchronized products just derive from the primitive semantics rules stated in Figure 15. The corresponding rules are stated in Figure 17,

▷ Operator: $d([\![p_1 \,\mathtt{op}\, p_2]\!]_{\mathcal{E}}) = d([\![p_1]\!]_{\mathcal{E}}) + d([\![p_2]\!]_{\mathcal{E}})$ and

$$[\![p_1 \,\mathtt{op}\, p_2]\!]_{\mathcal{E}}(k) = [\![p_1]\!]_{\mathcal{E}}(k) \,\mathtt{op}\, [\![p_2]\!]_{\mathcal{E}}(k - d([\![p_1]\!]_{\mathcal{E}}))$$

▷ Sync. product: $d([\![p_1 \,;\, p_2]\!]_{\mathcal{E}}) = d([\![p_1]\!]_{\mathcal{E}}) + d([\![p_2]\!]_{\mathcal{E}})$ and

$$[\![p_1 \,;\, p_2]\!]_{\mathcal{E}}(k) = ([\![p_1]\!]_{\mathcal{E}}(k), [\![p_2]\!]_{\mathcal{E}}(k - d([\![p_1]\!]_{\mathcal{E}})))$$

for every $k \in \mathbb{Z}$.

Figure 17: Derived static semantics rules

Example 4.6 Continuing Example 4.4, the tiled stream associated to `x1` is uniquely defined. It equals `c` when $k \geq 0$ and equals `0` when $k < 0$. The tiled stream associated to `x2` is also uniquely defined and equals `c` everywhere.

As another example, with uniquely defined semantics, the program `x3 = L(x3) + 0 + 1 + R(x3)` evaluates into a tiled stream of alternating `0` and `1`.

On the contrary, a program like `x = x` have many possible semantics per valuation of `x` although it can be typed. A program like `x = R(2 + x)`, that can also be typed, have no semantics since its value on 0 shall be infinite.

We easily check that:

Lemma 4.7 *When* $\mathcal{E}$ *is coherent with* Γ *then, for every subprogram* p_1 *that occurs in* p*, if* $\Gamma \vdash p_1 : (p_1, \alpha_1)$ *then* $[\![p_1]\!]_{\mathcal{E}}$ *is a tiled* α_1*-stream with* $d([\![p']\!]_{\mathcal{E}}) = d_1$.

4.4 Least fixpoint semantics

Since a program may have zero, one or several semantics as illustrated in Example 4.6, we privilege below the least fixpoint semantics that, when defined, provide a unique semantics.

Given a typed program p, let $\mathcal{E}_0$ be the valuation that maps every variable $\mathtt{x}$ that occurs in p to the constant 0. For every $n \in \mathbb{N}$, let then $\mathcal{E}_{n+1}$ be the valuation defined, for every variable $\mathtt{x} \in \mathcal{X}_p$, by $\mathcal{E}_{n+1}(\mathtt{x}) = [\![p_{\mathtt{x}}]\!]_{\mathcal{E}_n}$ for some/any assignment of the form $\mathtt{x} = p_{\mathtt{x}}$ that occurs in p where $[\![p']\!]_{\mathcal{E}_k}$ is computed following the rules of Figure 15.

A semantic $[\![]\!]_{\mathcal{E}}$ for the program p is said to be the least fixpoint semantics of p when the sequence $\{\mathcal{E}_n\}_{n\geq 0}$ defined above converges to $\mathcal{E}$ according to the following metrics.

The (ultrametric) distance between two tiled streams s_1 ans s_2 is defined to be $d(s_1, s_2) = 1/2^k n$ where n is the greatest integer such that, for all $k \in [-n, n]$ we have $s_1(k) = s_2(k)$. Then the distance between two environment $\mathcal{E}_1$ and $\mathcal{E}_2$ is define by $d(\mathcal{E}_1, \mathcal{E}_2) = \max\{d(\mathtt{x}, \mathtt{x}) : x \in \mathcal{X}_p\}$.

We easily check, by induction on the syntactical complexity of programs that such a semantics is uniquely defined when it exists. A program that has a least fixpoint semantics is called robust.

Remark 4.8 *In our application perspectives as in the proposed examples, it makes sense to define* 0 *as the least value for elements of type* `bool`*,* `eventSet(E)` *or even* `natural`*.*

Indeed, we do not expect any non terminating musical values that would necessitate an additional undefined value $\perp$*. On sets of events, the default musical value is silence and it is modeled by the empty set* 0*. When seen as control signal, the default boolean value is* `Off`*. It is modeled by the boolean* 0*. Elements of type* `natural` *can also be seen as intensity levels hence with* 0 *as the relevant default value.*

Of course, for more general basic types even as simple as `int` *or* `float`*, the least fixpoint semantics shall rather be defined with extra undefined value* $\perp$*, one for every basic type, that is absorbant w.r.t. every operators and functions that are assumed to have strict semantics. This is rather well known in Domain theory. As this goes out of the scope of this paper, it is thus not treated here.*

4.5 Out of time musical examples

Let `E` be a set of MIDI-like musical event defined as follows. For every note `N`, there is an event `N` for the *noteOn* event for that note and an event `Nc` for its continuation event *noteCont*. Of course, as in MIDI, by note we mean a pitch class but possibly extended with a track number, an instrument number, an energy level, etc. The main interest of such a variant is that the empty event set 0 denotes silence. Then, as shown in the following examples, melodic lines can easily be encoded by tiled streams of type `eventSet(E)`.

Example 4.9 (Simple melodies) We assume that the elementary durations of events are eighth notes. Writing `<N:d>` for every note `N` for the program `{N}+{Nc}+`$\cdots$`+{Nc}` when `Nc` is repeated $d-1$ times, then the tiled stream

```
a = <C4:2>
```

defines the note `C4` with a duration of two eighths, that is a quarter. Slightly more complex, the tiled stream

```
b = L(B3) + <C4:2> + [0,6]
```

with `[`p`:d]` meaning `d` times the sum of p, simply encodes a 4 beats measure composed of one quarter note `C4` followed by silences, that is preceded by an anacrusis made of an eighth note `B3`.

Such a fairly simple example clearly show how all finite examples that are given in Section 2 can be encoded in our proposal. The binary function *merge* that is used in Section 3 is just defined by $merge(p_1; p_2) = p_1 + p_2$ for every tiled stream p_1 and p_2.

Example 4.10 (An infinite canon) Another example is the infinite canon one can built as follows. Given four melodic lines `m1`, `m2`, `m3` and `m4` built with the techniques shown above, let `p` be the program defined by

```
x1 = m1+R(m2+m3+m4+x1)

x2 = x1*R(0)+m1+R(m2+m3+m4+x2)

x3 = x2*R(0)+m1+R(m2+m3+m4+x3)

x4 = x3*R(0)+m1+R(m2+m3+m4+x4)
```

with the newline separated sequence of `xi =`p_is is a shorthand notation for the (commutative) synchronized product

`R(x1 =`p_1`) ; R(x2 =`p_2`) ; R(x3 =`p_3`) ; R(x4 =`p_4`)`

In that construction, `xi *`$R(0)$ allows for using the synchronization interval of `xi` without its values as it results in the constant 0 tiled streams with a synchronization that equals the synchronization length of `xi`.

Observe that we need one melody at least to be of non zero synchronization length for these examples to have unique semantics. Observe also that such an encoding even allows for canons built on melodic lines with anacrusis and/or late conclusive notes.

5 Dynamic (in-time) T-calculus

We aim now at extended our programming language proposal so that it can handle outputs and inputs. This also poses the question of computability of the programs.

Handling outputs means producing tiled stream values in the order defined as the natural order on the domain $\mathbb{Z}$ of every tiled streams. Handling inputs means extracting from input streams some input synchronization intervals that can be used to dynamically position other (musical) tiled streams along the time line.

In both case, some type systems are used to guarantee that typed programs are not incoherent w.r.t. the flow of time: they can indeed be executed in a reactive way.

Indeed, numbers of statically well typed programs are obviously incoherent w.r.t. time flow as, for instance, e.g. `x = L(x) + m` with some finite melody `m` to be read in time.

Form now on, let p be a T-program and let $\mathcal{E}$ be some environment $\mathcal{E}$ coherent with a typing environment Γ such that $[\![p]\!]_{\mathcal{E}}$ is the least fixpoint semantics of p.

5.1 Computing (in-time) outputs

We aim here at finding simple (computable) conditions in such a way that the values of both $\mathcal{E}$ and $[\![p]\!]_{\mathcal{E}}$ can be computed. More precisely, we look for computable typing that ensures that there is some computable start date $k_0 \in \mathbb{Z}$ such that, for every $k \in \mathbb{Z}$:

(P1) if $k < k_0$ then $[\![p]\!]_{\mathcal{E}}(k) = 0$; we say that the program p is *backward silent*,

(P2) if $k \geq k_0$ then both $\mathcal{E}(k)$ and $[\![p]\!]_{\mathcal{E}}(k)$ are indeed computable; we say that the program p is *forward computable*.

We first study the forward computability property. This can be done by analyzing the recurrence schema that is induced by p. More precisely:

Definition 5.1 (Memory structure) *A mapping μ that maps program p and variable $\mathtt{x} \in \mathcal{X}_p$ that occurs in p to a set $\mu(p, \mathtt{x}) \subseteq \mathbb{Z}$ such that, for every $k \in \mathbb{Z}$:*

if, for every $\mathtt{x} \in \mathcal{X}_p$, for every $k' \in \mu(p, \mathtt{x})$, the value of $[\![\mathtt{x}]\!]_{\mathcal{E}}(k + k')$ is known, then the value $[\![p]\!]_{\mathcal{E}}(k)$ is uniquely determined,

is called an (translation invariant) memory structure *for the program p.*

Remark 5.2 *The uniform (time invariant) structure of the computations induced (when defined) by T-calculus programs allows for such definition of time invariant memory structure. A more general definition of the form $\mu(p, \mathtt{x}, k)$ that also specifies the date k at which we examine functional dependencies is not necessary.*

For instance, we can take $\mu(\mathtt{1 + x}, \mathtt{x}) = \{-1\}$. Another example, with `x` and `y` two distinct variables, we can take $\mu(\mathtt{1 + x}, \mathtt{y}) = \emptyset$. Indeed, μ describes some functional dependencies in the sense of data base theory.

We aim now at computing some syntax driven memory structure. Let μ be inductively defined by the rule described in Figure 18 where for any subpro-

▷ Constants: $\mu(\mathtt{c},\mathtt{x}) = \emptyset$,

▷ Variable: $\mu(\mathtt{y},\mathtt{x}) = \emptyset$ when $\mathtt{x}$ and $\mathtt{y}$ are distinct variables and $\mu(\mathtt{x},\mathtt{x}) = \{0\}$ otherwise,

▷ Mapping: $\mu(\mathtt{f}(p_1,\cdots,p_n),\mathtt{x}) =$

$$\bigcup_{1\leq i\leq n} \mu(p_i,\mathtt{x}) - \sum_{0\leq j<i} d(p_i),$$

▷ Assignment: $\mu(\mathtt{y}=p_y,\mathtt{x}) = \mu(p_{\mathtt{y}},\mathtt{x})$,

▷ Sync. reset : $\mu(\mathtt{R}(p_1),\mathtt{x}) = \mu(p_1,\mathtt{x})$,

▷ Sync. co-reset: $\mu(\mathtt{L}(p_1),\mathtt{x}) = \mu(p_1,\mathtt{x}) + d(p_1)$.

Figure 18: Sufficient functional dependency rules

gram p' of the program p, we denote $d(p') \in \mathbb{N}$ the synchronization length of the subprogram p' induced by the type environment Γ and, for any set $X \subseteq \mathbb{Z}$ and any $n \in \mathbb{Z}$, we denote by $X - n$ the set $\{m - n \in \mathbb{Z} : m \in X\}$.

Lemma 5.3 *The mapping μ inductively defined by the rules of Figure 18 is a* memory structure *for the program p.*

Proof. This follows from the fact that, due to the linear nature of T-calculus programs, the functional dependencies described in the definition of memory structure is invariant under translation. Then, the rules depicted in Figure 18 just compute such syntactical dependencies on 0. □

Example 5.4 An example computation of μ on the program `x= 1 + R(1 + L(x))` is depicted in Figure 19. For the sake of simplicity we have omitted to picture the reset operator. In that example, we have

$$\mu(\mathtt{1},\mathtt{x}) = \emptyset \text{ and } \mu(\mathtt{x},\mathtt{x}) = \{0\}$$

hence, by applying the co-reset rule, we have

$$\mu(\mathtt{L(x)},\mathtt{x}) = \{1\}$$

Then, by applying the mapping rule, we have

$$\mu(\mathtt{1 + L(x)},\mathtt{x}) = \{0\}$$

and thus, by applying the reset rule,

$$\mu(\mathtt{R(1 + L(x))},\mathtt{x}) = \{0\}$$

hence

$$\mu(\texttt{x= 1 + R(1 + L(x))},\texttt{x}) = \{-1\}$$

Then we check that, with least fixpoint semantics, we indeed have, for every $k \in \mathbb{Z}$,

$$[\![x]\!]_{\mathcal{E}}(k) = \begin{cases} 0 & \text{if } k < 0 \\ [\![x]\!]_{\mathcal{E}}(k-1) + 1 & \text{if } k \geq 0. \end{cases}$$

This computation rule is indeed coherent with our computation of the inductive memory structure μ.

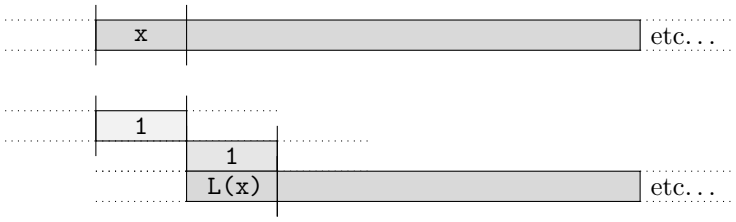

Figure 19: Computing a memory structure

Let $\bar{\mathbb{N}}$ be the set of positive integer extended with a greatest element ∞.

Definition 5.5 (Synchronization profile) *A synchronization profile for the program p is a triple*

$$(l, d, r) \in \bar{\mathbb{N}} \times \mathbb{N} \times \bar{\mathbb{N}}$$

such that d is the synchronization length of program p, i.e. $\Gamma \vdash (d, \alpha)$ for some type α, and such that, for every $k \in \mathbb{Z}$, if $[\![p]\!]_{\mathcal{E}}(k) \neq 0$then $d - l \leq k \leq d + r$, with $d - l \leq k$ (resp. $k \leq d + r$) that is true whenever $l = \infty$ (resp. $r = \infty$).

Before giving a set of rules to compute sync. profile types, the notion if sync. profile itself, and the way it evolves in a sum is depicted in Figure 20.

Remark 5.6 *The set of triples $(l, d, r) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ equipped with the product defined by*

$$(l_1, d_1, r_1) \cdot (l_2, d_2, r_2) = (\max(l_1, l_2 - d_1), d_1 + d_2, \max(r_1 - d_2, r_2)$$

is a submonoid of the free inverse monoid (see e.g. [17]) generated by one generator. In particular, the product is associative and it has $(0, 0, 0)$ as neutral element. The sync. profiles, that also allows ∞ as left or right value, form a submonoid of the filter completion of that monoid.

We consider now the rules described in Figure 21 where Δ is an environment that associates variables to their sync. profiles.

We say that a profile (l, d, r) is smaller than a profile (l', d', r') when $l \leq l$, $d \leq d'$ and $r \leq r'$. One can easily check that this is an order relation[1]. It is extended point wise to environment.

[1] it is even the reverse order of the natural order of the underlying inverse monoid

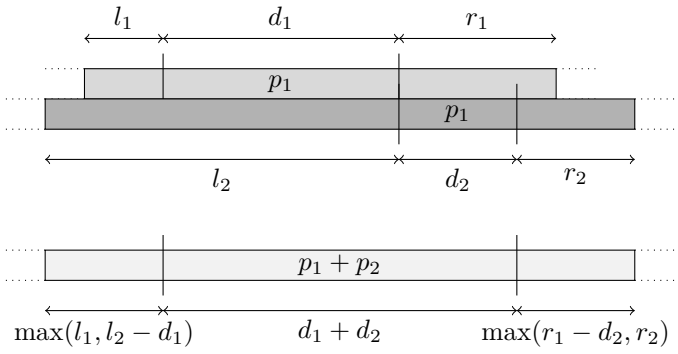

Figure 20: Computing sync. profiles

Lemma 5.7 *The least synchronization profiles (and the corresponding environment Δ) satisfying the rules describe in Figure 21 is computable.*

Proof. The computability of synchronization lengths follows from the fact that we assume p is typed in a given environment Γ.

Computability of left and right parts of sync. profile follows from the fact that the rules depicted in Figure 21 induce conjunctions of inequalities of the form

$$l_i \geq c_i + \sum_{j \in J} l_j$$

which, whenever $c_i \neq 0$ and $i \in J$, implies that $l_i = \infty$. That fact allows for accelerating the least solution computation that otherwise may be infinite as shown for instance by the program `x = 1 + R(x) + 1` with sync. profile $(0, 1, \infty)$. □

When, $\Delta \vdash p : (l, d, r)$ with finite l, we know that for every $k \leq l$ we have $[\![p]\!]_{\mathcal{E}} = 0$, that is, the program p satisfies the property (P1): it is *backward silent*.

Our main effectivity result is then stated in the following theorem.

Theorem 5.8 *If the program p is backward silent and for every variable $\mathtt{x}$ bound in p, the program p_x is backward silent, and if, for every $k \in \mu(\mathtt{x}, \mathtt{x})$ we have $k < 0$ then p also satisfies property (P2): it is forward playable.*

Moreover, the least fixpoint semantics $[\![p]\!]_{\mathcal{E}}$ or program p and the associated functions $\mathcal{E}(\mathtt{x})$ for $\mathtt{x} \in \mathcal{X}_p$ are functions computable by means of finite state sequential transducers.

Proof. Backward silence hypotheses ensure the existence of some $k_0 \in \mathbb{Z}$, such that for every $k' \leq k$, for every subprogram p' of p, we have $[\![p']\!]_{\mathcal{E}}(k') = 0$.

The fact that for every variable $\mathtt{x}$, every element of $\mu(\mathtt{x}, \mathtt{x})$ is strictly negative ensures that, at every instant, $[\![\mathtt{x}]\!]_{\mathcal{E}}$ only depends on the past its values hence $[\![p]\!]_{\mathcal{E}}$ is indeed forward playable.

▷ Constants:
$$\frac{}{\Delta \vdash \mathtt{c} : (0,1,0)}$$

▷ Variables:
$$\frac{(\mathtt{x},(l,d,r)) \in \Delta}{\Delta \vdash \mathtt{x} : (l,d,r)}$$

▷ Mapping:
$$\frac{\Delta \vdash p_i : (l_i,d_i,r_i) \quad (i \in [1,n])}{\Delta \vdash \mathtt{f(}p_1,\cdots,p_n\mathtt{)} : (l,d,r)}$$
with $l = \max\left(l_i - \sum_{1\leq j<i} d_j\right)$, $d = \sum_i d_i$,
and $r = \max\left(r_i - \sum_{i<j\leq n} d_j\right)$,

▷ Assignment:
$$\frac{\Delta \vdash \mathtt{x} : (l,d,r) \quad \Delta \vdash p : (l,d,r)}{\Delta \vdash \mathtt{x=}p : (l,d,r)}$$

▷ Sync. reset :
$$\frac{\Delta \vdash p : (l,d,r)}{\Delta \vdash \mathtt{R(}p\mathtt{)} : (l,0,d+r)}$$

▷ Sync. co-reset:
$$\frac{\Delta \vdash p : (l,d,r)}{\Delta \vdash \mathtt{L(}p\mathtt{)} : (l+d,0,r)}$$

Figure 21: Sync. profile type rules

Then, the finiteness of the memory comes from the fact that sets of the form $\mu(p', \mathtt{x})$ for subprograms p' of p and variables $\mathtt{x} \in \mathcal{X}_p$ of p are all finite. □

5.2 Monitoring (in-time) inputs

Assume now that the program p is as depicted in the hypothesis of Theorem 5.8. We aim now at monitoring input streams that is, converting 'on the fly' real time streams of input values into relevant tiled stream.

By default and as illustrated in Section 3, an input stream can be seen as a tiled stream s with synchronization length $d(s) = 0$ and, for every $k < 0$, $s(k) = 0$.

Our point is now to convert such an input tiled streams with zero length synchronization into a tiled stream with value dependent synchronization.

This is done via a new program construct `wait()` that is defined on an arbitrary tiled stream where, implicitly, the date 0 is understood as the date from which the tiled stream is listened to.

The syntax of T-calculus program is extended by the following construct: for some program p, some constant `c` and some positive integer constant `dmax`.

The typing rule for `wait(i,c,dmax)` is defined as depicted in Figure 23

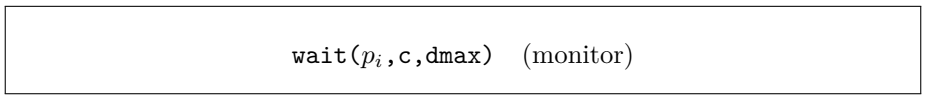

Figure 22: The `wait()` program construct

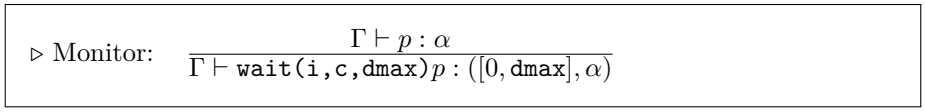

Figure 23: Input monitor type rule

The semantics of such a construct is defined as depicted in Figure 24

Since the synchronization length of a `wait()` construct is statically bounded, it has finitely many possible values and thus all decidability results, be they for type inference or for semantics that are stated above, remains valid.

A typical example of the `wait()` semantics is depicted in Figure 25. Of course, one may imagine to allow, in `wait()` construct, arbitrary boolean programs instead of constants `c`. Such a possibility, as many other possible syntactic sugar proposals, will have to be tested on more complex examples.

Remark 5.9 *One may also imagine to allow ∞ as maximal waiting bound in a* `wait()` *construct. This of course implies that some program may not terminate with infinite synchronization/waiting length.*

However, even assuming termination, this introduced infinite sets of possible synchronization length in typing rules hence possibly raising some new decidability question that may have negative answers.

5.3 More examples

The following example shows that typical stream (or signal) bounded processing functions can also be described as T-calculus programs.

Example 5.10 (Local stream processing) Let $F : (\mathbb{Z} \to \alpha_1) \to (\mathbb{Z} \to \alpha_2)$ be a stream processing function, e.g. an echo for instance. We assume that F is coherent with time flow, i.e. for every $k \in \mathbb{Z}$, $F(s)(k)$ only depends on values $s(k')$ of s with $k' \leq k$. We also assume that F is local. Together with time coherence, this means that there exists c_0 and $c \in \mathbb{N}$ with $c_0 \geq c$ and a mapping $g : \alpha_1^c \to \alpha_2$, such that, for every input stream $s : \mathbb{Z} \to \alpha_1$, for every $k \in \mathbb{Z}$,

$$F(s)(k + c_0) = g(s(k), s(k + 1), \cdots s(k + c - 1))$$

i.e. $F(s)$ is computed from a sliding window on s of length c.

▷ Monitor:

$$[\![\texttt{wait(p,c,dmax)}]\!]_{\mathcal{E}}(k) = \begin{cases} 0 & \text{if } k < 0, \\ [\![p]\!]_{\mathcal{E}}(k) & \text{if } 0 < d, \\ 0 & \text{if } d \leq k, \end{cases}$$

with $d([\![\texttt{wait(i,c,dmax)}]\!]_{\mathcal{E}}(k))$ the greatest integer $d \in \mathbb{N}$ such that $0 \leq d \leq \texttt{dmax}$ and $[\![i]\!](k) \neq \texttt{c}$ for every $0 \leq k < d$.

Figure 24: Semantics of input monitors

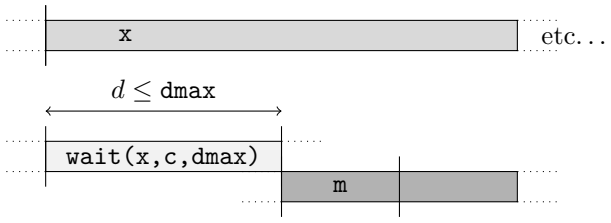

Figure 25: Waiting on a monitored input

Then F can be encoded by the program `Fs` defined as follows.

$\texttt{s}_1\texttt{= s}$

$\texttt{s}_2\texttt{= R(s}_1\texttt{) + 0 + s}$

...

$\texttt{s}_c\texttt{= R(s}_{c-1}\texttt{) + s}$

$\texttt{Fs = [0:c}_0\texttt{]+g(s}_c\texttt{)}$

In that encoding, we choose to make the synchronization length of the tiled stream s to be equal.

Remark 5.11 *By analyzing such an example in mode depth, we observe that adding a stream of (finite memory) states in the example above allows for the encoding of an arbitrary finite state sequential transducer with F now seen as the transition function of that transducer.*

In other words, by Theorem 5.8 and that example, when restricting to finite basic types, say event sets, there is a correspondance between the backward silent forward playable T-calculus programs and the finite state sequential transducers [21].

The use of the `wait()` construct allows for the definition of input dependent positioning of tiled streams.

Example 5.12 Let `x` be a tiled stream variable denoting a boolean input signal. Let p be some tiled program to fired either when `x` becomes true or within 40 time steps at most. Then this can be done by the program

$$\texttt{wait(x,1,40)*R(0) +}\ p$$

The construction `wait(x,1,40)*R(0)`, as in a previous example, allows for keeping, in the sum, the synchronization length of `wait(x,1,40)` while droping its values. Of course, a more adhoc syntax may be more appropriate in that case.

Remark 5.13 *In that last example, there might be a violation of causality. Indeed, given the least synchronization profile* (l, d, r) *of the program* p*, if* $l > 0$ *we need to start the program* p *at an unpredictable date since the synchronization length of* `wait(`p`,1,40)` *is unpredictable.*

To prevent this, we can forbid, by some extra static analysis, such phenomenon to occur. The synchronization profile type system described in Figure 21 can easily be extended in order to statically (and somehow drastically) cope with such phenomenon.

The problem raised by causal anticipation in music systems is much broader than what is illustrated in this example and the somehow drastic solution we propose here.

Indeed, as illustrated by score follower based music software such as *Antescofo* [4] where sorts of input monitors predict at every steps what it the distance to the end of the monitoring, there are other and more clever ways to cope with such causal incoherence. However, their studies and potential adaptation to our T-calculus proposal goes beyond the scope of the present paper.

6 Related work

The idea of distinguishing effective starts and stops from logical ones implicitly appears in the programming language Loco [5] via the `pre` and `post` program constructs. As far as we know, there have been no follow up of this work.

More than two decades later, our proposal, that makes such an idea explicit and rather quite formal, is based on the first author's study of rhythm representations [10]. These ideas have been developed in [1] and experimented in [14] in the context of music.

It is also inspired by both the music description language *Elody* [18] and the signal processing language *Faust* [6]. It can be seen as a tiled extension proposal of these two functional languages.

It is worth mentioning that the underlying concepts have also been studied in the abstract in number in the field of formal language theory [12, 9, 13, 11].

7 Conclusion

We have thus describe an encoding of symbolic music strings and streams into tiled streams. As illustrated in length in Sections 2 and 3, it is based on a tiled signal algebra that, by embedding both strings and streams data-type, allows for a more accurate description of typical musical constructions such as *play together* or *play one after the other*. The associated language, the T-calculus, is thus proposed as an implementation of these ideas. It relies on the notion of tiled product that combines both sequential product and parallel product features.

Throughout Sections 4 and Section 5, we describe computable type systems that allow for restricting to a large class of T-calculus programs that can effectively be computed in real (forward) time with bounded memory. This shows that the typed T-calculus can be seen as a programming language counterpart of finite state sequential transducers that have been well studied in formal language theory (see e.g. [21]). The added value of our approach lies in the allowed overlaps that can be defined via tiled stream resets and co-resets that, for music programing but also for system modeling that it provides more abstract modeling metaphors.

Last, the T-calculus is yet not a full programming language but indeed a calculus. No lambda abstraction allows for the definition of (conditional or recursive) program that would handle tiled streams. However, the study of the potential use and effectivity issues of such an extension is postponed to further work.

Acknowledgment

The authors wish to express gratitude for the deep comments and harsh critics the participants of the INEDIT workshop held in Bordeaux in spring 2013 made on an earlier version of this work.

References

[1] 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.

[2] L. Bigo. Symbolic representations and topological analysis of musical structures with spatial programming. In *JCAAAS*, Paris, 2010.

[3] L. Bigo, J.-L. Giavitto, and A. Spicher. Building topological spaces for musical objects. In *Mathematics and Computation in Music*, volume 6726 of *LNAI*, Paris, France, Juin 2011.

[4] A. Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, 2008.

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

[6] D. Fober, Y. Orlarey, and S. Letz. FAUST architectures design and OSC support. In *14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 231–216. IRCAM, 2011.

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

[8] 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)*, San-Diego, 2007. ACM Press.

[9] D. Janin. Quasi-recognizable vs MSO definable languages of one-dimensional overlapping tiles. In *Mathematical Found. of Comp. Science (MFCS)*, volume 7464 of *LNCS*, pages 516–528, 2012.

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

[11] D. Janin. Algebras, automata and logic for languages of labeled birooted trees. In *Int. Col. on Aut., Lang. and Programming (ICALP)*, volume 7966 of *LNCS*, pages 318–329. Springer, 2013.

[12] D. Janin. On languages of one-dimensional overlapping tiles. In *Int. Conf. on Current Thrends in Theo. and Prac. of Comp. Science (SOFSEM)*, volume 7741 of *LNCS*, pages 244–256, 2013.

[13] D. Janin. Overlaping tile automata. In *8th International Computer Science Symposium in Russia (CSR)*, volume 7913 of *LNCS*, pages 431–443. Springer, 2013.

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

[15] J. Kellendonk. The local structure of tilings and their integer group of coinvariants. *Comm. Math. Phys.*, 187:115–157, 1997.

[16] J. Kellendonk and M. V. Lawson. Tiling semigroups. *Journal of Algebra*, 224(1):140 – 150, 2000.

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

[18] S. Letz, Y. Orlarey, and D. Fober. Real-time composition in Elody. In *Proceedings of the International Computer Music Conference*, pages 336–339. ICMA, 2000.

[19] S. Letz et al. The LibAudioStream library, 2012. `http://libaudiostream.sourceforge.net/`.

[20] O. Michel, A. Spicher, and J.-L. Giavitto. Rule-based programming for integrative biological modeling. *Natural Computing*, 8(4):865–889, 2009.

[21] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.