HAL
archives-ouvertes.fr

# Flux média tuilés polymorphes: une sémantique opérationnelle en Haskell

Théis Bazin, David Janin

▶ To cite this version:

Théis Bazin, David Janin. Flux média tuilés polymorphes: une sémantique opérationnelle en Haskell. Journées Francophones des Langages Applicatifs (JFLA), Jan 2015, Val d'Ajol, France. 2015. <hal-01091736>

HAL Id: hal-01091736
https://hal.archives-ouvertes.fr/hal-01091736

Submitted on 6 Dec 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Flux média tuilés polymorphes: une sémantique opérationnelle en Haskell

Théis Bazin[1] *
ENS Cachan,
theis.bazin@ens-cachan.fr

David Janin [2] †‡
LaBRI, Bordeaux
david.janin@labri.fr

6 décembre 2014

**Résumé**

De nombreux outils sont aujourd'hui disponibles pour l'analyse et la production temps réel de flux média temporisés : son, vidéo, animation... Néanmoins, la coordination de ces outils, la synchronisation des flux qu'ils analysent et produisent, sur des échelles de temps de valeurs et même de nature différentes, reste une affaire délicate.

Le modèle des *Tiled Polymorphic Temporal Media* (ou TPTM), qui combine en un même formalisme le contenu média de ces flux et leurs marqueurs de synchronisation, vise a remédier à cela. Dans le modèle, le produit de deux flux ainsi enrichis, paramétré par ces marqueurs de synchronisation, est tout à la fois séquentiel et parallèle : c'est un produit tuilé.

D'un point de vue théorique, la sémantique de ces flux tuilés peut être décrite à l'aide des monoïdes inversifs. Pour l'aspect pratique, nous proposons ici, en Haskell, la première implémentation réellement polymorphe et inversive de ces *Tiled Polymorphic Temporal Media*. Notre implémentation permet en outre, via le mécanisme d'évaluation paresseuse d'Haskell, de distinguer simplement la syntaxe de ces flux – un système d'équations tuilées – de leur sémantique opérationnelle – la résolution de ce système à la volée.

## 1 Introduction

Dès les premiers ordinateurs, l'outil informatique est utilisé pour la conception et la production d'environnements sonores. En 1957, les chercheurs du Bell Labs [19] prouvent qu'un ordinateur peut synthétiser des sons variant dans le

*stagiaire au sein du LaBRI de juin à juillet 2014

†en délégation à l'INS2I/CNRS du 1/09/2013 au 31/08/2014 et à INRIA Bordeaux du 1/09/2013 au 28/02/2015

‡ce travail est partiellement soutenu par le projet INEDIT, ANR-12-CORD-009,

temps selon n'importe quelle échelle de hauteur ou selon n'importe quelle forme d'onde. Cette approche orientée signal, relativement bas niveau, traite des caractéristiques physiques et acoustiques du son, à l'aide d'outils comme la FFT ou les nombreuses autres transformées développées en traitement du signal.

Aujourd'hui, la recherche en acoustique informatique offre des résultats probants sur la simulation d'instruments de musique. Au-delà de l'éclairage que ces études donnent sur les propriétés acoustiques des instruments étudiés, elles permettent aussi de produire de la musique à l'aide d'ordinateurs, parfois même au moyen de modèles d'instruments physiquement impossibles, comme des trompettes à colonne d'air arbitrairement longue [19].

La nécessité de combiner ces sons dans le temps pour produire des pièces musicales conduit, en parallèle, au développement de nouveaux langages de description musicale. Ainsi, la norme *MIDI* définit par exemple une note comme un son déclenché par un événement de début, *NOTE_ON*, et terminé par un événement de fin, *NOTE_OFF*. Ce faisant, on passe d'une description concrète, synchrone, de la musique – le traitement du signal – à une description asynchrone de la musique – la programmation par événements.

Cette approche orientée événements autorise alors une description structurelle de la musique. Les notations musicales classiques de la musique occidentale peuvent être incorporées et enrichies dans ces nouveaux environnements de développement musical. Le logiciel LilyPond par exemple, propose, à l'instar de LaTeX, un langage haut niveau dédié à l'écriture et la mise en page de partitions.

De nombreuses suites logicielles, telles qu'*Open Music* [1], intègrent aujourd'hui des capacités à la fois de synthèse sonore et de composition musicale. Pour le développement proprement dit, plusieurs bibliothèques audio et musicales intégrées aux langages de programmation modernes aident à combiner et synchroniser synthèse sonore et contrôle événementiel afin de produire des systèmes musicaux interactifs complexes.

Néanmoins, la réalisation de ces environnements ne va pas de soi. La double nécessité de produire et de synchroniser de nombreuses sous-séquences musicales peut rendre l'écriture de ces environnements particulièrement fastidieuse. Les descriptions musicales obtenues sont souvent très complexes, difficiles à vérifier et difficiles à maintenir. Il manque encore des outils de description structurée de ces systèmes qui autoriseraient un développement hiérarchique et modulaire.

En un sens, malgré les nombreuses approches existantes, on ne dispose pas encore de modèles formels permettant de décrire les nombreux systèmes musicaux développés empiriquement à travers l'histoire de la musique. En particulier, la notation musicale classique, dont la première vocation est de permettre aux musiciens une lecture rapide des pièces musicales à jouer sur scène, ne saurait constituer un tel formalisme. Elle fait la part trop belle à l'implicite, un implicite résolu par la culture musicale de l'interprète.

Un des enjeux actuels de l'informatique musicale est donc de définir formellement la musique à un niveau abstrait, pour ensuite raisonner sur les programmes décrivant les systèmes musicaux réalisés et disposer d'un cadre de travail tout à la fois accessible et rigoureux. Par ailleurs, résoudre cette problématique de structuration de l'espace et du temps, particulièrement sensible en musique,

pourrait fournir des outils applicables à d'autres domaines de l'informatique des systèmes temporisés. De tels systèmes peuvent par exemple produire de la vidéo, des animations ou même du contrôle-commande de robots.

Le modèle des *Tiled Polymorphic Temporal Media* (ou TPTM) [12, 13, 15], qui généralise et explicite une approche ancienne [5], vise a résoudre cette difficulté [2, 16] en combinant en un même formalisme le contenu média des flux manipulés et leurs marqueurs de synchronisation. Dans ce modèle, le produit de deux flux média ainsi enrichis, paramétrés par ces marqueurs de synchronisation, est tout à la fois séquentiel et parallèle : c'est un produit *tuilé*. Pour la théorie [3, 13, 14], la sémantique des ces flux tuilés peut être décrite à l'aide des *monoïdes inversifs* [17]. Pour ce qui est de la pratique, nous proposons ici, en Haskell, la première implémentation réellement polymorphe et inversive de ces *Tiled Polymorphic Temporal Media*. Notre implémentation permet en outre, via le mécanisme d'évaluation paresseuse d'Haskell, de distinguer simplement la syntaxe de ces flux – un système d'équations tuilées – de leur sémantique opérationnelle – la résolution de ce système à la volée.

# 2 Du tuilage musical au tuilage polymorphe

Dans cette section, nous rappelons les caractéristiques principales de la notion de *Polymorphic Temporal Media* (ou PTM) proposée par Hudak [8] et nous montrons comment ce modèle, enrichi par tuilage, conduit aux *Tiled Polymorphic Temporal Media* [12, 13] dont il est question ici.

Ces définitions nous permettent de voir pourquoi l'implémentation des *Tiled Polymorphic Temporal Media* proposée dans [13] échoue partiellement à rendre compte de la structure de monoïde inversif de la sémantique des tuiles, ce qui fait l'objet de la nouvelle implémentation proposée dans la partie suivante.

## 2.1 Les *Polymorphic Temporal Media*

Hudak propose en 2004 [8] le modèle générique des *Polymorphic Temporal Media* qui s'adapte en particulier à la musique. Plus précisément : un *média* est une information transmise à un utilisateur. Un média est dit *temporisé* si son contenu varie selon une échelle de *temps*. On décrit le temps comme un axe indexé par des valeurs prises dans un ensemble donné. On parle de flux média temporisés *polymorphes* pour indiquer que les traitements effectués sur ces flux sont indépendants du type du média considéré : musique, vidéo, animation, etc. . . .

Dans l'approche de Hudak, implémentée en Haskell [11], tout flux média temporisé abstrait est engendré à partir de flux média élémentaires par *compositions séquentielles* ou *parallèles*. Les PTM sont ainsi définis comme le *type de*

*données polymorphe* :

```
data Media a = Prim a | Media a :+: Media a | Media a :=: Media a
```

où :

- `Prim a` représente un média atomique d'une durée fixe,
- `(:+:) :: Media a -> Media a -> Media a` est l'opérateur de composition séquentielle,
- `(:=:) :: Media a -> Media a -> Media a` est l'opérateur de composition parallèle.

Ce modèle, d'un grande simplicité syntaxique, demeure pourtant suffisamment général pour une utilisation pratique. Hudak propose divers exemples de types de base `T` qui permettent de modéliser à travers `Media T` de la musique ou encore des images animées.

Les propriétés minimales vérifiées par les échelles de temps considérées sont d'être totalement ordonnées – ce qui permet de construire un axe – et de disposer d'une structure de groupe – ce qui permet de naviguer sur cet axe de façon naturelle, en avant et en arrière, et de définir tout à la fois des *positions* et des *durées* sur cet axe. Par la suite, dans une approche polymorphe, nous supposerons que l'on dispose d'un groupe totalement ordonné $G$ utilisé comme échelle de temps sur les médias temporisés considérés.

En pratique, on peut utiliser $(\mathbb{Z}, +)$ – une échelle de temps *discrète* –, $(\mathbb{Q}, +)$ – une échelle de temps *dense* –, ou même, dans les approches théoriques, $(\mathbb{R}, +)$ – une échelle de temps *continue*. Hudak quant à lui utilise le type Haskell **Rational** qui permet un codage exact des nombres rationnels.

Cette approche de nature algébrique permet d'axiomatiser la sémantique des *Polymorphic Temporal Media*. L'analyse des programmes écrits dans ce formalisme devient plus facile, le système de typage d'Haskell offrant au programmeur une aide à la programmation. Le lecteur intéressé peut consulter l'ouvrage *The Haskell School of Music* [10] pour plus de détails. La théorie des *$\omega$-semi-groupes* [18] (voir aussi [6]) offre un éclairage pertinent quant aux difficultés inhérentes à la manipulation conjointe de flux finis et infinis.

La sémantique des opérateurs de génération de flux média temporisés, d'apparence très intuitive, n'est pas sans poser de réelles difficultés. En effet, la sémantique de `Prim` ne pose apparemment pas de difficultés particulières : ce constructeur `Prim` ne fait qu'encapsuler son argument. Il s'agit d'un simple plongement de `a` dans `Media a`. La sémantique du produit séquentiel `(:+:)` semble tout aussi simple. Dès que chaque *Polymorphic Temporal Media* dispose d'un *début* et d'une *fin* uniquement définis – qui correspondent au début du premier événement du média et à la fin du dernier –, le produit séquentiel `(m1:+: m2)` définit le média temporisé obtenu en synchronisant la fin de `m1` avec le début de `m2`.

Pourtant, avec Haskell dont le mécanisme d'évaluation paresseuse permet de définir des flux média temporisés potentiellement infinis, le produit séquentiel `(m1 :+: m2)` peut ne pas être défini. En effet, dès que le flux média `m1` est de durée infinie, le flux média `m2` ne sera jamais joué. *A priori*, il s'agira là d'une

faute de conception qui, pourtant, ne pourra pas être détectée [1].

La sémantique du produit parallèle `(:=:)` est encore plus délicate. *A priori*, plusieurs sémantiques naturelles sont envisageables. En effet, sous l'hypothèse que les durées des deux médias sont distinctes, au moins quatre sémantiques temporelles différentes de `(m1 :=: m2)` sembleraient acceptables :

1. `m1` et `m2` sont lancés en même temps, et le produit s'achève lorsque le plus long média termine. C'est cette sémantique qui est choisie dans la bibliothèque Euterpea [10] évoquée par la suite ;
2. `m1` et `m2` sont lancés en même temps, et le produit s'achève lorsque le plus court média termine, en tronquant le plus long média ;
3. `m1` et `m2` sont synchronisés selon leur milieu ; le plus court des deux médias est décalé de la moitié de la différence des deux durées par rapport au début du plus long ;
4. on refuse ce cas : on n'accepte de jouer en parallèle que des médias de même durée.

De façon plus critique, quelle que soit la sémantique choisie pour `(:=:)`, les points de synchronisation pour des *Polymorphic Temporal Media* sont soit le début et la fin du média considéré (premier et deuxième cas), soit le milieu de celui-ci (troisième cas). Or, ceci s'avère trop rigide comme illustré par la modélisation des *anacrouses* en musique [15].

Plus précisément, une anacrouse est une courte séquence musicale qui précède le début logique d'une séquence plus longue. En supposant que `mA` possède une telle anacrouse, dans un produit séquentiel de la forme `(m :+: mA)` où `m` est une unique mesure de musique, on souhaite que les premières notes de `mA`, les notes en anacrouse, se superposent avec la fin de la mesure `m`. Modéliser ce phénomène avec les *Polymorphic Temporal Media* se révèle particulièrement fastidieux et le résultat éventuel n'est en aucun cas compositionnel.

L'analyse de ce problème [14], via la modélisation formelle des superpositions partielles, nous a conduit à distinguer le contenu des séquences musicales de la façon dont elles sont synchronisées les unes avec les autres. Le modèle résultant [12, 13] est celui des *Tiled Polymorphic Temporal Media* dont il est question par la suite. Comme nous le verrons, il permet de remplacer le produit séquentiel et le produit parallèle par un seul opérateur : le produit tuilé.

## 2.2 Le tuilage en Euterpea

Euterpea [10] est une bibliothèque Haskell développée par Hudak *et al.* pour des applications musicales. Elle implémente, via le type `Music a`, le modèle des *Polymorphic Temporal Media*. Elle est réalisée en Haskell sur une base de temps indexée sur $\mathbb{Q}$, c'est à dire, le type **Rational**. Elle offre, tout à la fois, une base d'expérimentation pour ce modèle, un outil pour son enseignement et un environnement complet pour la composition et l'analyse musicales en Haskell

1. La détermination du caractère infini ou non d'un flux est, en toute généralité, indécidable.

(voir le manuel [10] pour plus de détails). Par extension, elle permet aussi de réaliser une première mise en œuvre du paradigme de programmation temporelle par tuilage [12, 13].

Plus précisément, les *Polymorphic Temporal Media tuilés* sont définis simplement par l'enrichissement des flux musicaux d'Euterpea à l'aide de deux points de synchronisation *Pre* et *Post*. Un exemple de flux tuilé est décrit Figure 1. En

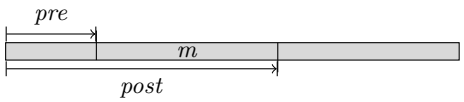


FIGURE 1 – Un flux musical tuilé

Haskell – et selon l'implémentation d'Euterpea –, ces tuiles sont définies comme le type de données **Tile** `a`, qui étend le type `Music a` de la façon suivante :

```
data Tile a = Tile {pre :: Rational,
                    post :: Rational,
                    mus :: Music a}
```

les deux constructeurs `pre` et `post`, de type **Rational**, permettent de localiser les points de synchronisation par leur distance (positive) par rapport au début du flux musical `mus`.

L'ensemble des *Tiled Polymorphic Temporal Media* est muni d'un produit : le *produit tuilé*. Celui-ci est essentiellement défini par deux étapes :

1. *synchronisation* du marqueur Post de la première tuile sur le marqueur Pre de la seconde ;
2. *fusion* des flux tuilés ainsi positionnés.

Un exemple de produit tuilé est décrit Figure 2. Noté %, ce produit est implémenté en Haskell de la façon suivante.

```
(%) :: Tile a -> Tile a -> Tile a
Tile pr1 po1 m1 % Tile pr2 po2 m2 =
    let d = po1 - pr2
    in Tile (max pr1 (pr1 - d)) (max po2 (po2 + d))
          (if d > 0 then m1 :=: mDelay d m2
           else mDelay (-d) m1 :=: m2 )
    where mDelay d m = (case signum d of
                          -1 -> m :+: r (-d)
                           0 ->m
                          +1 -> r d :+: m)
```

où `r d` est une tuile de silence de durée positive `d`.

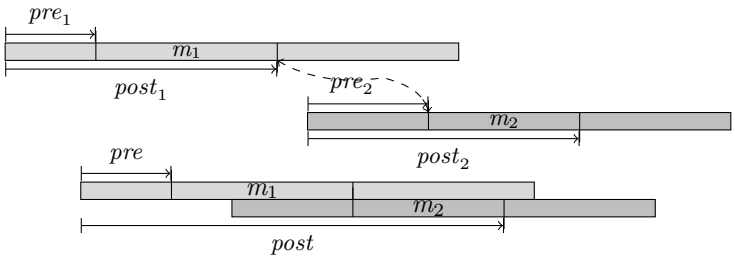


FIGURE 2 – Un exemple de produit tuilé

Dans cette implémentation, la synchronisation est assurée par la fonction `mDelay` qui, par insertion d'un silence au moyen du produit séquentiel `(:+:)`, permet d'aligner les deux flux sous-jacents de façon adéquate. La fusion est alors assurée par le produit parallèle `(:=:)` des flux résultants.

L'opérateur de produit tuilé possède des propriétés algébriques intéressantes. Plus précisément, on dit que deux flux média $m_1$ et $m_2$ sont *observationnellement équivalents*, ce que l'on note $m_1 \equiv m_2$, dès lors qu'ils se jouent de la même façon. Ainsi, dès lors que, pour tout flux média `m` on a l'équivalence

$$\texttt{m} \equiv \texttt{(m :=: m)}$$

c'est-à-dire, dès lors que le produit parallèle est idempotent, alors pour tout flux tuilé `t` il existe, modulo équivalence observationnelle, un unique flux `(inv t)`, nommé inverse de semi-groupe [17] du flux `t`, tel que

$$\begin{aligned} \texttt{t} &\equiv \texttt{t \% (inv t) \% t} \\ \texttt{(inv t)} &\equiv \texttt{(inv t) \% t \% (inv t)} \end{aligned}$$

l'équivalence observationnelle étant étendue aux flux média tuilés (voir [12,13]). Autrement dit, les tuiles forment un monoïde inversif [17].

En effet, on vérifie facilement que, dès lors que le produit parallèle est idempotent, la tuile `(inv t)` définie à partir de `t` en intervertissant `pre` et `pos` satisfait bien les équivalences ci-dessus. Il suffit alors de prouver (voir [17]) que le produit de tuiles idempotentes commute pour garantir l'unicité de cet inverse, ce qui, dans ce cas, est immédiat puisque *Pre* et *Pos* coïncident sur les idempotents.

L'opérateur inverse `inv` ainsi que les opérateurs de reset `re` et de co-reset `co` qu'il induit lorsque combiné avec le produit tuilé sont directement définis en Haskell de la façon suivante :

```
inv, re, co :: Tile a -> Tile a
inv (Tile pre post m) = Tile post pre m
re (Tile pre post m)  = Tile pre pre m
co (Tile pre post m)  = Tile post post m
```

Ils sont illustrés Figure 3.

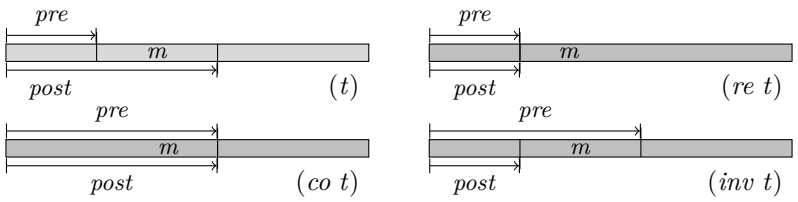


FIGURE 3 – Reset, co-reset et inverse d'une tuile [12, 13]

Des opérateurs primitifs définis ci-dessus, on peut déduire de nombreuses autres fonctions qui peuvent être exportées à l'utilisateur final [2].

Par exemple, la nature tout à la fois séquentielle et parallèle du produit tuilé permet de définir les opérateurs binaires *fork* et *join*. Le fork permet d'aligner deux tuiles sur leur $Pre$, c'est-à-dire de les faire démarrer en même temps. Le join permet d'aligner deux tuiles sur leur $Pos$, c'est-à-dire de les faire terminer en même temps. En Haskell, ils se définissent simplement par :

```
join, fork :: Tile a -> Tile a -> Tile a
join t1 t2 = t1 % co t2
fork t1 t2 = re t1 % t2
```

Ils sont illustrés Figure 4.

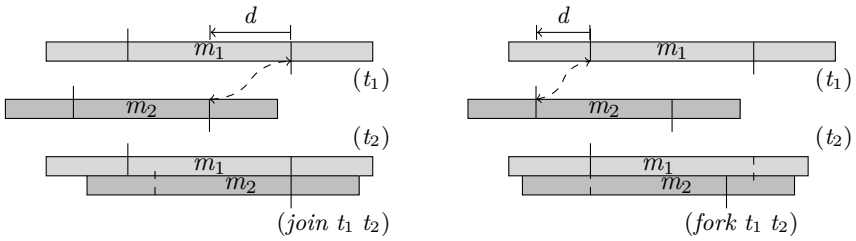


FIGURE 4 – *fork* et *join* de deux tuiles

Le tuilage apparaît donc comme une solution pertinente aux questions soulevées précédemment. Tout d'abord, il est construit autour d'un unique opérateur, et non plus des deux opérateurs de produit séquentiel et produit parallèle. Plus encore, le produit tuilé, contrairement au produit parallèle, possède une sémantique naturelle unique, ce qui simplifie et clarifie son étude.

En outre, par la séparation entre le contenu effectif des tuiles et les marqueurs logiques de synchronisation, les questions de superposition sont résolues de manière transparente à l'utilisateur : il suffit maintenant de tuiler correctement les

2. Dont on n'attend certainement pas qu'il soit un spécialiste de la théorie des semi-groupes !

blocs élémentaires avec lesquels on travaille, la correction des superpositions est assurée par le produit tuilé.

### 2.3 Limites de l'implémentation actuelle

L'implémentation décrite ci-dessus repose cependant sur les constructeurs de base des *Polymorphic Temporal Media*. Elle ne place pas le produit tuilé au centre des types déclarés, mais l'exprime plutôt comme une combinaison du produit séquentiel et du produit parallèle. Même si ces deux opérateurs peuvent être masqués à l'utilisateur, le produit tuilé hérite de leurs propriétés.

L'implémentation des *Polymorphic Temporal Media* décrite dans [8] est pleinement polymorphe et tire efficacement profit du système de classes de types d'Haskell. *A contrario*, le type **Tile** `a` décrit ci-dessus n'est pas réellement polymorphe, car basé sur le type `Music a`. Le silence, dont ce type fait usage, est une notion musicale et, en particulier, la vidéo par exemple n'est pas prise en charge.

Plus ennuyeux : la propriété d'idempotence de tous les flux par produit parallèle n'est pas satisfaite dans l'implémentation actuelle du type `Music a`. En effet, en Euterpea, jouer en parallèle le même morceau deux fois provoque une augmentation du volume, l'équivalence observationnelle est donc violée. À proprement parler, l'implémentation de **Tile** `a` ne définit pas un monoïde inversif.

Reformulée sur les tuiles elles-mêmes, une propriété suffisante pour qu'une implémentation des *Tiled Polymorphic Temporal Media* induise un monoïde inversif est la *loi d'impotence* : en appelant *tuile de contexte* toute tuile $t$ vérifiant $Pre(t) = Pos(t)$, alors une tuile $t$ est idempotente si et seulement si $t$ est une tuile de contexte.

La nouvelle implémentation proposée dans la section suivante est non seulement polymorphe, mais elle satisfait aussi cette loi d'idempotence.

## 3 Une nouvelle implémentation

Nous présentons ici notre implémentation des *Tiled Polymorphic Temporal Media*. Elle est primitive au sens où elle est réalisée indépendamment de toute implémentation des *Polymorphic Temporal Media* [8]. Elle s'inscrit néanmoins dans la lignée de ces dernières. En effet, les tuiles considérées sont engendrées à partir de signaux primitifs de durées constantes [3], de décalages temporels et de produits tuilés. Cette restriction à des signaux de durées connues nous permet d'identifier chaque signal primitif à un événement instantané : son lancement, suivi d'un décalage temporel : le délai correspondant à sa durée.

Syntaxiquement, ces expressions de tuiles peuvent être vues comme des descriptions en zigzag – via des décalages temporels positifs ou négatifs – de séquences temporisées d'événements instantanés. C'est l'implémentation qui est

3. c'est à dire définies dès le début du signal

suivie autour d'un nouveau type de données représentant l'arrière-plan des tuiles, **DList** `time a`, dont on détaille les constructeurs Section 3.2.

Donner une sémantique à ces tuiles, c'est-à-dire les jouer, nécessite d'ordonner les événements qu'elles contiennent dans le temps : les événements identiques lancés en même temps apparaissent alors naturellement. La propriété d'idempotence est ensuite simplement réalisée par suppression de ces doublons.

Plus précisément, l'ordonnancement du contenu des tuiles, nécessaire dans ce nouveau codage, primitif, est réalisé en deux temps. Une première étape, concomitante à la construction des **DList**, consiste à construire le profil temporel, ou **Tag**, de ces listes en zigzag. Ces **Tag** sont présentés Section 3.1. Ils sont au cœur de la sémantique du tuilage puisqu'ils constituent une sorte de typage temporel qui conditionne la synchronisation lors du produit tuilé.

La seconde étape, qui permet d'extraire de façon ordonnée les événements apparaissant dans les tuiles, est présentée Section 3.3. Cette extraction, réalisée dans le langage même des **DList**, est présentée comme une normalisation des tuiles. On constate en effet que deux tuiles ont même sémantique observationnelle lorsqu'elles ont même forme normale. On retrouve ici l'analogie classique entre exécution et normalisation qui apparaît déjà en sémantique du $\lambda$-calcul (voir par exemple [7]).

Dans ce codage, le profil temporel de chaque tuile, pré-calculé, indique la position des premiers événements à jouer. Il permet ainsi de guider, à la volée, le processus de normalisation. Immédiate dans le cas des tuiles finies, cette caractéristique prend tous son sens, via le mécanisme d'évaluation paresseuse de Haskell, dans le cas de l'exécution, c'est à dire de l'ordonnancement, des tuiles infinies. [4]

## 3.1 Profils temporels : le type Tag

Dans l'implémentation précédente [12, 13], le profil temporel d'une tuile `t` est implicitement constitué des positions relatives `pre t` et `post t` des points de synchronisation antérieur et postérieur par rapport au début du flux média, dont on peut déduire la durée de synchronisation :

```
dur :: Tile a -> Rational
dur t = (post t) - (pre t)
```

Dans l'implémentation présentée ici, afin de détecter la présence et la position des premiers événements dans une tuile, on explicite et implémente les profils temporels de la façon suivante :

```
data Tag time = Tag {getDur :: time,
                     getStart :: Maybe time}
```

où le type temporel `time` est un groupe additif totalement ordonné dont le neutre est noté **mzero**, l'addition est notée **mplus**, l'inverse `minverse` et le minimum **min**. On traduit cela par une classe de types **OrdGrp** des groupes ordonnés.

4. Bien sûr, en toute généralité, le profil temporel d'une tuile infinie n'est pas calculable ; voir [13] pour une discussion sur ce sujet.

Avant même de définir les tuiles elles-mêmes, nous pouvons définir les générateurs de leurs profils temporels.

```
tagUnit, tagEvent :: OrdGrp time => Tag time
tagUnit  = Tag mzero Nothing
tagEvent = Tag mzero (Just mzero)

tagDelay :: OrdGrp time => time -> Tag time
tagDelay dur = Tag dur Nothing

tagProd :: OrdGrp time => Tag time -> Tag time -> Tag time
tagProd (Tag dur1 start1) (Tag dur2 start2) = Tag (mplus dur1 dur2)
    (case (start1, start2) of
      (Nothing, Nothing)   -> Nothing
      (Just st, Nothing)   -> Just st
      (Nothing, Just st)   -> Just (mplus st dur1)
      (Just st1, Just st2) -> Just (min st1 (mplus st2 dur1)))
```

Notre intention derrière cette structure de données est la suivante. La valeur `getDur`, de type `time`, est, comme dans la version précédente, la distance relative du marqueur de synchronisation `pre` au marqueur de synchronisation `post`. La valeur `getStart` indique quant à elle la distance relative du marqueur de synchronisation `pre` aux premiers événements de la tuile : de type **Maybe** `time`, elle vaudra **Just** `d` lorsque cette distance est `d` ; fait nouveau – et intuitif –, elle vaudra **Nothing** lorsque la tuile ne contient aucun événement.

On peut vérifier que les éléments de type **Tag** `time`, engendrés par les opérateurs ci-dessus, forment un monoïde inversif de neutre `tagUnit`, de produit `tagProd`, et dont l'inverse est donné par

```
tagInv :: OrdGrp time => Tag time -> Tag time
tagInv (Tag dur start) =
              let newStart = case (start) of
                    Nothing -> Nothing
                    Just st -> Just (mplus st (minverse dur))
              in Tag (minverse dur) newStart
```

## 3.2 Zigzags temporels : le type DList

On présente ici le type **DList** `time a`, qui engendre des séquences d'événements de type `a` combinés par des décalages (ou re-synchronisations) temporels, positifs ou négatifs, et des produits tuilés. Ce type de données pourrait certes être implémenté par de simples listes composées d'événements ou de décalages. Néanmoins, le produit tuilé restant un constructeur primitif, nous préférons un codage syntaxique. La normalisation de ces listes en zigzag, qui coderont nos tuiles, est bien plus spécifique que celle des listes.Elle est réalisée dans la partie

suivante. La structure (récursive) des objets de type **DList** est la suivante :

```
data DList time a = Event a
                  | Sync time
                  | Prod (Tag time) (DList time a) (DList time a)
```

où le constructeur **Event** encapsule les événements de type `a`, le constructeur **Sync** permet de définir des décalage de durée positive ou négative, et **Prod** permet de définir des produits tuilés. Remarquons la présence du profil temporel, de type **Tag** `time`, dans le constructeur **Prod** qu'il conviendra de calculer lors de la production de produit. Les profils temporels des autres constructeurs, calculables en temps constant, sont laissés implicites. Notons aussi le constructeur **DList** prend en argument en particulier le type `time` de l'échelle de temps utilisée : cet arrière-plan pour les tuiles est bien générique dans sa gestion des profils temporels. Les générateurs de **DList** sont définis de la façon suivante :

```
sync :: time -> DList time a
sync = Sync

make :: a -> DList time a
make = Event

(+++) :: (OrdGrp time, Eq a) =>
            DList time a -> DList time a -> DList time a
(+++) t1 t2 = case (t1, t2) of
    (Sync mzero, _) -> t2
    (_, Sync mzero) -> t1
    otherwise -> let newTag = tagProd (getTag t1) (getTag t2)
                 in case (getStart newTag) of
                   Nothing -> Sync (getDur newTag)
                   otherwise -> Prod newTag t1 t2
```

avec la fonction `getTag` définie par

```
getTag :: OrdGrp time => DList time a -> Tag time
getTag (Sync d)       = tagDelay d
getTag (Event _)      = tagEvent
getTag (Prod tag _ _) = tag
```

Le calcul du profil temporel des **DList** est réalisé pas à pas lors du calcul du produit de `Dlist`, noté `(+++)`. Remarquons aussi, dans le code de ce même produit, la simplification immédiate qui est effectuée en l'absence d'événement, c'est-à-dire lorsque `getStart newTag` vaut **Nothing**.

À proprement parler, les **DList** sont bien entendu des structures arborescentes. Néanmoins, modulo associativité du produit,elles se représentent de fa-

çon pertinentes sous la forme de zigzags temporels comme l'illustre l'exemple suivant :

```
ex = sync (-2) +++ make 'a' +++ sync 4 +++ make 'b'
                           +++ sync (-3) +++ make 'c' +++ sync 2
```

qui peut être décrit par le zigzag représenté Figure 5. Sur cette figure, les évènements sont représentés par des noeuds étiquetés, et les décalages temporels sont représentés par des flèches étiquetées par des durées. On peut aussi vérifier qu'on a bien `getTag ex` qui vaut **`Tag (Just -2) 1`**.

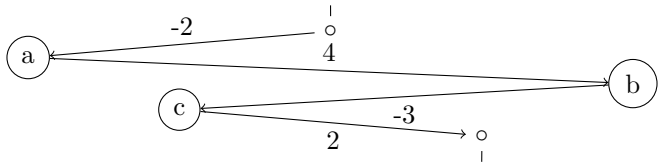


FIGURE 5 – Représentation Zigzag de la **DList** `ex`.

Les autres opérateurs que l'on peut vouloir définir sur les tuiles (voir Section 2.2) se décrivent facilement au niveau des **DList**. Par exemple, on peut définir le reset de la façon suivante :

```
re :: (OrdGrp time, Eq a) => DList time a -> DList time a
re t = let dur = getDur (getTag t)
           backDelay = sync (minverse dur)
       in t +++ backDelay
```

Le code de la fonction `reset` peut être expliqué de la façon suivante. On effectue un produit à droite par un délai négatif de durée l'opposé de la durée de la tuile initiale, *i.e.* une insertion en fin de liste. Ceci signifie qu'une fois toute la tuile initiale parcourue, on est comme attendu sur le marqueur de sortie d'origine, et l'on recule alors de la durée de la tuile, de telle sorte que `pre t` est maintenant égal à `pos t` dans la tuile résultante `t`.

Le co-reset et l'inverse sont implémentés de façon analogue, le premier à l'aide d'un produit à gauche par un délai, le second par deux produits : un à gauche et un à droite. Toutes ces opérations s'effectuent donc aussi en temps constant.

Remarquons cependant que les tuiles construites à partir de **DList** arbitraires ne sont pas immédiatement utilisables. En effet, l'information y est intriquée. Plus précisément, des événements situés au même instant dans l'échelle de temps peuvent se trouver dans des zones très éloignées de l'arbre syntaxique. La procédure de normalisation présentée dans la section suivante permet de remédier à cela.

## 3.3 Normalization des zigzags : fonctions *head* et *tail*

On détaille dans cette section la procédure de normalisation (ou désintrication) des tuiles codées par les **DList**. Le but est de permettre le rendu des tuiles de façon efficace et utilisable en contexte scénique. À cet effet, on récupère pas à pas les événements contenus dans la tuile à travers une fonction `headTail` qui extrait de son argument l'ensemble des événements situés à la date minimale dans la tuile (*i.e.*, les premiers événements). Cette partie est désignée comme la *tête* de la **DList**, on la renvoie accompagnée de la tuile contenant les événements résiduels, la *queue*. La normalisation est implémentée au niveau des **DList**.

Plus en détail, on vise à implémenter la fonction `headTail` dont le type est le suivant :

```
headTail :: (OrdGrp time, Ord a) =>
              DList time a -> (DList time a, DList time a)
```

de telle sorte que la paire renvoyée `(headTail t == (head t, tail t))` vérifie l'invariant dit par la suite *de reconstruction* (où `===` est l'équivalence observationnelle) :

```
t === head t +++ tail t
```

Cet invariant assure que la sémantique des tuiles est bien maintenue (modulo équivalence observationnelle) au cours de la normalisation. Il permet la reconstruction des tuiles après découpage en tête et queue. On maintient aussi l'invariant :

```
start (head t) == start t
```

ce qui signifie que le marqueur d'entrée de la tête `head t` est positionné de la même façon que le marqueur d'entrée de la tuile `t` par rapport aux premiers événements.

La normalisation consiste alors à appeler récursivement la fonction `headTail` sur `tail t` tant que la queue contient des événements médias, la tuile normalisée étant construite au fur et à mesure en utilisant l'invariant de reconstruction énoncé ci-dessus.

```
normalize :: (OrdGrp time, Ord a) => DList time a -> DList time a
normalize l = let (head, tail) = headTail l in
 case (tail) of
   Sync mzero -> head
   otherwise  -> head +++ (normalize tail)
```

Cette procédure de normalisation est illustrée Figure 6 où est représenté le résultat de la normalisation sur l'exemple donné Figure 5, ainsi que la *tête* et la *queue* de ce même exemple. Plus en détail, une tuile normalisée apparaît comme un arbre filiforme composé d'une suite d'événements média et de délais strictement positifs – on parcourt les événements dans l'ordre chronologique, avec d'éventuels délais négatifs au début et à la toute fin de la tuile pour gérer le positionnement des marqueurs d'entrée et de sortie.

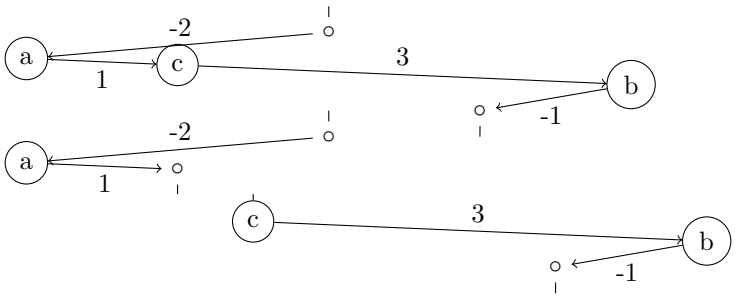


FIGURE 6 – Représentations normalisées des **DList** `ex`, `head ex` et `tail ex`.

En termes d'implémentation, la fonction `headTail` consiste simplement en un appel à une fonction auxiliaire `headTail_tuple` qui, elle, est la fonction véritablement importante dans le processus de normalisation. Elle est typée de la façon suivante :

```
headTail_tuple :: (OrdGrp time, Ord a) =>
                  DList time a -> HeadTail time a
```

où

```
data HeadTail time a = HeadTail
  {toEvents :: time,         -- Delai jusqu'aux evenements de tete
   events   :: Set.Set a,    -- Evenements de tete
   toNext   :: time,         -- Delai jusqu'aux evenements suivants
   tailT    :: DList time a -- Queue
   }
```

La fonction `headTail` est alors implémentée de la façon suivante :

```
headTail t =
    let HeadTail toEvents events toNext tailT = headTail_tuple t
    in ((sync toEvents) +++ fromList (Set.toList events)
        +++ (sync toNext),
       tailT)
```

où `fromList :: (Eq a, OrdGrp time) =>[a] -> DList time a` prend une liste d'événements médias en entrée et retourne la **DList** contenant ces événements lancés en simultané.

La correction (partielle) est assurée par la loi de reconstruction dont on vérifie facilement la préservation. La terminaison, quant à elle, provient du fait que la valeur `toNext` reste strictement positive tant qu'elle est définie, c'est-à-dire tant qu'il y a des événements dans la queue. En effet, on en déduit facilement la décroissance stricte de la largeur de l'intervalle contenant les événements de `tailT`. Sous l'hypothèse que l'ordre sur le groupe `time` est *bien fondé*, on aboutit donc ultimement à une queue ne contenant aucun événement.

Pour ce qui est de la complexité, remarquons que la fonction `headTail_tuple` est naturellement guidée par les profils de synchronisation. Après une initialisation adéquate sur les **DList** primitives, le cas récursif, sur les produits tuilés de sous-tuiles, est facilement guidé par la comparaison des dates des premiers événements de ces sous-tuiles, dates qui sont elles-mêmes aisément calculables à partir des valeurs `getStart` et `getDur` contenues dans les profils temporels. La fonction `headTail_tuple` peut ainsi collecter les premiers événements des tuiles complexes en ne visitant que les branches qui les contiennent.

La complexité temporelle d'un appel `headTail_tuple t` est donc en

$$\mathcal{O}(h(t)f(t))$$

pour $h(t)$ la profondeur des premiers (au sens temporel) événements de tête de la tuile $t$, et $f(t)$ le nombre de ces événements de tête. On vérifie facilement que la complexité temporelle de `normalize t` est en

$$\mathcal{O}(h(t)n(t))$$

pour $h(t)$ la hauteur de la tuile $t$, et $n(t)$ le nombre des événements contenus dans la tuile $t$. Les tuiles sans événements sont en effet agglomérées directement lors du calcul du produit.

## 3.4 Implémentation de Tile a

La notion de **DList** présentée ci-dessus nous offre un arrière-plan effectif et efficace pour le codage des tuiles et la réalisation des opérations élémentaires de construction sur celles-ci. Plus précisément, une tuile est définit comme étant l'encapsulation d'une **DList**, dans laquelle on exige la cohérence du type des événements et du type des durées. Pour cela, on définit la classe de types **Timed** `a` par :

```
class Timed a where
  type Time a :: *         -- Echelle de induite par  a
  duration :: a -> Time a -- Duree d'un media elementaire
```

On peut alors (re)définir le constructeur **Tile** par :

```
data Tile a = Tile (DList (Time a) a)
```

avec le produit tuilé simplement défini par :

```
Tile l1 % Tile l2 = Tile (l1 +++ l2)
```

Pour une compatibilité ascendante, on peut aussi redéfinir les constructeurs `pre` et `post` par :

```
pre (Tile dl) = case (getTag dl) of
  (Tag dur Nothing)   -> (max mzero (minverse dur))
  (Tag dur (Just st)) -> (max (max mzero (minverse dur))
                              (minverse st))
```

et

```
post (Tile dl) = let (Tag dur _) = (getTag dl)
                 in mplus (pre dl) dur
```

La date de référence pour servir d'origine locale est définie ici par la date de l'évènement le plus tôt parmi : le marqueur de synchronization *Pre*, le marqueur de synchronization *Post* ou, dans le cas où il existe, le premier évènement.

Apparemment, le codage [12, 13] particulièrement simple des TPTM au dessus des PTM semble témoigner du caractère primitif de l'algèbre induite par la définition des *Polymorphic Temporal Media*. Mais l'inverse est tout aussi vrai. L'implémentation primitive des tuiles que nous proposons ici permet, tout aussi facilement, de coder les *Polymorphic Temporal Media*.

En effet, rappelons que les PTM se définissent en toute généralité à partir d'objets média primitifs de type `a` dont on peut mesurer la durée, c'est à dire instance de **Timed**. Ces objets média sont alors combinés par produits séquentiels et parallèles. Autrement dit, on peut définir la classe de types **TMedia** `m` par :

```
class TMedia m where
  none    :: (Timed a, OrdGrp (Time a))       => Time a -> m a
  prim    :: (Eq a, Timed a, OrdGrp (Time a)) => a -> m a
  dur     :: (Timed a, OrdGrp (Time a))       => m a -> Time a
  seqProd :: (Eq a, Timed a, OrdGrp (Time a)) => m a -> m a -> m a
  parProd :: (Eq a, Timed a, OrdGrp (Time a)) => m a -> m a -> m a
```

avec

```
instance TMedia Tile where
  none dur      = Tile (sync dur)
  prim a        = Tile ((make a) +++ (sync (duration a)))
  dur (Tile dl)  = getDur (getTag dl)
  seqProd       = (%)
  parProd t1 t2 = let reset (Tile dl) = Tile (re dl) in
    case (dur m1 <= dur m2) of
      True  -> (reset m1) % m2
      False -> (reset m2) % m1
```

Notons que `(make a)` traite le media `a` comme un évènement instantané, ce qui peut sembler curieux lorsque `(duration a)` est non nul. Il faut comprendre `(make a)` comme le lancement de l'évènement `a`, la tuile `(sync (duration a))` simulant alors la durée (constante) du media `a`.

La procédure de normalisation nous permet alors de définir une égalité sémantique :

```
instance (Ord a, OrdGrp (Time a)) => Eq (Tile a) where
  (Tile (Sync d1)) == (Tile (Sync d2)) = d1 == d2
  (Tile l1) == (Tile l2) =
```

```
let (HeadTail te1 e1 tn1 t1) = headTail_tuple l1
    (HeadTail te2 e2 tn2 t2) = headTail_tuple l2
in (te1 == te2) && (e1 == e2)
   && (tn1 == tn2) && (Tile t1 == Tile t2)
```

On vérifie alors facilement que, dans le cas fini, l'équivalence `==`, satisfait, sur notre codage des *Polymorphic Temporal Media*, tous les axiomes décrits dans [9]. Autrement dit, nous avons bien là un codage complet et cohérent des *Polymorphic Temporal Media*.

Pour finir, nous pouvons mentionner que la forme normale proposée dans la section précédente apparaît comme une généralisation au cas des tuiles de la notion de forme normale des *Polymorphic Temporal Media* déjà décrite dans [8].

# 4 Bilan et perspectives

Cette nouvelle implémentation des tuiles est véritablement polymorphe. Elle offre une interface transparente et efficace à l'utilisateur, qui peut déclarer ses propres types de médias temporisés, et les instancier – grâce aux classes de types développées – comme médias tuilés. L'interface exportée, le type **Tile** `a`, permettent en outre de ne travailler que sur la représentation abstraite, algébrique des tuiles. L'implémentation sous-forme de **DList** est masquée à l'utilisateur, ce qui permet au programmeur de mettre à jour indépendamment l'implémentation effective et la signature exportée pour le module.

La fonction de normalisation réalise la sémantique. *A priori*, la gestion des tuiles infinies présentées dans [12,13] s'adapte sans difficulté à cette nouvelle implémentation. La procédure de normalisation à la volée, couplée à l'évaluation paresseuse d'Haskell, devrait permettre le rendu de ces tuiles. Bien entendu, pour une utilisation à la volée effective, il faut implémenter les fonctions d'exécution des événements primitifs. Dans la version testée, la fonction `play` de la bibliothèque Euterpea n'est pas optimisée pour le rendu à la volée et les appels introduisent une latence trop importante. L'utilisation en pratique des outils proposés reste donc à développer.

Une expérimentation avec l'utilisateur final, le musicien, devrait permettre de développer une bibliothèque de haut-niveau adaptée à la composition musicale. Naturellement, les structures de données et les primitives proposées ici peuvent faire l'objet d'autres implémentations afin d'être intégrées aux plateformes de composition existantes. Par exemple, une extension en CLOS [4] pour une intégration à la suite logiciel OpenMusic [1] est actuellement à l'étude.

## Remerciements

La présentation des tuiles faite ici a largement bénéficié des nombreux commentaires et des critiques des rapporteurs. Par ailleurs, une revue de code de Paul Hudak, fort pertinente, nous a offert le recul permettant de mettre bien mieux en perspective l'implementation proposée ici avec son propre travail sur

les flux media polymorphes. Bien entendu, les auteurs gardent seuls la responsabilité des maladresses et des erreurs qui pourraient demeurer dans cette présentation.

# Références


[1] C. Agon, J. Bresson, and G.Assayag. *The OM composer's Book, Vol.1 & Vol.2.* Collection Musique/Sciences. Ircam/Delatour, 2006.

[2] F. Berthaut, D. Janin, and M. DeSainteCatherine. libTuile : un moteur d'exécution multi-échelle de processus musicaux hiérarchisés. In *Actes des Journées d'informatique Musicale (JIM)*, 2013.

[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] J. Bresson, C. Agon, and G. Assayag. Visual Lisp / CLOS programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1), 2009.

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

[6] A. Dicky and D. Janin. Embedding finite and infinite words into overlapping tiles. In *Developments in Language Theory (DLT)*, volume 8633 of *LNCS*, pages 339–347, Ekaterinburg, Russia, 2014. Springer.

[7] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, 1989.

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

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

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

[11] 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.

[12] P. Hudak and D. Janin. Programmer avec des tuiles musicales : le t-calcul en euterpea. In *Actes des Journées d'informatique Musicale (JIM)*, Bourges, France, 2014.

[13] P. Hudak and D. Janin. Tiled polymorphic temporal media. In *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*, page (in print), Gothenburg, Sweden, 2014. ACM Press.

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

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

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

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

[18] D. Perrin and J.-E. Pin. *Infinite Words : Automata, Semigroups, Logic and Games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004.

[19] C. Roads. *L'audionumérique*. Dunod, 1998.