Pour un raffinement spatio-temporel tuilé

Simon Archipoff, David Janin

Pour un raffinement spatio-temporel tuilé

Simon Archipoff & David Janin

LaBRI, Université de Bordeaux, Bordeaux INP,
Inria Bordeaux Sud-Ouest
351, cours de la Libération
33 405 Talence cedex

Résumé

Dans le domaine de la construction et de la manipulation par programme de flux média temporisé, nous étudions la problématique du raffinement, c'est-à-dire le problème de la substitution de flux élémentaires par des flux plus complexes. Dans le cas de construction purement séquentielle, la notion de liste permet de faire cela sans difficulté. C'est du raffinement temporel. Le bind de la monade liste permet de réaliser ces substitutions.

En présence de parallélisme, les choses sont plus complexes : aucune construction ne semble s'imposer a priori. Dans cet article, nous montrons comment la modélisation par tuilage, qui explicite dans sa syntaxe les points de synchronisation qui doivent être préservés lors d'un raffinement, permet de définir, sans ambiguïté et tout aussi simplement que dans le cas des listes, un raffinement spatio-temporel.

Incidemment, cette approche rompt avec l'usage usuel des monades en programmation fonctionnelle qui permettent de représenter la séquentialité. Sur la monade des tuiles comme sur la monade des listes, le bind permet aussi de modéliser le raffinement.

1. Introduction

Modélisation et programmation de flux spatio-temporels. Dans cet article, nous nous intéressons à la modélisation et à la programmation de flux temporisés, c'est-à-dire de séquences de valeurs placées dans le temps. L'un des domaines d'application que nous visons tout particulièrement est celui des systèmes multimédia interactifs, dans lequel ces flux décrivent de l'audio, de la musique, de la vidéo, ou même des animations. Dans ce domaine, les flux manipulés ne sont pas seulement temporisés, ils sont aussi spatialisés. Dans le cas de l'audio, on manipule souvent des flux audio multipistes qui doivent être agencés les uns par rapport aux autres aussi bien dans l'espace que dans le temps. En un sens, nous nous intéressons donc aux flux spatio-temporels même si, pour des raisons de simplicité, nous nous contenterons de les appeler flux temporisés.

Nous parlons bien ici, tout à la fois, de modélisation *et* de programmation, puisque l'un des enjeux de cet axe de recherche est de pouvoir manipuler des modèles de flux temporisés dont on puisse avoir une représentation simple tout en disposant d'un langage de programmation dédié qui permette effectivement de les mettre en œuvre.

Dans ce contexte, l'algèbre s'impose naturellement comme outil d'étude de la sémantique de ces langages. Par exemple, Hudak [6] définit la notion de flux média temporisé polymorphe comme modèle sous-jacent au langage applicatif *Euterpea* pour la composition musicale [5, 7], enchâssé dans le langage de programmation fonctionnelle *Haskell* [8]. Dans cette approche, deux opérateurs, la composition

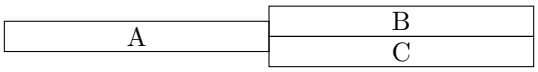


FIGURE 1 – Représentation visuelle d'une composition spatio-temporelle.

parallèles et la composition séquentielle, permettent de composer dans le temps et dans l'espace les flux temporisé. La composition séquentielle d'un flux A avec le résultat de la composition parallèle de deux autres flux B et C est décrite Figure 1.

Dans cette figure, l'axe du temps est implicitement représenté de la gauche vers la droite, l'espace occupé par les mises en parallèle est représenté verticalement. Bien entendu, cette représentation visuelle ne capture pas complètement la sémantique de ces compositions. Par exemple, la composition parallèle est commutative. Autrement dit, la position relative des flux temporisés sur l'axe vertical est arbitraire.

L'approche d'Hudak, entre algèbre et programmation fonctionnelle, peut faire apparaître une correspondance entre la *représentation graphique* des flux temporisés, issue de l'algèbre sous-jacente, et la *représentation programmatique* des mêmes flux, induite par leur codage dans le langage dédié. On le voit bien Figure 1, on peut dessiner les compositions séquentielles et parallèles de flux. Autrement dit, on retrouve l'idée, très ancienne, d'une *programmation visuelle* qui pourrait s'abstraire de la syntaxe pour pouvoir se concentrer sur une sémantique qui peut être visualisée.

D'une certaine façon, l'écriture de la musique occidentale tout comme celle des partitions d'orgues de barbarie sont des exemples de langages de programmation visuelle définies bien avant l'invention des ordinateurs. Cet aspect, qui pourrait sembler banal à un programmeur, est une composante essentielle de la définition du langage *Euterpea* [7] qui vise à ouvrir la pratique de la programmation à de nouveaux publics tels que, tout particulièrement, les compositeurs.

Raffinement de flux spatio-temporel. La question qui nous intéresse plus particulièrement dans cet article est celle du raffinement des flux spatio-temporel.

Considérons une composition de flux élémentaires d'un type donné. Le raffinement des flux de cette composition consiste à y remplacer chaque flux élémentaire par un flux plus complexe, éventuellement d'un autre type. La correspondance entre un flux élémentaire et le flux complexe qui le remplace est fixée arbitrairement par le programmeur.

Cette construction apparaît naturellement en analyse ou en composition musicale. Par exemple, une structure classique en musique populaire est le « $AABA$ ». Elle décrit, dans une approche abstraite, une phrase musicale « A » jouée deux fois, suivie d'une phrase musicale « B » jouée une fois, pour conclure ensuite par une répétition de la phrase initiale « A ». Passer de cette représentation abstraite à une représentation plus concrète revient donc à remplacer A et B par les séquences de notes qui les définissent. Autre exemple, l'interprétation musicale, qui consiste à passer d'une représentation symbolique (les notes) à une exécution (les sons) peut aussi s'apparenter à la substitution de chaque note par le son qui lui correspond.

Dans le cas purement séquentiel, le raffinement ne pose pas de problème. Chaque lettre est remplacée par son raffinement, et le tout est concaténé. il n'y a aucune ambiguïté. Cette situation est illustrée Figure 2 où dans la composition séquentielle de A suivie de B nous remplaçons A par la séquence $f(A) = a_1a_2$ et B par la séquence $f(B) = b_1b_2b_3$. Remarquons qu'une telle substitution

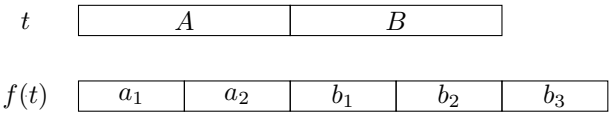


FIGURE 2 – Raffinement séquentiel.

induit un morphisme des chaînes de caractères sur l'alphabet $\{A, B, \cdots\}$ dans les chaînes de caractères sur l'alphabet $\{a_1, a_2, b_1, b_2, b_3, \cdots\}$.

En musique cependant, les choses ne sont pas si simples. Les mélodies associées aux éléments d'un « $AABA$ » peuvent dépasser de leurs intervalles de temps théoriques. C'est par exemple le cas avec l'anacrouse, une sorte d'anticipation mélodique du premier temps fort d'une séquence musicale [12]. Dans ce cas, les mélodies associées aux A et B peuvent se superposer partiellement dans la réalisation du « $AABA$ ». Il y a du parallélisme.

De même, pour l'interprétation, la durée des notes jouées peut ne pas correspondre aux durées théoriques des notes écrites. Dans un jeu *staccato* ces notes peuvent être plus courtes, séparées par des silences. Dans un jeu *legato*, ces notes peuvent être plus longues. Dans ce dernier cas, chaque fin de note recouvre partiellement le début de la note qui suit. Il y a donc à nouveau une certaine forme de parallélisme.

Dans le cas purement séquentiel nous pouvions calculer sans ambiguïté la séquence raffinée. En présence de parallélisme, ce n'est plus le cas. Pour illustrer cette situation, reprenons l'exemple de la Figure 1 dans lequel nous souhaitons raffiner l'élément A par $f(A) = a_1a_2$, B par $f(B) = b_1b_2b_3$ et C par $f(C) = c_1c_2c_3c_4c_5c_6$. Il apparaît, comme illustré Figure 3, qu'au moins trois raffinements sont possibles. En effet, dans le cas (3a) nous supposons implicitement que B et C ont été synchronisés

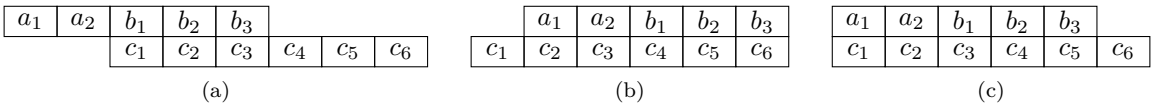


FIGURE 3 – Raffinements en présence de parallélisme.

sur leurs débuts [1], dans le cas (3b) nous supposons implicitement que B et C ont été synchronisés sur leurs fins. Dans le dernier cas (3c), apparemment plus arbitraire, nous supposons que B et C ont été synchronisé sur leur deux tiers. Autrement dit, en présence de parallélisme, le raffinement est ambigu. Lever cette ambiguïté revient à expliciter, au niveau abstrait, la façon dont les éléments sont synchronisés, c'est-à-dire positionnés dans le temps les uns par rapport aux autres.

Plus précisément, dans une composition séquentielle de A suivi de B, la fin de A est implicitement synchronisée avec le début de B. Cette sémantique est visible graphiquement et semble s'imposer d'elle même. Dans le cas de la composition parallèle de B et C, bien que la synchronisation sur les débuts de B et C puisse sembler naturelle, d'autres sémantiques sont possibles. Pourtant, dès lors que B et C ont la même longueur, rien ne permet de distinguer graphiquement la sémantique choisie.

Pour lever cette ambiguïté, il nous faut expliciter des contraintes de synchronisation. Cette situation est illustrée Figure 4 dans laquelle les synchronisations entre B et C (en rouge) mais aussi entre A et B (en bleu) sont rendu explicites. Pour faire cela, nous représentons explicitement la paire de points de A et B qui sont synchronisés et la paire de points de B et C qui le sont aussi. Plus précisément, ces points de synchronisation sont localement définis par leur position relative en

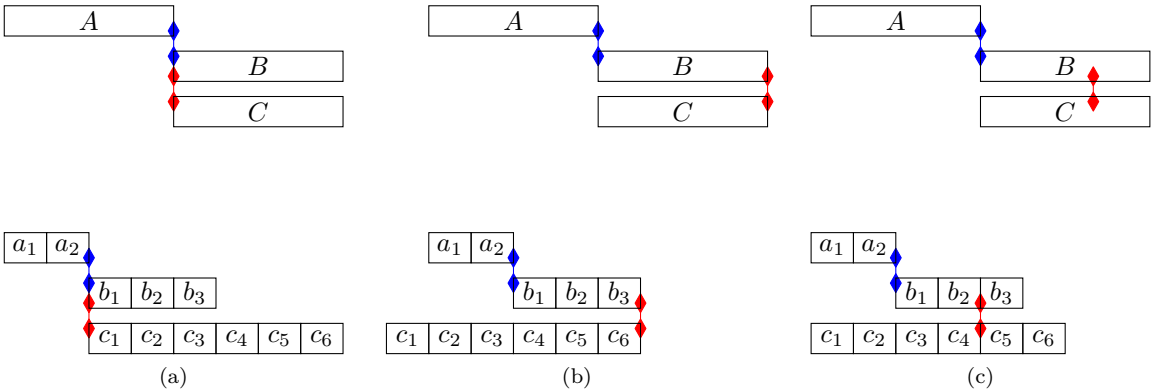


FIGURE 4 – Raffinements en présence de parallélisme avec synchronisations explicites.

fonction du début et de la fin des flux sur lesquelles ils apparaissent. Le raffinement peut alors être défini en préservant ces synchronisations. On peut ainsi reconstruire, sans ambiguïté, les trois cas de raffinement évoqués précédemment.

Ces exemples pourraient sembler arbitraires, la notion de points de synchronisation pouvant être définie de façon ad hoc. L'objet de cet article est de démontrer qu'il n'en est rien. La modélisation par tuilage [4, 3, 13], rappelée ci-dessous, plus générale que la modélisation séquentielle, permet de modéliser ces phénomènes de façon algébrique. Les points de synchronisation qui sont préservés par raffinement dérivent simplement des constructeurs utilisés pour la définition et la composition des modèles tuilés.

1. c'est là une sémantique usuelle du produit parallèle

2. Le modèle des tuiles

Le modèle des tuiles avec superpositions, simplement appelées tuiles dans la suite, permet de représenter des séquences temporisées enrichies de deux marqueurs de synchronisation : un marqueur de “début” noté $\curlyvee$ et un marqueur de “fin” noté $\downarrow$. En un sens, ces marqueurs représentent le début et la fin logique des flux tuilés sous-jacents puisqu'ils peuvent être décorrélés du début et de la fin effective des flux qu'ils marquent.

Le produit tuilé de deux flux ainsi enrichis se définit en effet comme le produit parallèle de ces deux flux positionnés l'un par rapport à l'autre en synchronisant le marqueur de fin du premier composant avec le marqueur de début du second composant. Cette construction générale est décrite Figure 5. C'est ce mécanisme qui va nous permettre de modéliser les exemples précédents.

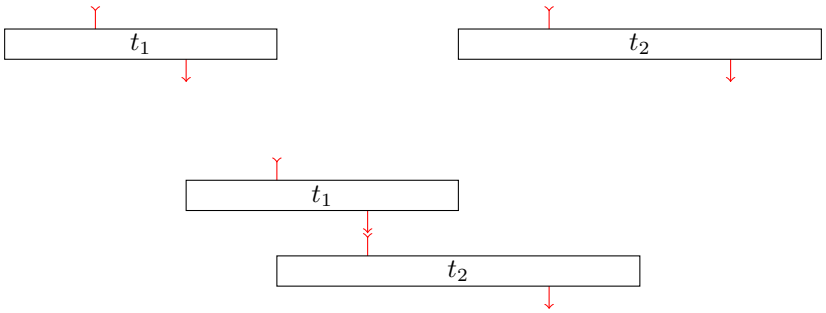


FIGURE 5 – Deux flux tuilés (en haut) et leur produit (en bas).

Dans le cas le plus simple, le modèle des tuiles apparaît comme une extension du modèle des chaînes de caractères. Plus précisément, chaque lettre A est tout d'abord enrichie des deux marqueurs respectivement positionnés au début et à la fin de A. Le mot obtenu peut être représenté par $\curlyvee A\downarrow$. Le produit $\curlyvee A\downarrow\cdot\curlyvee B\downarrow$ de deux lettres ainsi enrichit vaut alors $\curlyvee AB\downarrow$. Les marqueurs intermédiaires ont été effacés. Par extension, on a aussi $\curlyvee AB\downarrow\cdot\curlyvee C\downarrow = \curlyvee ABC\downarrow$, etc. . . , ce qui montre comment les chaînes de caractères sont plongées dans les tuiles.

Le tuilage proprement dit apparaît alors avec les opérations de *reset* et *coreset* qui reviennent, pour la première, à déplacer le marquer de fin sur le marqueur de début et, pour la second, à déplacer le marqueur de début sur le marqueur de fin. On a ainsi $\mathit{reset}(\curlyvee A\downarrow) = \curlyvee\downarrow A$ et $\mathit{coreset}(\curlyvee A\downarrow) = A\curlyvee\downarrow$.

Le produit séquentiel de A avec le produit parallèle de B et C illustré Figure 1 peut ainsi être codé de deux façon. Avec le *reset*, l'expression $\curlyvee A\downarrow\cdot\mathit{reset}(\curlyvee B\downarrow)\cdot\curlyvee C\downarrow$ définit une synchronisation de C avec le début de B comme illustrée Figure 6 dans le cas (6a). Au contraire, avec le *coreset*, l'expression $\curlyvee A\downarrow\cdot\curlyvee B\downarrow\cdot\mathit{coreset}(\curlyvee C\downarrow)$ définit une synchronisation de C avec la fin de B comme illustrée Figure 6 dans le cas (6b). Dans cette figure, on voit apparaître explicitement les synchronisations évoquées

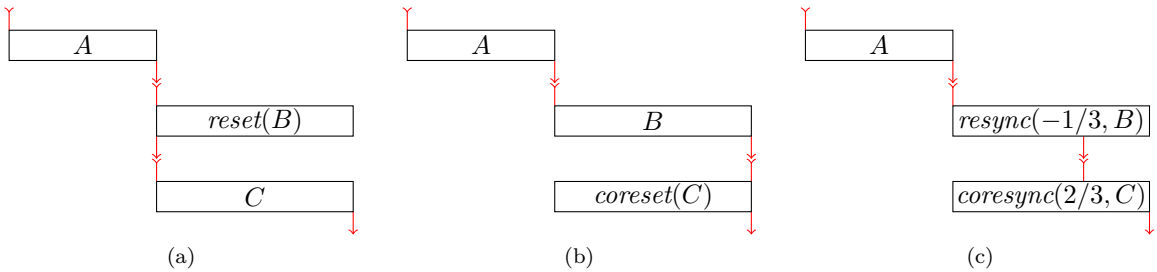


FIGURE 6 – Définition de points de synchronisation par produit tuilé.

ci-dessus, dans les cas (4a) et (4b) de la Figure 4.

Dans le cas le plus général, afin de pouvoir modéliser, en particulier, le cas (4c) de la Figure 4 qui est décrit Figure 6 dans le cas (6c), nous étendons les tuiles à l'aide de temporisation explicite, des opérateurs supplémentaires [4] nous permettant d'étirer leur durée ou de déplacer, de façon

proportionelle, leurs marqueurs de synchronisation. La syntaxe et la sémantique de ces opérateurs sont décrites section suivante.

Remarquons au passage que nous donnons dans cet article une sémantique temporisée aux flux manipulés, c'est-à-dire une sémantique dans laquelle la durée des flux est mesurée. Il apparaît alors deux formes de synchronisation :

— Une *synchronisation explicite* décrite ci-dessus, qui provient du produit tuilé.
— Une *synchronisation induite* qui provient du fait que, dès lors que les synchronisations explicites sont réalisées, d'autre points coïncident dans le temps du simple fait que les flux se déroulent a priori à la même vitesse.

Cette situation est décrite Figure 7 qui illustre le produit $reset(A) \cdot B \cdot C$ dans lequel la durée de A vaut deux fois la durée de B et C. On le devine, le raffinement préservera les synchronisations

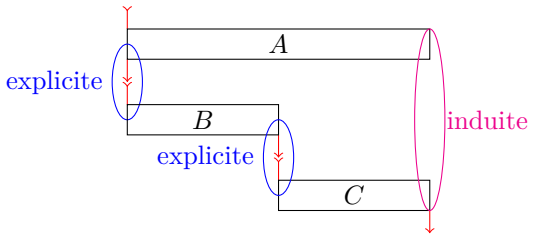


FIGURE 7 – Synchronisation explicite et induite

explicites mais les synchronisations induites pourront être perdues. En effet, avec le raffinement que nous nous proposons de mettre en œuvre, l'échelle de temps associée à chaque flux atomique est implicitement locale puisque, via substitution, elle peut varier d'un élément atomique à un autre.

3. Implémentation des tuiles

L'implémentation des tuiles proposée ici est décrite dans le langage *Haskell*. Bien entendu, d'autres langages de programmation pourraient être utilisés. Le caractère fonctionnel pur du langage *Haskell* permet cependant de rester au plus près de l'algèbre des tuiles, le *DSL* associé au modèle des tuiles conduisant à déclarer un ensemble d'équations de points-fixes qui les définissent [14, 10, 9, 15].

La classe `Tile`

Le modèle des tuiles temporisées se présente avant tout sous la forme d'une classe de type, plusieurs instances de la classe pouvant alors être proposées.

```
class Tile a  where
    type TDur a :: *
    type TVal a :: *
    val         :: (TVal a) -> a
    unit        :: a
    (%)         :: a -> a -> a
    delay       :: Num (TDur a) => (TDur a) -> a
    re          :: Num (TDur a) => a -> a
    co          :: Num (TDur a) => a -> a
    resync      :: Num (TDur a) => (TDur a) -> a -> a
    coresync    :: Num (TDur a) => (TDur a) -> a -> a
    dur         :: Num (TDur a) => a -> (TDur a)
    stretch     :: Num (TDur a) => (TDur a) -> a -> a
    re          =  resync (-1)
    co          =  coresync 1
    delay d     =  stretch d unit
```

Dans cette classe, le type `TDur a` représente le type temporel associé aux tuile de type `a`. Il doit être une instance de la classe `Num`. Le type `TVal a` représente les valeurs qui sont portées et placées dans le temps par les tuiles.

La tuile `unit` représente une valeur vide (silencieuse) tuilée sur une durée 1. La tuile `val v` représente la valeur `v` tuilée sur une durée 1. L'opérateur `%` est le produit tuilé. La fonction `dur` associe à chaque tuile donnée en argument sa « durée » qui est toujours définie comme la distance relative, mesurée en temps, entre le marqueur de « début » et le marqueur de « fin ». Les premiers invariants de la classe `Tile` sont donc :

— `dur unit == dur (val v) == 1`
— `dur (a1 % a2) == (dur a1) + (dur a2)`

La fonction `stretch` permet d'étirer ou de contracter une tuile du facteur passé en argument. Dans cet étirement, les positions des marqueurs de « début » et de « fin » relativement au début réel [2] du flux sous-jacent doivent être préservées. Les équations typiques satisfaites par la fonction `stretch` sont les suivantes :

— `dur (stretch f t) == f * (dur t)`,
— `resync d (stretch f t) == stretch f (resync d t)`
— `coresync d (stretch f t) == stretch f (coresync d t)`

La fonction `delay` construite par étirement ou contraction de la tuile silencieuse `unit` produit donc une tuile silencieuse de la valeur spécifiée.

Les fonctions `resync` et `coresync` permettent respectivement de déplacer les marqueurs de « fin » et de « début », d'une valeur proportionelle à la durée de la tuile passée en argument. Ainsi, déplacer le marqueur de « fin » d'un facteur -1, à l'aide de `resync (-1)` revient à le faire coïncider avec le marqueur de « début ». C'est donc le codage par défaut de la fonction `reset`. De la même manière, déplacer le marqueur de « début » d'un facteur 1, à l'aide de `coresync 1` revient à le faire coïncider avec le marqueur de « fin ». C'est donc le codage par défaut de la fonction `coreset`. Ces fonctions sont illustrées Figure 8.

FIGURE 8 – Sémantique des primitives de Tile.

Remarque. Bien que la fonction `delay` puisse être combinée avec le produit tuilé pour déplacer les marqueurs de synchronisation d'une tuile, la proportionnalité de ces déplacements n'est pas forcément préservée par raffinement contrairement au déplacement induit par les fonctions `resync` et `coresync`. Par exemple, l'expression `a % delay (-(dur a))` (voir Figure 9a) est équivalente à `re a` (Figure 9b). Néanmoins, cette équivalence ne sera pas forcément préservée par tout raffinement. Pour le raffinement de a en `value b % value b`, la tuile représentée Figure 9a' n'est plus équivalente au reset d'une tuile, car `dur a` n'est pas égal à `dur (value b % value b)`.

En revanche, la tuile représentée par la Figure 9b' est toujours le reset de la tuile.

2. Un début réel de flux qui devra toujours être défini, contrairement à sa fin dans le cas de manipulation de flux paresseux potentiellement infini.

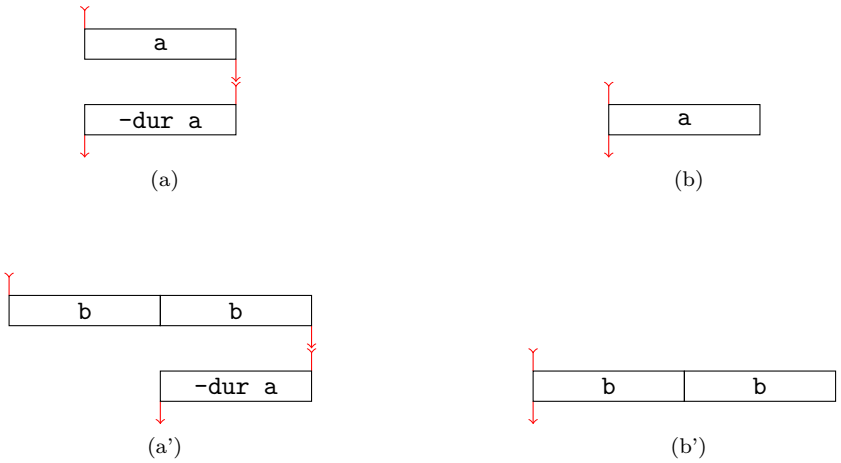


FIGURE 9 – Deux tuiles (a) et (b) équivalentes et leur raffinements (a') et (b').

Instantiation syntaxique de la classe `Tile`

Nous proposons ici un codage syntaxique des tuiles, c'est-à-dire que nous conservons toute l'information, sans même faire d'hypothèse sur l'associativité du produit tuilé :

```
data STile d a = Prod (STile d a) (STile d a)
               | Value a
               | Unit
               | Mod d d d (STile d a)
```

Alors que la sémantique intentionnelle des constructeurs `Unit`, `Value` et `Prod` est claire, la sémantique attendue du constructeur `Mod`, illustrée Figure 10, nécessite quelques explications. La sémantique d'un terme de la forme `Mod f d1 d2` est définie par l'application d'un étirement de facteur `f` sur la tuile argument, puis, de façon simultanée, le déplacement du marqueur de début d'un facteur `d1` et le déplacement du marqueur de fin d'un facteur `d2`.

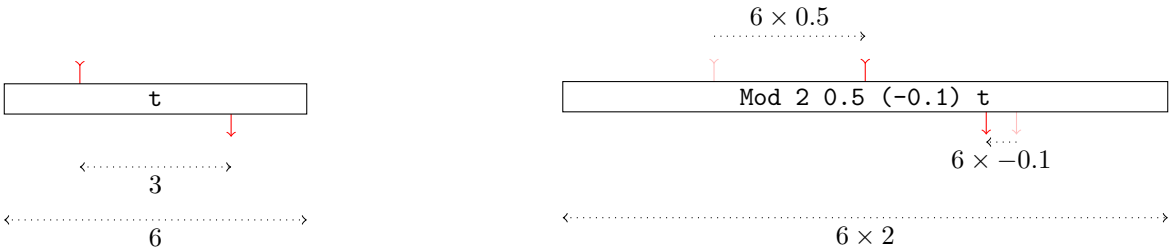


FIGURE 10 – Sémantique de Mod.

Comme primitive d'accès aux structures syntaxiques `STile`, nous définissons principalement `foldT` qui permet de réaliser une réduction de l'arbre en partant des feuilles et paramétrée pour chacun des constructeurs de `STile`. Les paramètres sont identifiés par leur initiale.

```
foldT p v u m t =
  let recFold = foldT p v u m
  in case t of
    (Prod t1 t2) -> p (recFold t1) (recFold t2)
    (Value a) -> v a
    (Unit) -> u
    (Mod f pre_s post_s t) -> m f pre_s post_s (recFold t)
```

Ce type est naturellement un foncteur, nous pouvons définir `fmap` ainsi :

```
instance Functor (STile d) where
  fmap f t = foldT Prod  (Value . f) Unit Mod t
```

Les tuiles syntaxiques forment une instance de la classe `Tile` de la façon suivante :

```
instance Num d => Tile (STile d a) where
    type TDur (STile d a) = d
    type TVal (STile d a) = a
    unit = Unit
    val = Value
    (%) = Prod
    dur t = foldT (+) (const 1) 1 (\f pre post d -> (d*f) * (1 - pre + post)) t
    resync d   = Mod 1 0 d
    coresync d = Mod 1 d 0
    stretch f  = Mod f 0 0
```

les fonctions `re`, `co` et `delay` étant implémentées par défaut.

La fonction `dur` a été implémentée grâce à `foldT` dont les paramètres, listés ci-dessous, méritent quelques explications :

- `(+)` : pour le constructeur `Prod` car si `t = Prod t1 t2`, alors `dur t = dur t1 + dur t2`
- `const 1` : pour le constructeur `Value a`, car par définition la durée est de 1 et on ignore `a` (`const` est la fonction qui prend deux paramètres et retourne le premier, ignorant le second).
- `1` : pour l'unité car la durée de `Unit` est 1 par définition.
- `\f pre post d -> (d*f) * (1 - pre + post)` : pour `Mod f pre post t` sachant que `d = dur t`.

4. La monade

La programmation fonctionnelle, tout particulièrement avec *Haskell* fait usage de monades, pour structurer les programmes et garantir qu'ils vérifient certaines propriétés. Le cas qui a motivé l'apparition des monade en programmation est la gestion des entrées/sorties dans un langage pur [16] via la monade IO. Elle permet tout à la fois de contrôler la portée des effets de bord et de garantir que chaque effet de bord souhaité se produit une et une seule fois et au moment opportun. Nous pouvons citer aussi la monade $Maybe$ qui peut servir à représenter un calcul dont des étapes peuvent échouer : elle garantit que le calcul aboutira si et seulement si toutes les étapes réussissent. Plus généralement, une monade est une abstraction de calculs pouvant être composés. Très concrètement, une monade est formée par :

- un constructeur de type m,
- un opérateur return :: $a \to m\ a$ qui encapsule une valeur de type a grâce à m,
- un opérateur $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$ qui permet la composition de deux monades.

Les fonctions return et $\gg\!=$ vérifient certaines propriétés que nous détaillerons plus tard. La première se comporte comme un neutre, et la seconde est associative.

La monade *liste* définie dans la bibliothèque standard *Haskell* a été une source d'inspiration. Nous pouvons la voir comme la modélisation de calculs non déterministes (une liste représentant l'ensemble des valeurs possibles) mais nous pouvons aussi la voir comme la modélisation du raffinement. Étant donné une fonction qui à chaque élément de la liste associe une autre liste qui le décrit, l'opérateur $\gg\!=$ va calculer la liste qui les décrit tous.

Dans notre cas, les opérateurs monadiques permettent de garantir que les schémas de synchronisation sont préservés lors du raffinement. Par exemple si les débuts de A et B sont synchronisés, alors les débuts de leur raffinement par f le seront également.

Une tuile est codée par un ensemble de valeur synchronisées entre elles par un arbre de points de synchronisation. L'opérateur $\gg\!=$ de la monade substitue ces valeurs par des tuiles, respectant à la fois le temps et la synchronisation.

4.1. Implémentation

L'instantiation de la monade est faite de manière purement syntaxique.

```
instance Num d => Monad (STile d) where
  return = Value
  t >>= f = foldT Prod f Unit Mod t
```

La fonction `return` place son paramètre dans une tuile minimal, grâce à `value`. L'expression `tile >>= f` vaut `tile` où toutes les feuilles `Value x` ont été remplacée par `f x`. Toutes les synchronisations présentes dans `tile` se retrouvent également dans `t >>= f`.

4.2. Respect des lois

Les monades se doivent de respecter trois propriétés :

```
return a >>= k          =  k a
t >>= return            =  t
t >>= (\x -> k x >>= h)  =  (t >>= k) >>= h
```

Montrons que la première loi est respectée.

Démonstration. Déroulons le calcul :

```
return a >>= k
foldT Prod k Unit Mod (return a) -- def >>=
(case x of
  ...
  Value a -> k a
  ...            ) (Value a)         -- def foldT + def return
k a                                  -- reduction
```

□

Montrons que la seconde loi est respectée.

Démonstration. Déroulons le calcul :

```
t >>= return
foldT Prod return Unit Mod t -- def >>=
foldT Prod Value  Unit Mod t -- def return
id t
t
```

□

La dernière loi est respectée car la substitution syntaxique est associative.

Démonstration. Par récurrence sur `t`, avec l'hypothèse :

$$\texttt{t >>= (x-> k x >>= h) = (t >>= k) >>= h.}$$

Cas de base Comme le cas `t = Unit` est trivial, considérons seulement le cas `t = Value a` :

```
t >>= (\x -> k x >>= h)
(Value a) >>= (\x -> k x >>= h)  -- def t
(return a) >>= (\x -> k x >>= h) -- def return
(\x -> k x >>= h) a              -- loi monade 1
k a >>= h                        -- reduction
((return a) >>= k) >>= h         -- loi monade 1
(t >>= k) >>= h                  -- def t
```

Cas inductif Supposons `t = Prod ta tb` (le cas `Mod` est très similaire) :

```
t >>= (\x -> k x >>= h)
Prod ta tb >>= (\x -> k x >>= h)                          -- def t
foldT Prod (\x -> k x >>= h) Unit Mod (Prod ta tb)  -- def >>=
Prod (ta >>= (\x -> k x >>= h) (item avec tb)             -- def foldT + réduction
Prod ((ta >>= k) >>= h) (idem avec tb)                    -- hypothèse récurrence
foldT Prod h Unit Mod (Prod (ta >>= k) (tb >>= k))  -- def foldT
(Prod ta tb >>= k) >>= h                                  -- def >>=
(t >>= k) >>= h                                           -- def t
```

□

Remarque. Cette implémentation n'utilise que des propriétés syntaxiques et est complètement agnostique vis-à-vis de la sémantique.

4.3. Exemple

Pour illustrer le fonctionnement de cette monade, nous reprenons l'exemple de la figure 3c avec la synchronisation aux 2/3. Nous pouvons la coder ainsi (avec `val = value`) :

```
t :: STile Rational Char
t = a % b % c
    where a = val 'A'
          b = resync   (-1/3) (val 'B')
          c = coresync (2/3)  (val 'C')

f 'A' = val "a1" % val "a2" % val "a3"
f 'B' = val "b1" % val "b2" % val "b3"
f 'C' = val "c1" % val "c2" % val "c3" % val "c4" % val "c5" % val "c6"

t_raff :: STile Rational String
t_raff = t >>= f
```

Dans `t_raff` les synchronisations décrites dans `t` sont préservées. En effet, la définition de `t_raff` est strictement équivalente à :

```
t_raff = a % b % c
  where a = f 'A'
        b = resync   (-1/3) (f 'B')
        c = coresync (2/3)  (f 'C')
```

5. Tuiles applicatives

Les foncteurs applicatifs permettent une notion d'application de fonction. Ils sont définis par deux fonctions : `pure` qui encapsule une valeur et `<*>` qui correspond à l'application de fonctions. Ces fonctions doivent vérifier les propriétés de neutralité et de compositionnalité.

```
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

La bibliothèque standard *Haskell* exige que l'instance d'`Applicative` soit sémantiquement une instance précise induite par la monade (si c'est aussi une instance de monade), c'est-à-dire que :

```
pure = return
(<*>) :: STile d (a -> b) -> STile d a -> STile d b
(<*>) fab  fa = fab >>= (\f ->
                fa  >>= (\a ->
                return (f a)))
```

Avec cette implémentation, si l'on a `tab :: STile (a -> b)` et `ta :: STile a`, alors la tuile `tab <*> ta` aura la structure de la tuile `tab` et ses feuilles `Value f` auront été substituées par `(fmap f ta)`. Par exemple `(val id % val id) <*> t` est égal à `t % t`. Plus concrètement, si `m` est un type représentant une note de musique et est doté d'une fonction `volume` permettant de modifier le volume d'une note, nous pouvons implémenter une fonction `echo3` :

```
echo3 t = let e = val (volume 1) % re (val (volume 0.6)
                                       % val (volume 0.2))
          in e <*> t
```

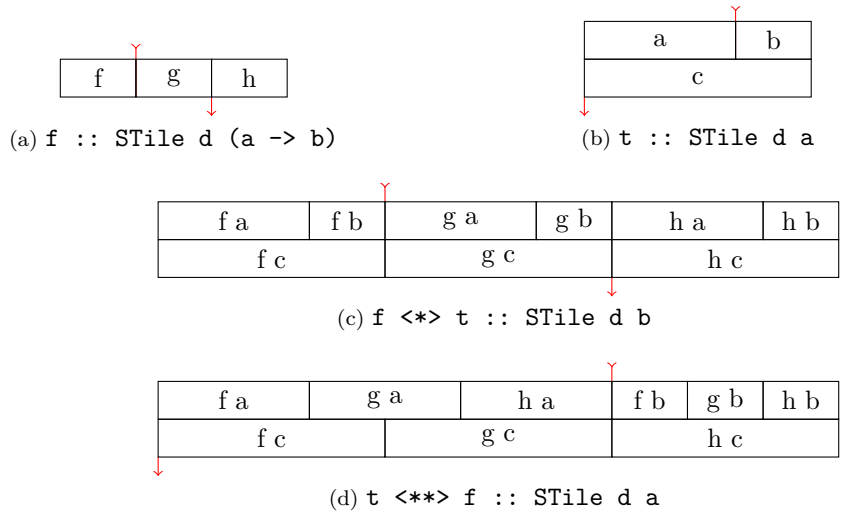
Une autre instance préservant la structure du paramètre aurait été possible, nous pouvons l'obtenir en échangeant les positions de la tuile de type `a -> b` et celle de la tuile de type `a`.

```
(<**>) ta tab = pure (\a f -> f a) <*> ta <*> tab
```

Cet opérateur préserve toujours la structure de la première opérande, mais cette fois il s'agit de la tuile portant les paramètres à appliquer à la fonction. Nous pouvons nous en servir pour modifier une tuile de sorte que chaque valeur soit introduite par une valeur, calculée par une fonction `intro_val` :

```
introduction_val t = t <**> co (val intro_val) % val id
```

Une illustration de la sémantique de ces deux opérateurs est donnée par la figure 11. Nous y considérons deux tuiles `f` et `t`, la première portant des valeurs de type `a -> b`, (voir figure 11a), et la seconde portant des valeurs de type `a` (figure 11b). L'application de la première tuile à la seconde `f <*> t` est illustrée par la figure 11c, la macrostructure de cette tuile est celle de `f` tandis que la microstructure est celle de `t`. Pour la tuile `t <**> f` illustrée à la figure 11d, la macrostructure vient de la tuile `t`.



(a) `f :: STile d (a -> b)`

(b) `t :: STile d a`

(c) `f <*> t :: STile d b`

(d) `t <**> f :: STile d a`

FIGURE 11 – Différence entre `<*>` et `<**>`

6. Rendu

Pour calculer le rendu d'un média tuilé, plusieurs questions se posent. En particulier, comment traiter le fait que les valeurs sont noyées dans une structure complexe ? Et comment traiter le fait que le phénomène temporel n'est pas parallèle, comme la vidéo ou la musique ?

6.1. Représentation *Piano roll*

Une manière naturelle de représenter la sémantique d'une tuile est de la voir comme un rouleau de piano pneumatique. Il s'agit d'une longue bande de papier perforée qui encode l'état du clavier (quelles touches et quelles pédales sont activées) instant par instant pour jouer un morceau de musique. C'est peu ou prou ce que ce modèle de tuile représente : l'évolution d'un ensemble de valeurs au cours du temps, comme l'illustre la figure 12.

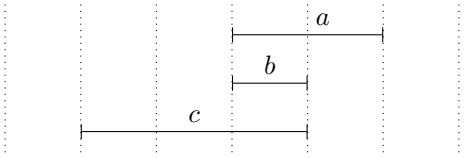


FIGURE 12 – La représentation piano roll d'une tuile

Sur cette représentation, la description exacte des synchronisations est perdue : nous ne pouvons pas dire si a et b commencent en même temps ou bien si b termine à la moitié de a, ou une unité de temps avant la fin, ou encore après le début. Il ne reste que les synchronisations par rapport à un référentiel externe : le temps.

6.2. Normalisation

Avec cette définition des tuiles, la position d'une valeur dans l'arbre syntaxique n'est pas corrélée avec sa position dans le temps : on s'autorise à séparer les valeurs par des durées positives ou négatives. De la même manière, la première valeur du *coreset* d'une tuile peut être placée à une durée négative de son point d'entrée.

Pour ces raisons, pour calculer un rendu, il faut pouvoir réordonner les valeurs d'après leur date de début. Ces procédures de rendu sont décrites dans [11, 2, 1].

Nous pouvons encoder ces piano roll dans notre modèle par une classe particulière de tuile qui vérifie plusieurs propriétés :

— tous les délais sont positifs sauf éventuellement deux, un avant les valeur, un après,
— toutes les valeurs a_i de durée d_i commençant simultanément à l'instant t sont codées par une tuile T_t de la forme $\%_i$ stretch $d_i\,0\,(-1)\,(\text{val}\ a_i)$,
— il existe un et un seul délai strictement positif entre chaque T_t et T_{t+1}.

Nous pouvons calculer une telle tuile à l'aide d'une fonction qui va extraire la liste des premières valeurs (et leur durée) à être actives :

```
extractFirstsValue Unit = (0, [], Unit)
extractFirstsValue (Value x) = (0, [(x,1)], Unit)
extractFirstsValue (Prod tile1 tile2) =
  let (d1,l1,t1) = extractFirstsValue tile1
      (d2,l2,t2) = extractFirstsValue tile2
  in case compareFirstD tile1 tile2 of
    LT -> (d1,l1, t1 % tile2)
    GT -> (d2 + dur tile1, l2, tile1 % t2)
    EQ -> (d1, l1 ++ l2, t1 % t2)
extractFirstsValue (Mod f pre post tile) =
  let (d,l,t) = extractFirstsValue tile
  in (f * (d - pre), map (\(x,y) -> (x,y*f)) l, Mod f pre post t)
```

Cette fonction suit un chemin dans l'arbre jusqu'aux feuilles contenant les premières valeurs à être jouées, et les remplace par des silences sans affecter la structure. La fonction `compareFirstD` compare les dates des premières valeurs de deux tuiles et gère l'éventualité que ces dates n'existent pas.

```
ht :: (Ord d, Num d) => STile d a -> (STile d a, STile d a)
ht tile =
  let (d,l,t) = extractFirstsValue tile
      head = delay d % foldl (\a (x,d) -> a % re (stretch d (val x))) (delay 0) l
      tail = delay (-d) % t
  in (head,tail)

norm tile = case firstD tile of
  Nothing -> delay (dur tile)
  Just _ -> let (h,t) = ht tile
            in h % norm t
```

La fonction `ht` coupe la tuile en deux parties. La première contient les premières valeurs actives et la durée jusqu'au début de ces valeurs, et la seconde le reste de la tuile. La « tête » est constituée du délai pour les premières valeurs, et de l'accumulation des valeurs commençant à cet instant. L'expression `re (stretch d (val x))` produit la tuile valant `x` sur une durée `d`, et dont les marqueurs de synchronisation se situent tous les deux au début de `x`. Nous construisons ainsi une tuile dont le rendu est équivalent et dont la forme est décrite plus haut.

La fonction `norm` effectue récursivement la conversion vers la classe de tuile qui nous intéresse.

Dans le cas général, le bind n'est pas stable par cette procédure de normalisation : $t \gg\!= f \neq (\text{norm}\ t) \gg\!= f$. La raison est que la normalisation transforme des synchronisations explicites en synchronisations induites, qui ne sont préservées par le bind que si la fonction de raffinement retourne une tuile de même durée pour toute valeur.

7. Conclusion

Les monades permettent de contraindre les opérations que l'on peut effectuer sur un type de données. Ici, nous nous en servons pour préserver des structures de synchronisations complexes entre des valeurs lors du raffinement de ces valeurs.

Dans cet article, nous considérons des phénomènes sur un domaine à une dimension. Dans l'avenir, nous souhaitons étudier dans quelle mesure une extension de notre modèle aux phénomènes en deux dimensions ou plus permet de conserver ces propriétés. Nous souhaitons aussi chercher comment définir la notion d'application de fonction tuilée, une fonction qui évolue au cours du temps, avec une synchronisation entre la tuile fonctionnelle et la tuile argument.

Nous travaillons actuellement à la mise en œuvre de ce raffinement dans un système musical réactif temps réel, où les durées et les valeurs ne sont connues qu'à l'exécution.

Références

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

[2] T. Bazin and D. Janin. Flux média tuilés polymorphes : une sémantique opérationnelle en Haskell. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2015.

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

[4] 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.

[5] P. Hudak. *The Haskell School of Expression : Learning Functional Programming through Multimedia*. Cambridge University Press, New-York, 2000.

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

[7] P. Hudak. *The Haskell School of Music : From signals to Symphonies*. 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)*. ACM Press, 2007.

[9] 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.

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

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

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

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

[14] 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.

[15] D. Janin and M. Desainte-Catherine. Des signaux aux symphonies : pour une modélisation homogène des objets sonores. In *Actes des Journées d'informatique Musicale (JIM)*, 2015.

[16] P. Wadler. Monads for functional programming, 1995.