Programmer avec des tuiles musicales: le T-calcul en Euterpea

Paul Hudak, David Janin

PROGRAMMER AVEC DES TUILES MUSICALES: LE T-CALCUL EN EUTERPEA

Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520-8285
paul.hudak@yale.edu

David Janin
LaBRI, CNRS UMR 5800,
IPB, Université de Bordeaux,
F-33405 Talence
david.janin@labri.fr

RÉSUMÉ

Euterpea est un langage de programmation dédié à la création et à la manipulation de contenus media temporisés – son, musique, animations, vidéo, etc.... Il est enchassé dans un langage de programmation fonctionnelle avec typage polymorphe : Haskell. Il hérite ainsi de toute la souplesse et la robustesse d'un langage de programmation moderne.

Le T-calcul est une proposition abstraite de modélisation temporelle qui, à travers une seule opération de composition : le produit tuilé, permet tout à la fois la composition séquentielle et la composition parallèle de contenus temporisés.

En présentant ici une intégration du T-calcul dans le language Euterpea, nous réalisons un outil qui devrait permettre d'évaluer la puissance métaphorique du tuilage temporel combinée avec la puissance programmatique du langage Euterpea.

1. INTRODUCTION

Programmer des pièces musicales

Il existe aujourd'hui de nombreux langages de programmation dédiés à la composition musicale tels que, par exemple, CLISP/CLOS [4] pour OpenMusic [1], Elody [25], ou Euterpea/Haskell [9].

Ces langages permettent de décrire des séquences de notes, mais offrent aussi, et de plus en plus, des opérations abstraites de manipulation de séquences musicales entières, en intégrant des paradigmes de programmation classiques tels que ceux de la programmation fonctionnelle paresseuse [10] et de la programmation synchrone [5]. La musique écrite dans ces langages peut ainsi résulter de l'*exécution* d'algorithmes de génération de flux musicaux.

Une première évidence, ces langages s'éloignent des notations musicales traditionnelles telles qu'elles ont été développées pour la musique occidentale. Ces notations avaient en effet vocation à être facilement lisibles par les interprètes. Au contraire, les programmes musicaux écrits dans ces langages de composition sont destinés à être interprétés par des ordinateurs. Ainsi, les flux musicaux produits peuvent toujours être traduits en notations musicales classiques, mais seulement lorsque c'est nécessaire et possible.

Dans ces langages, bon nombre de structures de données et de contrôle usuelles des langages de programmation sont ainsi disponibles pour cette composition algorithmique. Les listes, les arbres, et même les fonctions, permettent de prendre de la hauteur en offrant aux compositeurs des métaphores de modélisation musicale plus adaptées à une pensée musicale abstraite.

Outils pérennes vs innovations artistiques

Qu'on ne s'y trompe pas, même s'il peut être amusant de se demander quel usage artistique les compositeurs d'antan auraient pu faire de ces nouveaux langages de programmation musicale, la première raison d'être de ces outils est d'offrir un gain de productivité. En utilisant ces langages, le compositeur doit pouvoir déléguer à l'outil informatique la réalisation de tâches répétitives et fastidieuses. Depuis sa naissance, en musique comme en comptabilité, c'est là, la première fonction de l'outil informatique.

Bien entendu, ces langages peuvent ouvrir de nouvelles pistes de composition musicale. C'est une évidence dans le traitement et la manipulation du son. C'est aussi de plus en plus fréquent pour l'écriture musicale elle-même, en musique dite populaire comme en musique dite savante. Cependant, même dans ces cas, il faut garder à l'esprit cette caractéristique qui a fait le succès des outils informatiques : le gain de productivité que ces langages peuvent offrir.

Les critères d'évaluation de ces nouveaux langages ne sauraient donc se limiter à leur valeur ajoutée sur le plan artistique, mais bien, tout autant, sur leur capacité à intégrer les bons usages de la programmation moderne : robustesse, réutilisabilité, souplesse, fiabilité, modularité, etc.

L'enjeu est d'importance puisque de nombreuses problématiques rencontrées dans la programmation de flux musicaux sont transversales à la plupart des outils de gestion et de manipulation de media temporisés [8] tels que les flux audios et musicaux bien sûr, mais aussi les flux vidéo, les animations, aussi bien que les flux de contrôle-

travail partiellement soutenu par ANR-12-CORD-0009

commande.

Autrement dit, si l'on peut souhaiter participer à l'engouement pour ces technologies du numérique par l'apport de techniques et d'applications innovantes dont les limites ne semblent être que celle de notre imagination, une activité de recherche de fond, plus lente, mathématiquement bien fondée, n'en demeure pas moins indispensable pour garantir, à termes, la pérennité des métaphores et concepts mis en œuvre.

Des outils pour structurer l'espace et le temps

Paradoxalement, les structures musicales induites par ces langages de programmation peuvent aussi souffrir de limitations.

Par exemple, la modélisation de séquences musicales à l'aide de grammaires génératives *hors contexte*, un outil conceptuel d'une importance considérable pour la description des langages de programmation eux-mêmes, peut sembler prometteuse [24]. Mais, c'est là une évidence, de nombreuses constructions musicales ne s'y prêtent pas vraiment. Des pièces polyrythmiques à la structure complexe telles que, par exemple, les arabesques de Debussy, peuvent mettre à mal ce type d'approche.

Les caractéristiques spatiale et temporelle du langage musical induisent en effet une problématique majeure à laquelle ces langages de programmation doivent répondre. Le compositeur/programmeur doit disposer d'opérations de composition de structures musicales permettant, selon les cas, de positionner ces structures l'une *après* l'autre (composition séquentielle) ou bien l'une *en même temps* que l'autre (composition parallèle).

La théorie des langages classiques qui fait la force des grammaires génératives pourrait se montrer insuffisante. Elles reposent largement sur le *monoïde libre*, c'est-à-dire, pour un alphabet de symboles donné, l'*ensemble des mots finis* sur cet alphabet, équipé de l'*opération binaire de concaténation* consistant à positionner deux mots l'un après l'autre. Son extension à des structures parallèles n'est pas sans poser de réelles difficultés théoriques. La recherche, la mise au point, et le développement d'extensions à ce modèle intrinsèquement séquentiel qui intègreraient des opérateurs de composition parallèle a déjà fait l'objet de nombreux travaux ces dernières décennies.

Les structures arborescentes, les diagrammes d'évènements, les monoïdes de traces, pour n'en citer que quelques-uns, sont autant de propositions d'extensions du monoïde libre pour l'intégration et la modélisation de ces phénomènes comportementaux plus complexes qui nécessitent une structuration de la l'espace *et* du temps.

Cette problématique de recherche fondamentale, transversale à de nombreux domaines de la recherche en informatique et, au-delà, des sciences du numériques, est loin d'être close.

La modélisation par tuilage spatio-temporel

Dans la continuité des propositions existantes de langages pour la programmation musicale [1, 25, 9], et, tout autant, dans la continuité des approches structuralistes de nature plus théorique [24], la *modélisation par tuilage* que nous nous proposons de développer ici, est une tentative de résoudre cette délicate combinaison d'opérateurs séquentiels et parallèles. Pour cela, nous ne proposons plus qu'un seul opérateur, la *composition tuilée*.

Plus précisément, en nous appuyant sur le concept de *flux media temporisés* [8] enrichi par des *profils de synchronisation* [3], nous obtenons une notion de *flux media temporels tuilés* dont le produit, en autorisant des superpositions partielles, apparaît tout à la fois comme un opérateur séquentiel et un opérateur parallèle.

En pratique, la composition tuilée s'inspire de travaux déjà anciens. Le produit de tuilage ne fait qu'offrir une alternative à la notion de barre de mesure en musique. Il permet par exemple de modéliser la notion musicale d'anacrouse [12]. Le produit tuilé apparaît déjà dans le langage LOCO [6], une adaptation du langage *Logo* à la manipulation de flux musicaux. Ces auteurs proposent en effet des opérateurs *pre* et *post* qui permettent de décrire explicitement la façon dont deux séquences musicales pourront être combinées en séquence avec des superpositions partielles.

La formalisation du produit tuilé, redécouvert dans une étude d'outils pour la modélisation rythmique [12], nous a conduits à proposer une algèbre dédiée à la synchronisation des flux musicaux [3]. Deux outils de manipulations de flux audio tuilés, la librairie *libTuiles* et l'interface *liveTuiles* [19], dérivent de cette approche. In fine, une proposition abstraite : le T-Calcul [18], propose d'intégrer ces concepts à un langage de programmation.

En théorie, le produit de tuilage apparaît aussi dans les monoïdes inversifs [23]. Plus précisément, les éléments du monoïde inversif libre, décrit de façon effective par Scheiblich[28] et Munn[26] dans les années 70, sont des arbres à deux racines dont le produit est, précisément, un produit tuilé. La théorie des monoïdes inversifs est donc naturellement adaptée à la description de structures avec superpositions partielles comme l'illustre le théorème de représentation de Stephen [29] qui généralise la présentation de Scheiblich et Munn à tout monoïde inversif. Plus généralement, la théorie des monoïdes inversifs induit une notion de structure dans un espace de dimension arbitraire [20, 21, 22], dont le produit est défini sans ambiguïté, et dont l'usage comme outil de modélisation pour les systèmes informatisés semble prometteur [17].

Une théorie des langages adaptée à la description et à la manipulation de sous-ensembles de monoïdes inversifs est actuellement en cours de développement. Elle offre des outils traditionnels de manipulation des langages, qu'ils soient logiques [14], algébriques [11, 13] ou qu'ils s'appuient sur la théorie des automates [16, 15, 13].

Le produit tuilé, tel que nous proposons de le manipuler en vue d'applications musicales, est donc une construction particulièrement bien fondée au cœur d'une théorie

mathématique dont la robustesse n'est plus à démontrer, et qui apparaît tout naturellement, comme nous allons le voir ici, dès lors que l'on cherche à abstraire un tant soit peu les concepts sous-jacents dans la notion de mixage.

2. DES FLUX MEDIA AUX FLUX MEDIA TUILÉS

Un *flux média temporisé*, de type `Temporal a`, est une structure abstraite représentant des données qui sont positionnées dans le temps relativement au début du flux media. Éventuellement produites par un programme, ces structures peuvent être, en théorie, de durée infinie. En effet, la durée d'un flux média temporisé peut dépendre d'événements externes dont la date d'arrivée est inconnue.

Figure 1. Un flux média temporisé fini m_1 et un flux média temporisé infini m_2.

Les opérations naturellement associées à ces flux média temporisés sont : la *composition séquentielle*

$$m_1 :+: m_2$$

qui modélise le lancement de deux flux média temporisés l'un après l'autre, le premier étant nécessairement fini, et, la *composition parallèle*

$$m_1 :=: m_2$$

qui modélise le lancement de deux flux média temporisés en même temps (voir Figure 2).

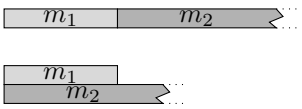

Figure 2. Composition séquentielle $m_1 :+: m_2$ et composition parallèle $m_1 :=: m_2$ des flux m_1 et m_2.

Il apparait que ces deux opérations sont des cas particuliers d'une opération plus générale : le *produit synchronisé* ou *produit tuilés*.

Formellement, on définit les flux media temporisés tuilés, de type `Tile a`, comme suit. Un flux media temporisé m est enrichi de deux marqueurs de synchronisation in et out.

Le marqueur d'entrée in est défini par la distance en temps

$$s \in [0, dur(m)]$$

qui le sépare du début du flux. Le marqueur de sortie out est, lui, défini par la distance en temps

$$d \in [-s, dur(m) - s]$$

qui le sépare du marqueur in.

Lorsque d est positif ou nul, comme illustré Figure 3, on dit que le flux tuilé est *positif* :

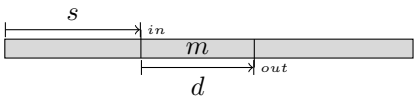

Figure 3. Un flux media temporel tuilé positif

Dans ce cas, le marqueur d'entrée in se trouve *avant* le marqueur de sortie out.

Lorsque d est négatif ou nul, comme illustré Figure 4, on dit que le flux tuilé est *négatif* :

Dans ce cas, le marqueur d'entrée in se trouve *après* le marqueur de sortie out.

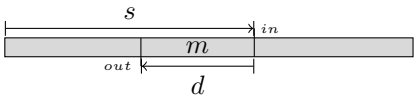

Figure 4. Un flux media temporel tuilé négatif

Remarque. Cette définition des flux tuilés reste valide avec des flux infinis, la position de référence pour positionner in et out sur un flux étant toujours le début de ce flux.

Le *produit synchronisé* $t_1 \,\%\, t_2$ de deux flux tuilés

$$t_1 = \langle s_1, d_1, m_1 \rangle \quad \text{et} \quad t_2 = \langle s_2, d_2, m_2 \rangle$$

se définit alors en deux étapes.

Synchronisation : cette première étape consiste à positionner les deux flux t_1 et t_2 l'un par rapport à l'autre, de telle sorte que le marqueur de sortie out_1 du premier flux tuilé t_1 coïncide avec le le marqueur d'entrée in_2 du second flux tuilé t_2.

Fusion : cette seconde étape consiste alors à fusionner [1] les deux flux sous-jacents ainsi positionnés, en conservant in_1 comme marqueur d'entrée et out_2 comme marqueur de sortie résultant de ce produit.

Cette construction est illustrée Figure 5. Dans cette figure, qui met en œuvre deux tuiles positives, on a $s = s_1$. C'est là un cas particulier. En toute généralité, comme l'illustre une autre instance de produit Figure 6, on aura

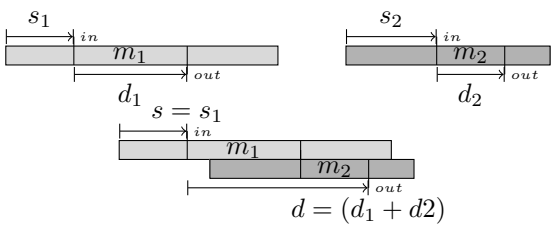

Figure 5. Une instance de produit synchronisé

$$s = \max(s_1, s_2 - d_1) \quad \text{et} \quad d = d_1 + d_2$$

1 . Fusion qui dépendra du media considéré : mixage pour de l'audio, union pour de la musique, superposition pour de la vidéo, etc.

On peut vérifier que ces formules restent les mêmes lorsque t_1 ou t_2 sont l'une, l'autre ou toutes les deux, des tuiles négatives.

L'implémentation en *Haskell/Euterpea* des flux media tuilés et du produit de synchronisation résultant ne fait que mettre en œuvre ces schémas. Les opérateurs qui peuvent dériver de ces structures sont décrits dans la suite, à la fois par leur implémentation en *Haskell* et par le schéma correspondant.

3. IMPLÉMENTATION EN HASKELL

Le type de donnée `Tile a` est codé de la façon suivante à partir du type `Music a`, qui est une instance particulière du type `Temporal a` :

```
> module Tiles where
> import Euterpea

> data Tile a = Tile {
>         syncT :: Dur,
>         durT  :: Dur,
>         musT  :: Music a }
>              deriving (Show,Eq,Ord)
```

Le module importé, *Euterpea*, contient en particulier l'implémentation du type `Music a`. Dans ce module, le type `Dur` est définit comme rationnel. Le lecteur intéressé pourra consulter le manuel [9] pour une description exhaustive et didactique des nombreuses fonctionnalités proposées par ce module : du *signal* à la *symphonie*.

Le produit synchronisé, noté $\%$, est alors défini par :

```
> (%) :: Tile a → Tile a → Tile a
> Tile s1 d1 m1 % Tile s2 d2 m2 =
>     let d = s1+d1-s2
>     in Tile (max s1 (s2-d1)) (d1+d2)
>             (if d>0 then (m1 :=: delayM d m2)
>                     else (delayM (-d) m1 :=: m2))
```

Dans ce codage, on retrouve les calculs décrits page précédente pour produire les valeurs et durées de synchronisation résultantes.

Le calcul du flux musical résultant est effectué par un produit parallèle :=: des deux composantes musicales `m1` et `m2` associées aux arguments. La synchronisation de ces deux composantes est mise en œuvre par le retard (fonction `delayM`) appliqué au flux musical qui vient en second. Il s'agira de `m2` lorsque `s1 > s2 - d1` (voir Figure 5). Il s'agira de `m1` lorsque `s1 < s2 - d1` (voir Figure 6).

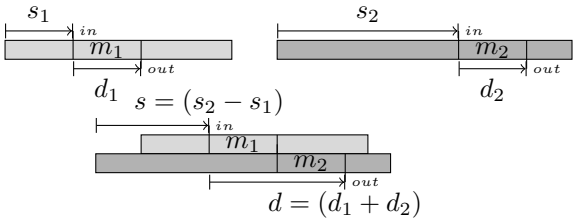

Figure 6. Une autre instance de produit synchronisé

Trois fonctions sur les tuiles : le *reset*, le *co-reset* et l'*inverse*, découlent de ces définitions. Elles sont, comme on le vera, liées les unes aux autres.

```
> re,co,inv :: Tile a → Tile a
> re (Tile s d m) = Tile s 0 m
> co  (Tile s d m) = Tile (s+d) 0 m
> inv (Tile s d m) = Tile (s+d) (-d) m
```

Leur effet sur une tuile t est décrit Figure 7.

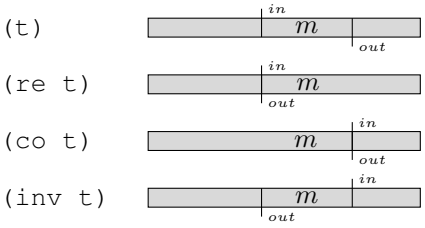

Figure 7. Reset, co-reset and inverse

4. PREMIÈRE EXPÉRIMENTATION

Chaque note, de type `Music Pitch` en Euterpea, est décrite comme un triplet $(\mathtt{n}, \mathtt{o}, \mathtt{d})$ avec un nom [2] `n`, une octave [3] `o` et une durée [4] d. Ces notes peuvent être converties en notes tuilées à l'aide de la fonction `t` définie par :

```
> t :: (Octave → Dur → Music Pitch)
>                  → Octave → Dur → Tile Pitch
> t n o d = Tile 0 d (n o d)
```

De même pour les silences, avec la fonction `r` définie par :

```
> r :: Dur → Tile a
> r d = Tile 0 d (rest d)
```

On peut ainsi proposer un premier exemple :

```
> fj1 = t c 4 en % t d 4 en % t e 4 en % t c 4 en
> fj2 = t e 4 en % t f 4 en % t g 4 qn
> fj3 = t g 4 sn % t a 4 sn % t g 4 sn % t f 4 sn
>                           % t e 4 en % t c 4 en
> fj4 = t c 4 en % t g 3 en % t c 4 qn
> fj  = fj1 % fj1 %
        re (fj2 % fj2 % fj3 % fj3 % fj4 % fj4)
> fjr = fj % fj % fj % fj

> test1 = playT (fjr % r 4)
```

qui jouera *Frère Jacques* en canon.

Une fonction `tempoT` permet de changer le tempo des tuiles. Elle est codée de la façon suivante, héritant en quelque sorte de la fonction `tempo` sur les objets musicaux sous-jacents :

2 . En notation anglaise, de `a` pour *la* à `g` pour *sol*, avec `c` pour *do*, `cf` (*c flat*) pour *do* bémol, `cs` (*c flat*) pour *do* dièse, etc.

3 . C'est-à-dire un numéro d'octave, le do à la clé correspondant à `c 4`, étant précédé de `b 3` et suivie de `d 4`, un clavier de piano allant typiquement de `a 0`, le *la* le plus grave, à `c 8`, le *do* le plus aigu.

4 . En valeur symbolique valant fraction de ronde, avec des constantes prédéfinies, en notation anglaise, telle que `wn` (*whole note*) pour la ronde, valant 1, ou bien `qn` (*quarter note*) pour la noire, valant $1/4$, ou encore `en` (*eight note*) pour la croche, calant $1/8$, etc.

```
> tempoT :: Dur → Tile a → Tile a
> tempoT r (Tile s d m)
>                  = Tile (s/r) (d/r) (tempo r m)
```

Plus généralement, toute fonction agissant sur les objets musicaux peut être appliquée aux tuiles, lorsqu'on ne souhaite pas modifier les paramètres de synchronisation, grâce à la fonction d'ordre supérieur `liftT` définie par :

```
> liftT :: (Music a → Music b) → (Tile a → Tile b)
> liftT f (Tile s d m) = Tile s d (f m)
```

L'exemple suivant nous permettra de jouer *frère Jacques* sur un piano Rhodes deux fois plus vite.

```
> fjR = litfT (intrument rhodesPiano) fjr
> test2 = playT (tempoT 2 (fjR % r 4))
```

5. ÉQUIVALENCE OBSERVATIONNELLE

Un concept important en Euterpea est l'équivalence observationnelle. Deux flux musicaux `m1` et `m2` sont jugées équivalent, ce qu'on note

$$\texttt{m1} \Longrightarrow \texttt{m2}$$

lorsqu'ils produisent le même effet à l'execution, c'est à dire lorsque `play m1` et `play m2` produisent le même effet, où `play` désigne la fonction *jouant* les flux musicaux.

Cette *équivalence observationnelle* se généralise aux flux tuilés en disant que deux flux tuilés `t1` et `t2` sont observationnellement équivalents lorsqu'ils produisent le même effet à l'exécution quel que soit le contexte d'exécution dans lequel on les joue. Cette mise en contexte nous permet de rendre compte des paramètres de synchronisation.

Formellement, en notant `playT` la fonction qui permet de *jouer* le flux media d'une tuile *entre* les points de synchronisation in et out, deux flux tuilés `t1` et `t2` seront équivalents, ce qu'on notera toujours

$$\texttt{t1} \Longrightarrow \texttt{t2}$$

lorsque, pour toutes tuiles supplémentaires `t3` et `t4`, les flux tuilés

`(t3 % t1 % t4)` et `(t3 % t2 % t4)`

produisent le même effet lorsque joués par la fonction `playT`.

Autrement dit, on définit la fonction `tToM` qui extrait le flux musical d'une tuile entre les points de synchronisation in et out par :

```
> tToM :: Tile a → Music a
> tToM (Tile s d m) =
>             removeZeros (takeM d  (dropM s  m))
```

la fonction `(dropM s m)` supprimant le début d'une durée `s` du flux musical `m`, la fonction `(takeM d m)` ne gardant que le début d'une durée `d` du flux musical `m`, ou rien lorsque `d` est négatif, et la fonction `(removeZeros m)` supprimant les silences du flux musical `m` en conservant la position des objets musicaux non silencieux.

On peut alors définir la fonction `playT` par :

```
> playT = play ∘ tToM
```

où le point désigne la composition fonctionnelle.

On vérifie que deux flux tuilés `t1` et `t2` sont équivalents si et seulement si, pour toutes durées `d1` et `d2` on a bien :

```
tToM ((r d1) % t1 % (r d2))
                ═══
        tToM ((r d1) % t2 % (r d2))
```

où `(r d)` désigne la tuile vide , ou silence, de durée `d` (voir ci-dessous).

En supposant que pour tout flux temporel `m` on a bien :

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

ce qui se justifie dès lors qu'on fait abstraction de l'incrément en volume sonore pouvant résulter de la superposition du flux `m` à lui-même, on a les équivalences suivantes sur les flux tuilés :

$$\begin{aligned}
\texttt{m1 \% (m2 \% m3)} &\Longrightarrow \texttt{(m1 \% m2) \% m3} \\
\texttt{m} &\Longrightarrow \texttt{(m \% (inv m) \% m)} \\
\texttt{(re m)} &\Longrightarrow \texttt{(m \% (inv m))} \\
\texttt{(co m)} &\Longrightarrow \texttt{((inv m) \% m)}
\end{aligned}$$

La première équivalence indique que l'ordre dans lequel on évalue les composants d'un produit tuilé n'a pas d'importance. C'est l'équivalence d'associativité. La seconde équivalence indique que l'opération d'inversion est une involution. La troisième, que l'inverse d'un flux est bien un inverse au sens de la théorie des semigroupes (voir remarque ci-dessous). Les quatrième et cinquième équivalences montrent le reset et le co-reset sont en fait des opérations dérivées du produit tuilé et de l'inversion.

Remarque. On peut poursuivre plus en détail l'étude de cette équivalence est découvrir, comme en théorie des semigroupes inversifs, que les flux tuilés de durée de synchronisation nulle sont les seuls qui satisfont n'importe laquelle des équivalences suivantes :

$$\begin{aligned}
\texttt{m} &\Longrightarrow \texttt{(m \% m)} \\
\texttt{m} &\Longrightarrow \texttt{(inv m)}
\end{aligned}$$

et que, de plus, ils commutent, c'est-à-dire que

$$\texttt{m \% n} \Longrightarrow \texttt{n \% m}$$

lorsque `m` et `n` ont une durée de synchronisation nulle.

On peut en déduire que, pour tout flux tuilé `m` il n'existe, à équivalence près, qu'un unique flux tuilé `n` tel que

$$\begin{aligned}
\texttt{m} &\Longrightarrow \texttt{(m \% n \% m)} \\
\texttt{n} &\Longrightarrow \texttt{(n \% m \% n)}
\end{aligned}$$

et qu'il s'agit de `(inv m)`.

Autrement dit, en constatant de plus que le silence de durée nulle, `(r 0)`, constitue un élément neutre pour le produit tuilé, les flux tuilés munis du produit de synchronisation forment bien, à équivalence observationnelle près, un monoïde inversif [23].

Dans ce langage de manipulation de tuiles, nous retrouvons aussi les produits séquentielles et parallèles de Euterpea.

En effet, dans le cas flux musicaux finis, on définit la fonction `mToT` qui transforme tout flux musical fini `m` en un flux tuilé par :

```
> mToT :: Music a → Tile a
> mToT m =
>      let d  = dur(m) in Tile 0 d m
```

On constate que ce codage est injectif vis à vis de l'équivalence observationnelle. On constate de surcroit que ce codage est fonctoriel vis à vis de la composition séquentielle. En effet, pour tout flux musical fini `m1` et `m2` on a bien :

```
mToT (m1 :+: m2) == (mToT m1) % (mToT m2)
```

Autrement dit, le produit de synchronisation code, via la foncteur `mToT` le produit séquentiel.

Dans le cas de flux (potentiellement) infinis, l'encodage décrit si-dessous échoue. En effet, le produit séquentielle de deux flux infinis ne peut être définit de façon satisfaisante puisque le second flux se voit en quelque sorte repoussé à l'infini. Il ne pourra être joué.

On retrouve ici une problématique abordée et résolue par la théorie des ω-semigroupes [27] en distinguant les structures finies (les *strings*) et les structures infinies (les *streams*). Pourtant, il apparait que la manipulation conjointe de ces deux types d'objets peut être faites dès lors qu'ils sont tuilés (voir[7] pour plus de détails).

En pratique, on définit la fonction `sToT` qui transforme tout flux musical (potentiellement) infini `m` en un flux tuilé par :

```
> sToT :: Music a → Tile a
> sToT m = Tile 0 0 m
```

On vérifie que ce codage est injectif sur les flux musicaux infinis. De plus, il est fonctoriel vis à vis de la composition parallèle. En effet, pour tout flux musical (potentiellement) infini `m1` et `m2` on a bien :

```
sToT (m1 :=: m2) == (sToT m1) % (sToT m2)
```

Autrement dit, le produit tuilé encode, via le foncteur `sToT`, le produit parallèle.

6. RESYNCHRONISATIONS

Un premier jeux de fonctions définit sur les flux tuilés permet de manipuler la position des points de synchronisation positionné sur un flux musical tuilé.

Il s'agit de la fonction `resync` qui permet de déplacer le point d'entrée in de la synchronisation, et de la fonction `coresync` qui permet de déplacer de façon duale le point de sortie out de la synchronisation.

La fonction `resync` est définit par :

```
> resync :: Dur → Tile a → Tile a
> resync o (Tile s d m) =  let ns = s+o in
>      if (ns < 0) then
>              (Tile 0 (d+s-ns) (delayM (-ns) m))
>      else  (Tile ns (d-o) m)
```

La fonction `coresync` est, quant à elle, définit par :

```
> coresync :: Dur → Tile a → Tile a
> coresync o (Tile s d m) = let ns = s+d+o in
>      if (ns < 0) then
>               (Tile (s+o) (-s-o) (delayM (-ns) m))
>      else   (Tile s (d+o) m)
```

Le comportement de ces fonctions est illustré Figure 8 pour un offset $o > 0$.

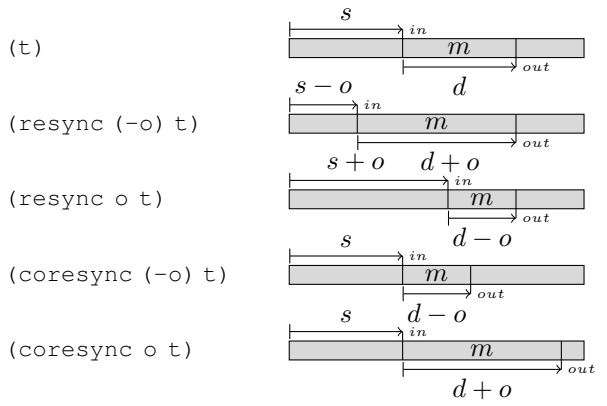

Figure 8. Resynchronisation et co-resynchronisation

Il apparait que ces fonctions induisent des actions du groupe des nombres rationnels muni de l'addition sur l'ensemble des flux tuilés. En pratique, pour tout flux musical tuilé `t` et tout offset rationnel `a` et `b` on a en effet :

```
          resync 0 t == t
resync a (resync b t) == resync (a+b) t

         coresync 0 t == t
coresync a (coresync b t) == coresync (a+b) t
```

De plus, les fonctions `resync` et `coresync` commutent en quelque sorte selon l'équivalence

```
resync a (coresync b t) == coresync b (resync a t)
```

Elles sont aussi duales l'un de l'autre au sens des équivalences suivantes :

```
  resync a (inv t) == (inv (coresync a t))
coresync a (inv t) == (inv (resync a t))
```

Enfin, leur comportement vis à vis du produit tuilé est décrit par les équivalences observationnelles suivantes. Pour tout flux musical tuilé `t1` et `t2` et tout offset de durée `a`, on a :

```
  resync a (t1 % t2) == (resync a t1) % t2
coresync a (t1 % t2) == t1 % (coresync t2)
```

La fonction `shift`, qui pourrait dériver des fonctions `sync` et `resync`, consiste à décaler les points d'entrée et de sortie de synchronisation d'une même valeur. Elle est codé par :

```
> shift :: Dur → Tile a → Tile a
> shift o (Tile s d m) = let ns = s+o in
>     if (ns < 0) then (Tile (0) (d) (delayM (-ns) m))
>                 else (Tile (s+o) d m)
```

Cette fonction est liée aux fonctions `sync` et `resync` de la façon suivante :

```
(shift a t) === (resync a (coresync a t))
```

Remarquons qu'elle est fonctorielle vis à vis du produit tuilé. On a en effet :

```
shift n (t1 % t2) === (shift n t1) % (shift n t2)
```

Pour illustrer la puissance de la métaphore du tuilage au sein d'un langage de programmation complet tel que Haskell, on propose ci-dessous le codage d'une fonction `crossing` qui simule, par répétition et décalage successifs, le *croisement* de deux flux musicaux tuilés.

```
> crossing :: Dur → Tile a → Tile a → Tile a
> crossing o t1 t2 =
>         if ((centerT(t1) > 0)
>                 && (centerT(t2) > 0)) then
>             let v1 = (resync o t1)
>                 v2 = (resync (-o) t2)
>             in (t1 % t2 % (crossing o v1 v2))
>         else (t1 % t2)
```

Cette fonction est alors mis en oeuvre dans l'exemple musicale suivant.

```
> train1 =  liftT (instrument RhodesPiano)
>   (r en % t a 3 en % t c 4 en % t e 4 en
>                       % t d 4 en % r (3*en))
> train2 =  liftT (instrument RhodesPiano)
>   (t e 4 en % t g 3 en % t a 3 en
>                        % t a 3 en % r (4*en))
> crossLine = (crossing (-en) train1 train2)
```

Une ligne de percussion régulière `percLine` est construite et lancée en parallèle avec la ligne `crossLine` pour permettre à l'auditeur de conserver une référence temporelle régulière qui contraste avec le *croisement* des lignes mélodiques `train1` et `train2`.

```
> bassDrum = (liftT (instrument Percussion)
>       (Tile 0 wn (perc AcousticBassDrum wn)))
> hiHatDrum = (liftT (instrument Percussion)
>       (Tile 0 qn (perc ClosedHiHat qn)))
> percLine = ( (re ((r en) % bassDrum))
> % (hiHatDrum % hiHatDrum % hiHatDrum % hiHatDrum))
> %\ (re percLine)
```

L'audition de l'ensemble se fait par le lancement de `testCrossing` codé par :

```
> testCrossing = playT ((re percLine) % crossLine)
```

A noter que la définition de `percLine` est récursive. Pour cela, fait appel à un nouvelle opérateur de composition séquentiel, `%\`, dont la définition et l'utilité sont décrites dans la section suivante.

7. CONTRACTIONS ET EXPANSIONS

Lors d'une resynchronisation, l'invariant et le flux musicale sous-jacent au flux tuilés. Un jeu de fonction sensiblement différent consiste à préserver la taille de la synchronisation tout en contractant ou étirant le flux musicale sous-jacent. Il s'agit de la fonction `strech` qui permet d'étirer et de contracter le flux musical en maintenant la position du point de sortie de synchronisation *out* sur le flux musical, et de la fonction `costretch` qui étire ou contracte de façon, duale, le flux musical en maintenant le point d'entrée de synchronisation *out* sur le flux musical.

La fonction `stretch` est définit par :

```
> stretch :: Dur → Tile a  → Tile a
> stretch r (Tile s d m) =
>     let valid = assert (r > 0) in
>      (Tile (s*r + d*(r-1)) d (tempo (1/r) m))
```

Dans cette définition, la fonction `assert` est importé du module Haskell `Control.Exception`.

La fonction `costretch` est quant à elle définit par :

```
> costretch :: Dur → Tile a → Tile a
> costretch r (Tile s d m) =
>     let valid = assert (r > 0) in
>      (Tile (s*r) d (tempo (1/r) m))
```

Le comportement de ces fonctions est décrit Figure 9 avec un facteur $r > 1$ et en notant (abusivement) $m * r$ le flux $tempo\ (1/r)\ m$, et m/r le flux $tempo\ r\ m$.

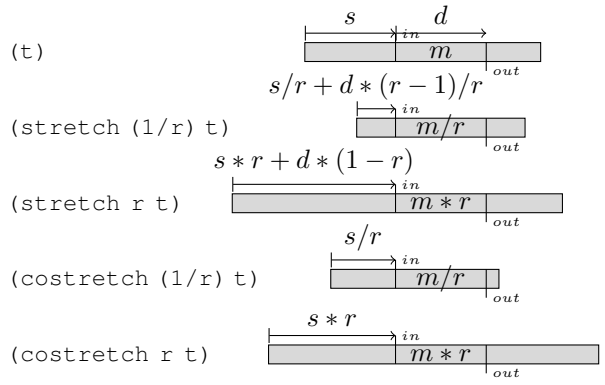

Figure 9. Stretch et co-stretch

De façon analogue aux fonctions de resynchronisation, ces fonctions induisent une action du groupe des nombres rationnels strictement positif munie de la multiplication sur l'ensemble des flux tuilés. Pour tout flux musical tuilé `t` et tout offset de durée `a` et `b` on a :

```
                stretch 1 t === t
stretch a (stretch b t) === stretch (a*b) t

              costretch 1 t === t
costretch a (costretch b t) === costretch (a*b) t
```

Ces deux fonctions sont duales l'un de l'autre au sens des équivalences suivantes :

```
   stretch a (inv t) ≡≡ (inv (costretch a t))
 costretch a (inv t) ≡≡ (inv (stretch a t))
```

Enfin, le comportement de ces fonctions vis à vis du produit synchronisé est décrit par les equations suivante. Pour tout flux musical tuilé `t1` et `t2` et tout facteur `a` > 0, on a :

```
 stretch a (t1 % t2) ≡≡
                (tempoT (1/a) t1) % (stretch a t2)

costretch a (t1 % t2) ≡≡
                (costretch a t1) % (tempoT (1/a) t2)
```

L'utilisation de `stretch` et `costretch` peut être illustré par les transformations rythmiques déjà évoquées dans [12]. En partant d'un rythme de marche :

```
> march = t c 4 qn % r qn % t g 4 qn % r qn
```

on peut, par une contraction, engendrer un rythme de valse :

```
> waltz = (stretch (2/3) march)
```

dont seul sont joués les deuxième et troisième temps lorsque l'interval entre in et out est interprété comme une mesure à trois temps.

On peut aussi engendrer le *tumbao* de la salsa brésilienne :

```
> tumbao = (stretch (5/4) march)
```

où le do est joué sur le 4ième temps de la mesure qui précède, et le sol sur la levée du 2ième temps, lorsque l'interval entre in et out est interprété comme une mesure à quatre temps.

Ainsi, avec une structure rythmique répétitive marquant le début de chaque mesure, c'est à dire le point in, définit par `firstBeat` on peut tester ces structures rythmiques de la façon suivante :

```
> firstBeat = bassDrum % r wn %\ (re firstBeat)
> testWaltz = re firstBeat
>       % waltz % waltz % waltz % waltz
> testSalsa = re firstBeat
>       % tumbao % tumbao % tumbao % tumbao
```

avec, là encore, une définition recursive de `firstBeat` à l'aide de l'opérateur `%\` décrit dans la section suivante.

8. TUILES RÉCURSIVES

Dans les exemples ci-dessus, nous avons vu comment les structures de contrôle classique des langages de programmation permettent de définir *algorithmiquement* des flux tuilés finis complexes.

Le langage Haskell, reposant sur un principe d'évaluation paresseuse, permet aussi de définir des objets potentiellement infinis qui sont évalués à la demande. Nous souhaiterions pouvoir utiliser cette caractéristique pour définir des tuiles potentiellement infinit.

Par exemple, étant donné une tuile finie t, on souhaiterait pouvoir définir une tuile `x` de support musical infini, via une equation de la forme

```
x = t % (re x)
```

Dans une telle equation, l'appel récursif se fait sur le `reset` de la variable `x` afin de produire une tuile dont la distance de synchronisation serait celle de la tuile `t`. Pourtant, malgré cette précaution, l'évaluation de cette équation en Haskell boucle !

En effet, le modèle des flux tuilés est ainsi fait que les superpositions en amont du point de synchronization in peuvent provenir de tuiles situées arbitrairement en aval dans une suite de produit synchronisé. Notre implémentation boucle donc sur une telle équation car elle doit dérouler toute les répétitions de `x` pour en connaitre les anticipations : le typage de la tuile `x` ainsi définit par équation échoue à être calculé par Haskell.

Dans [18], des conditions necessaires et suffisantes, calculables, sont décrites pour résoudre cette récursion. Néanmoins, elles sont proposés dans une définition de T-calcul pure, particulièrement limité. Il ne contient pas de structure de contrôle telle que les conditionnelles. Dans l'implémentation du T-calcul proposé ici, qui hérite de toute l'expressivité d'Haskell, le calcul du type d'un flux tuilé définit par equation est, en toute généralité, indécidable.

Pour remédier à cela, nous proposons un nouveau produit synchrone, partiel, qui résout ce calcul de type potentiellement cyclique en « coupant » les anticipations qui y conduise. Plus précisément, on propose le produit restreint `%\` définit par :

```
> (%\) :: Tile a → Tile a → Tile a
> Tile s1 d1 m1 %\ ~(Tile s2 d2 m2) =
>    let valid = assert (d2 == 0)
>     Tile s1 d1
>           (m1 :=: delayM (s1 + d1) (dropM s2 m2))
```

Remarque. Le *tilde* dans cette définition est une technicité qui permet de gouverner l'évaluation paresseuse. On impose aussi que la durée de synchronisation du flux tuilé passé en deuxième argument soit nul.

On vérifie facilement qu'une équation de la forme

```
> x = t %\ (re x)
```

peut maintenant être typé. Le type de la tuile `x` est bien te type de la tuile `t`. L'anticipation induite par `(re x)` a été supprimée dans le produit restreint. La position du point d'entrée de synchronisation in est donc uniquement déterminé celui de la tuile `t`.

Bien entendu, ne nombreux produits synchronisés ne semblent pas pouvoir être codé par ce produit restreint. On échouerais donc à résoudre, en toute généralité, le problème de la définition de tuiles récursives ? Il n'en est rien.

On constate que pour tout flux tuilé `t1` et `t2`, en notant `s2` la durée de l'anticipation de synchronisation `syncT(t2)` du flux tuilé `t2`, on a l'équivalence observationnelle

```
t1 % t2 ≡≡
shift s2 ( (shift (-s2) t1) %\ (shift (-s2) t2))
```

Autrement dit, dès lors que les durées d'anticipation sont prévisibles, on peut re-écrire les produit synchronisés en produit restreint `%\` qui se prettent mieux aux définitions récursives. Ainsi, l'équation évoquée au début de ce chapitre pourra être implémentée par la définition récursive équivalente :

```
> x = shift (syncT(t))
>     ((shift (- syncT(t)) t) %\ shift (-syncT(t)))
```

Cette équation peut maintenant être évaluée paresseusement par Haskell. Cette solution se généralise à tout système d'équations de tuiles, de la forme `x = e`, à charge pour le concepteur de préciser lui-même le décalage qu'il faudra appliquer, inductivement, aux membres `e` de ces équations.

9. CONCLUSION

Le T-calcul en Euterpea apparait clairement, via les nombreuses propriétés algébriques qu'il satisfait, comme un formalisme particulièrement robuste pour une description hiérarchique de la structure temporel des flux media temporisés.

D'un point de vue théorique, les flux tuilés sont engendrés à partir des notes et des silences. Muni du produit synchronisé, ils constituent un monoïde inversif. Il pourrait sembler analogue au monoïde inversif libre [28, 26]. Néanmoins, les éléments du monoïdes inversif libre sont des arbres à deux racines. Au contraire, les flux tuilés ressemblent à des sortes de tresses. En effet, l'équivalence observationnelle sur les flux media tuilés unifie toute paire de notes identiques qui se trouvent à une même distance temporelle du point d'entrée de la synchronisation. L'étude de la structure de ces classes de flux tuilés équivalents par observation reste à poursuivre.

De façon plus prosaïque, cette implémentation du T-calcul en Euterpea doit permettre de pouvoir tester le modèle des flux tuilés. En visant tout particulièrement à décrire l'agencement temporel des objects musicaux les uns par rapport aux autres, à l'aide du produit synchronisé et des fonctions de transformations associées (`resync`, `coresync`,
`stretch` et `costretch`), cette implementation pourrait permettre de développer une nouvelle classification des structures rythmiques. En effet, celle ci ne procéderait pas seulement par division du temps comme c'est le cas, par exemple, dans la notation rythmique occidentale, pas plus que par accumulation, comme c'est le cas, par exemple, dans la notation rythmique indienne. Une expérimentation dans cette direction est actuellement en cours.

Ce codage du T-calcul en Euterpea ouvre aussi la voie à une expérimentation artistique. Il est possible que la métaphore des tuiles permettent de mieux rendre compte des intentions d'un compositeur. En particulier, la manipulation de flux musicaux tuilés donnent tous son sens à l'expression musicale *démarrer en même temps*. En effet, cette expression ne signifie aucunement, comme l'illustre la notion d'anacrouse, jouer en même temps la première note, mais, plutôt, se synchroniser sur un même premier temps fort. Une telle expérimentation artistique nécessitera sans doute de produire des fonctions de manipulation adaptées au style de musique envisagée. Les fonctions proposées ici ont une légitimité mathématique qui pourrait manquer de sens musical.

Bien entendu, l'implémentation proposée ici ne traite que des flux musicaux symboliques. Une intégration des flux audio tuilés reste à mettre en oeuvre. Elle pourrait s'appuyer sur la *libTuiles* déjà réalisée [2, 19]. Une telle extension offrirait sans doute un gain de productivité important pour la création et la production de musique électroacoustique. Une bonne partie du mixage, parfois fastidieux et répétitif, peut en effet être factorisé grace à la métaphore du tuilage : chaque tuile audio intègre une fois pour toute, dans ses points de synchronisation, toute l'information necessaire pour positionner, *avant* ou bien *après*, les autres tuiles audio.

10. REFERENCES

[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. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *Electr. Notes Theor. Comput. Sci.*, 11 :1–21, 1998.

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

[7] A. Dicky and D. Janin. Embedding finite and infinite words into overlapping tiles. Research report RR-1475-13, LaBRI, Université de Bordeaux, 2013.

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

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

[10] 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.

[11] 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.

[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. 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.

[14] 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. Springer, 2013.

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

[16] D. Janin. Walking automata in the free inverse monoid. Research report RR-1464-12, LaBRI, Université de Bordeaux, 2013. (revised May 2013).

[17] D. Janin. *Towards a higher dimensional string theory for the modeling of computerized systems*, volume 8327 of *LNCS*, pages 7–20. Springer, 2014.

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

[19] D. Janin, F. Berthaut, and M. DeSainteCatherine. Multi-scale design of interactive music systems : the libTuiles experiment. In *10th Conference on Sound and Music Computing (SMC)*, 2013.

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

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

[22] J. Kellendonk and M. V. Lawson. Universal groups for point-sets and tilings. *Journal of Algebra*, 276 :462–492, 2004.

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

[24] F. Lerdahl and R. Jackendoff. *A generative theory of tonal music*. MIT Press series on cognitive theory and mental representation. MIT Press, 1983.

[25] 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.

[26] W. D. Munn. Free inverse semigroups. *Proceeedings of the London Mathematical Society*, 29(3) :385–404, 1974.

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

[28] H. E. Scheiblich. Free inverse semigroups. *Semigroup Forum*, 4 :351–359, 1972.

[29] J.B. Stephen. Presentations of inverse monoids. *Journal of Pure and Applied Algebra*, 63 :81–112, 1990.