An Efficient Implementation of Tiled Polymorphic Temporal Media

Simon Archipoff

LaBRI
FARM, 2015
Research Context

Tools and method for conception and interpretation of musical performances
We want them:

○ Simple

○ Reliable
The existing: Polymorphic Temporal Media

- Atomic media

- $m_1 :+ m_2$

- $m_1 ::= m_2$
Tiles for multiscale modeling

As example, Bob Dylan’s song “Blowin in the wind”
Music and lyrics have different structures
Tiles for multiscale modeling

As example, Bob Dylan’s song “Blowin in the wind”
Music and lyrics have different structures

How many roads must a man walk down before you call him a man
Tiles for multiscale modeling

As example, Bob Dylan’s song “Blowin in the wind”
Music and lyrics have different structures

How many roads must a man walk down before you call him a man

How can we represent both structures?
Tiling by bars:

How many roads must a man walk down before you call him a man?
Tiling by bars:

How many roads must a man walk down?

Tiling by 4 bars/verses:

How many roads...down before you call him a man?
Tiled Polymorphic Temporal Media

pre γ

polymorphic temporal media

post ↘
Tiled Polymorphic Temporal Media

synchronization

merge

polymorphic temporal media

\[
\begin{align*}
\text{pre } & \gamma \\
\text{post } & \\
\end{align*}
\]
Write a player for tiles that is:

- real-time
- polymorphic (Audio, MIDI, OSC, arbitrary IO, ...)

We start from a syntactic implementation of TPTM
Construction primitives of TPTM

delay :: Duration -> Tile a
event :: a -> Tile a
(%) :: Tile a -> Tile a -> Tile a
Construction primitives of TPTM

delay :: Duration -> Tile a

event :: a -> Tile a

(%) :: Tile a -> Tile a -> Tile a

delay (−2)  

−2 

event e

−2
Syntactic representation of tiles

This syntax describe “zigzag” tiles:

\[
\begin{align*}
 & b \\
 & \quad \rightarrow 2 \quad \rightarrow c \\
 & \quad \quad \rightarrow -5 \quad \rightarrow d \\
 & e \quad \rightarrow 2 \quad \rightarrow f \\
 & \quad \quad \rightarrow 1 \\
\end{align*}
\]

In order to play it we have to order the events:

\[ e, b, f, a, c, d \]
Syntactic representation of tiles

This syntax describe “zigzag” tiles:

In order to play it we have to order the events:
On-the-fly normalization

headT :: Tile a -> Tile a

headT t

tailT :: Tile a -> Tile a

tailT t

\[ t \equiv \text{headT } t \% \text{tailT } t \]
Normal form

\[ \text{norm } t = \text{headT } t \]

\[ \% \text{headT} (\text{tailT } t) \]

\[ \% \text{headT} (\text{tailT}^2 t) \]

\[ \% \text{headT} (\text{tailT}^3 t) \]

\[ \vdots \]

In order to compute a normal form in real time, we need good algorithmic properties for \text{headT} and \text{tailT}.

Syntactic implementations suffer from two problems.
Problem 1: Accumulation of delays

No bound to the number of delays in the syntactic representation

\[ \text{delay } a \% \text{ delay } b \equiv \text{delay} (a + b) \]
Problem 1: Accumulation of delays

No bound to the number of delays in the syntactic representation

\[
\text{delay } a \% \text{ delay } b \equiv \text{delay}(a + b)
\]

Example: normalization with a naive implementation of tailT
Problem 2: right parenthesized tiles

The first event to be played can be the deepest leaf of an imbalanced syntactic tree.
Problem 2: right parenthesized tiles

The first event to be played can be the deepest leaf of an imbalanced syntactic tree.

Example: normalization of a right parenthesized tile
With the proposed implementation

- The number of delay is linear in the number of events (problem 1 solved)
- The structure is balanced (problem 2 solved)
With the proposed implementation

- The number of delay is linear in the number of events (problem 1 solved)
- The structure is balanced (problem 2 solved)

Example: the same right parenthesized tile as before
New implementation principle

A tile is composed of:

- two markers
- a set of event positioned in time relatively to the pre marker
data Tile e = Tile Duration (SHeap e)
New implementation code

```haskell
data Tile e = Tile Duration (SHheap e)
```

The set of event is implemented by Sleator and Tarjan's skew heap:

```haskell
data SHheap a = Empty
  | SHheap Duration (MSet a) (SHheap a) (SHheap a)
```
Correspondence between the heaped implementation and syntactic implementation
Tiled product implementation

\[(\text{Tile } d_1 \ h_1) \% (\text{Tile } d_2 \ h_2) = \text{Tile} \ (d_1 + d_2) \ (\text{mergeSH} \ h_1 \ (\text{shiftSH} \ d_1 \ h_2))\]
Tiled product implementation

\[(\text{Tile } d_1 \ h_1) \% (\text{Tile } d_2 \ h_2) = \text{Tile } (d_1 + d_2) (\text{mergeSH } h_1 (\text{shiftSH } d_1 \ h_2))\]
Tiled product implementation

\[(\text{Tile } d_1 \ h_1) \% (\text{Tile } d_2 \ h_2) = \text{Tile } (d_1 + d_2) \ (\text{mergeSH } h_1 \ (\text{shiftSH } d_1 \ h_2))\]
Tiled product implementation

\[(\text{Tile } d_1 h_1) \% (\text{Tile } d_2 h_2) = \text{Tile } (d_1 + d_2) (\text{mergeSH } h_1 (\text{shiftSH } d_1 h_2))\]
Skew heaps merge

- here the case $d_1 < d_2$
- All rightmost paths are short

- merge following the rightmost path
- swap child so the tree grows from the inside
Amortized complexity

\( n_e \) is the number of events in the tile

\[
\begin{array}{ccc}
\% & \text{headT} & \text{tailT} \\
O(\log(n_e)) & O(1) & O(\log(n_e)) \\
\end{array}
\]

Space complexity: \( O(n_e) \)
A word on infinite tiles

With syntactic encoding:

\[ \sum -d_i = \bot \]
A word on infinite tiles

With syntactic encoding:

\[ \sum -d_i = \bot \]

With our implementation:
Thank you 😊