Structured Reactive Programming with Polymorphic Temporal Tiles

S. Archipoff & D. Janin,
Bordeaux INP, UMR CNRS LaBRI,
Inria BSO, University of Bordeaux
@ICFP/FARM, Nara, Japan,
2016
This work is dedicated to the memory Paul Hudak.

Our proposal, implemented in Haskell, eventually result from combining ideas both from Functional Reactive Programing and Polymorphic Temporal Media.
1. Opening

In a world where every computed object is rendered in time

programming language constructs should derive from mathematical properties that hold in this world... and not the opposite which is very likely to fail...
Research context

Goal
Yet another programing language for reactive temporal media systems

Main expected features
- abstract enough (structured for user),
- softly realtime (timed over real passing time),
- usable on stage (reliable),
- pervasive (mathematically robust).
Research context
Programing languages for the design of multimedia reactive systems

Existing proposals

- Functional reactive programing (reactive & timestamped),
- Polymorphic Temporal Media (structured & algebraic),
- Synchronous languages (fast & robust),
- Timed IO-automata (well-defined & checkable),
- others . . .

but mostly incomparable!
Our proposal

DSL proposal

A model-based approach: three layers

- Input-Output streams: Timed event lists (back-end),
- Polymorphic timed streams: Queue lists (mid-end),
- Handy data types: Temporal tiles (front-end).

justified by category theoretic and algebraic properties.

Moto: The more mathematically robust, the easier to use and the longer to last.
2. Queue lists

In a world where every computed object is rendered in time, what they are and when they are combine nicely!
Queue lists (Basic)
Semantics model (almost FRP)

Let $d$ be a type for durations, i.e. reals (continuous) or integers (discrete) extended with $+\infty$.

Let $a$ be a type for temporal values, i.e. to make it “simple”: pairs $\text{duration} \times \text{value}$

A queue list is a mapping

$$q :: d^* \rightarrow \mathcal{P}(a)$$

where $d^*$ denotes zero or positive durations understood as passing time from origin.
Queue lists (Basic)

A queue list $q$ example with relative durations.

\[ \begin{align*}
(3, a) &\rightarrow (5, b) \\
(3, c) &\rightarrow (2, d) \\
(3, e) &\rightarrow (4, a)
\end{align*} \]
Constructors

- **fromAtomsQ**: \( \mathcal{P}(a) \rightarrow \text{QList } d \ a \)
- **shiftQ**: \( d^* \rightarrow \text{QList } d \ a \rightarrow \text{QList } d \ a \)
- **mergeQ**: \( \text{QList } d \ a \rightarrow \text{QList } d \ a \rightarrow \text{QList } d \ a \)

with associated **syntactical normal form**.

Getters

- **atomsQ**: \( \text{QList } d \ a \rightarrow \mathcal{P}(a) \) \( \quad \begin{cases} \text{headQ} \end{cases} \)
- **delayToTailQ**: \( \text{QList } d \ a \rightarrow d^* \)
- **tailQ**: \( \text{QList } d \ a \rightarrow \text{QList } d \ a \)

with a **list flavor**.
A queue list $q$ example with relative durations with \textit{atoms, delay to tail} and \textit{tail}.

\begin{itemize}
\item $4$ \textcolor{red}{$(3, a)$}
\item $2$ \textcolor{green}{$(2, d)$}
\item $4$ \textcolor{red}{$(5, b)$}
\item $2$ \textcolor{green}{$(3, c)$}
\item $2$ \textcolor{green}{$(3, e)$}
\end{itemize}
Queue lists (Basic)
with some (quick checkable) invariants

Head/tail invariants with durations

\[ \text{atomsQ} \circ \text{fromAtomsQ} \ a =\ a \] (1)

\[ \text{mergeQ} \ (\text{fromAtomsQ} \circ \text{atomsQ} \ q) \]
\[ (\text{shiftQ} \ (\text{delayToTailQ} \ q) \ (\text{tailQ} \ q)) = q \] (2)

The meaning of delay to tail

\[ \text{if} \ \ \text{delayToTailQ} \ q = 0 \ \ \text{then} \ q = \text{emptyQ} \] (3)

with \( \text{emptyQ} = \text{fromAtomsQ} \ \emptyset \):
Queue lists (Categorical properties)

General warning
We are dealing with temporal types...with associated durations.

Trick
Restrict to duration preserving functions.

Duration (or life expectancy)
Default duration (e.g. for queue list) is “infinite”...
Queue lists (Categorical properties)

Functor
Single point to point application
\[ \text{fmapQ} :: (a \rightarrow b) \rightarrow \text{QList} \, d \, a \rightarrow \text{QList} \, d \, b \]

For classical usage

Applicative Functor
More and more merged applications
\[ \langle \ast \rangle_Q :: \text{QList} \, d \, (a \rightarrow b) \rightarrow \text{QList} \, d \, a \rightarrow \text{QList} \, d \, b \]

with \( \text{pureQ} = \text{fromAtomQ} \).
A bit weird?
Queue lists (Categorical properties)

Monad
Merging sub-queue lists into a single one

\[ \text{joinQ} :: \text{QList} \ d \ (\text{QList} \ d \ a) \to \text{QList} \ d \ a \]

with \( \text{returnQ} = \text{fromAtomQ} \)

and substituting named time slots by queue lists

\[ \text{bindQ} :: \text{QList} \ d \ a \to (a \to (\text{QList} \ d \ b)) \to \text{QList} \ d \ b \]

with \( \text{bindQ} \ q \ f = \text{joinQ} (\text{fmapQ} \ f \ q) \).

A flavor of conception by refinement?
Queue lists (Categorical properties)

Product: \( QList \, d \, (a + b) \)

Forking queue list transforms.

\[
\text{factorPQ} :: (QList \, d \, c \rightarrow \, QList \, d \, a) \rightarrow (QList \, d \, c \rightarrow \, QList \, d \, b) \\
\quad \rightarrow (QList \, d \, c \rightarrow \, QList \, d \, (a + b))
\]

\[
\begin{array}{c}
\text{f} \\ \downarrow \\
\text{g}
\end{array} \\
\begin{array}{c}
\text{QList} \, d \, c \\
\downarrow \\
\text{QList} \, d \, a \\
\downarrow \\
\text{QList} \, d \, (a + b)
\end{array}
\]

with projections

\[
\begin{align*}
\text{fromLeftQ} & : \, QList \, d \, (a + b) \rightarrow \, QList \, d \, a \\
\text{fromRightQ} & : \, QList \, d \, (a + b) \rightarrow \, QList \, d \, b
\end{align*}
\]

a flavor of asynchronous data-flow programming?
Queue lists (Categorical properties)
Restricting further to emptyQ preserving functions.

Weak sum: $\textbf{QList} \ d \ (a + b)$
Joining two queue list transforms.

$\text{factorSQ} :: (\textbf{QList} \ d \ a \rightarrow \textbf{QList} \ d \ c) \rightarrow (\textbf{QList} \ d \ b \rightarrow \textbf{QList} \ d \ c) \rightarrow (\textbf{QList} \ d \ (a + b) \rightarrow \textbf{QList} \ d \ c)$

with injections

$\text{toListQ} : \textbf{QList} \ d \ a \rightarrow \textbf{QList} \ d \ (a + b)$
$\text{toRightQ} : \textbf{QList} \ d \ b \rightarrow \textbf{QList} \ d \ (a + b)$

a stronger flavor of asynchronous data-flow programming?
Queue lists (Categorical properties)

Exponent
Applying distinct transforms over time

\[
\text{applyQ} :: \text{QList} \, d \, (\text{QList} \, d \, a \rightarrow \text{QList} \, d \, b) \rightarrow \text{QList} \, d \, a \rightarrow \text{QList} \, d \, b
\]

a flavor of dynamic changes of transforms

An application architecture example

```
<table>
<thead>
<tr>
<th>GUI</th>
<th>QList d int</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piano In</td>
<td>QList d Midi</td>
</tr>
<tr>
<td></td>
<td>QList d Midi</td>
</tr>
<tr>
<td>Pianio Out</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Bind</th>
<th>QList d (QList d Midi \rightarrow QList d Midi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apply</td>
<td>QList d Midi</td>
</tr>
<tr>
<td></td>
<td>QList d Midi</td>
</tr>
</tbody>
</table>
```

```
3. Reactive kernel

In a world where every computed object is rendered in time, object rendering is controlled by events!
Reactive kernel

Expected runtime architecture

With temporal values parenthesized by pairs of *On* and *Off* events:

\[
\text{eventToQList} \rightarrow \text{applyQListFunc } f \rightarrow \text{tileToEvent}
\]

Input: well parenthesized pairs of *On* and *Off* events

Program: a function \( f \)

\( \text{QList } d \ iv \rightarrow \text{QList } d \ ov \)

Output: well parenthesized pairs of *On* and *Off* events
Reactive kernel
The need of unknowns

**Unknown duration**
In a reactive context, unknown durations arises from two sides:
- duration of *timed values* from *On* to *Off* events,
- duration of *delay to tail* between successive *On* events.

**Unknown tails**
In a reactive context, the ail of the input queue list tail is (recurrently) unknown.
Reactive kernel
Frozen application and updates

Frozen application
An application $f \ p :: QList \ d \ a$ may need to be frozen in some additional constructor

$$QRec \ f \ a :: QList \ d \ a$$

with partial known argument $p$.

Class type $Updatable \ (d, a) \ t$
Frozen argument $p :: t$ need to be updated. An Haskell class type

$$Updatable \ (\ldots) \ t$$

closed under usual type constructs, provides generic update functions both for unknown durations and unknown tail.
Reactive kernel
Resulting runtime architecture

```
<table>
<thead>
<tr>
<th>Input:  pairs of On iv and Off iv events</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program: function f</td>
</tr>
<tr>
<td>QList d iv → QList d ov</td>
</tr>
<tr>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>Output:  pairs of On ov and Off ov events</td>
</tr>
<tr>
<td>-----------------------------------------</td>
</tr>
</tbody>
</table>

Current implementation: The running function f can effectively be built with all primitive and categorical constructors previously defined.

Warning: non causal functions are easily definable...
4. Temporal Tiles

In a world where every computed object is rendered in time, synchronization delays match durations.
Temporal Tiles
Primitive temporal tiles

Temporal values: from temporal values \((d, v)\)
Values rendered from \(\Upsilon\) to ↓.

\[
\begin{align*}
\Upsilon & \quad \xrightarrow{(d, v)}
\end{align*}
\]

Delays
Temporal values with no value.

\[
\begin{align*}
\Upsilon & \quad \xrightarrow{d}
\end{align*}
\]
Temporal Tiles

Additive operators

Sum (synchronisation) : \( x + y \)

Negation (sync. inversion) : \( -x \)

Difference (generalized sync.) : \( x - y \)
Temporal Tiles

Implementation

Synchronization syntactic sugar over queue lists

```
data Tile d v = Tile d d (QList d v)
```
Temporal Tiles

A code example

Tiled sum

\[
\text{plusT} \ (\text{Tile} \ d1 \ \text{ad1} \ q1) \ (\text{Tile} \ d2 \ \text{ad2} \ q2) = \\
\quad \text{let } dt = \text{ad1} - (d1 + \text{ad2}) \\
\quad \quad d = d1 + d2 \\
\quad \quad \text{case } (\text{compare} \ 0 \ dt) \ \text{of} \\
\quad \quad \quad \text{LT } \to \ \text{Tile} \ d \ \text{ad1} \\
\quad \quad \quad \quad \quad (\text{mergeQ} \ q1 \ (\text{shiftQ} \ dt \ q2)) \\
\quad \quad \quad \text{EQ } \to \ \text{Tile} \ d \ \text{ad1} \\
\quad \quad \quad \quad \quad (\text{mergeQ} \ q1 \ q2) \\
\quad \quad \quad \text{GT } \to \ \text{Tile} \ d \ (\text{ad1} - dt) \\
\quad \quad \quad \quad \quad (\text{mergeQ} \ (\text{shiftQ} \ (-dt) \ q1) \ q2)
\]

Temporal tiles are really just a front-end to queue lists programing!
Temporal Tiles
Algebraic properties I

Delays encode durations
Delays with addition and negation are in one-to-one correspondance with durations.

The additive tile algebra
For every tile $x$, the tile $-x$ is the unique tile $y$ such that

$$x + y + x = y \quad \text{and} \quad y + x + y = y$$

With the zero delay 0, temporal tiles with sum form an inverse monoid.
Temporal Tiles

Multiplicative operators

Reset and coreset: \( re(x) = x - x \) and \( co(x) = -x + x \)

Stretch: \( f \cdot (\text{Tile } d \cdot a \cdot dq) = \text{Tile } (f \ast d) \cdot (f \ast ad) \cdot (\text{stretch}\ Q \cdot fq) \)

Product: \( x \ast y = re(\text{stretch}(|y|, x)) + \text{stretch}(|y|, x) \)
Temporal Tiles
Algebraic properties II

The multiplicative tile algebra
In the case a tile $x$ is of non zero duration, define

$$\frac{1}{x} = (1/|x|^2) \ast x$$

Then, for every tile $x$ on non zero duration, the tile $1/x$ is the unique tile $y$ such that

$$x \ast y \ast x = y \quad \text{and} \quad y \ast x \ast y = y$$

With the unit delay 1, temporal tiles with product form a commutative inverse monoid.

Tile syntax
Tiles with sum and product are Num and Frac instances, i.e. no additional syntax needed!
Temporal Tiles

Algebraic properties III

**Categorical properties**

Categorical properties of queue lists can be lifted to tiles.

**T-calculus**

The resulting DSL prototype, developed in Haskell over UISF, has been released in $\alpha$ version.
5. Conclusion
Conclusion

Model based design

Specialized UI

Tile (Models)

T-calcul (Programs)

System (Apps)

Gestures, motions, controls, ...  
Music, Video, Animation, ...
Conclusion

What is done

▶ a robust and versatile model for combining timed values,
▶ an robust reactive/realtime functional language front-end,
▶ a implementation prototype in Haskell for live experiments,

What remains to be done

▶ better runtime with automatic freeze/unfreeze (scheduling),
▶ assisted analysis of temporal causality (not too fast),
▶ assisted analysis of memory needs (not too slow),
▶ more experiments...and many more questions...
Conclusion

What is done

► a robust and versatile model for combining timed values,
► an robust reactive/realtime functional language front-end,
► a implementation prototype in Haskell for live experiments,

What remains to be done

► better runtime with automatic freeze/unfreeze (scheduling),
► assisted analysis of temporal causality (not too fast),
► assisted analysis of memory needs (not too slow),
► more experiments...and many more questions...
Conclusion

In a world where every computed object is rendered in time

many things have already been observed,

but not necessarily by the same observer.