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

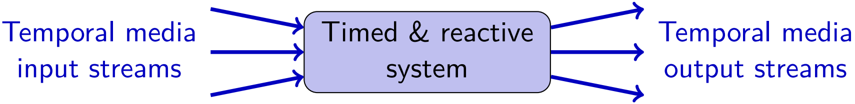
In a world where every computed object is rendered in time

programing 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 duration $\times$ value

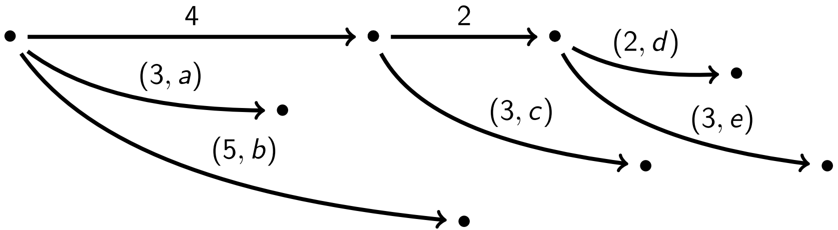
A queue list is a mapping

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

where d^* denotes zero or positive durations
understood as passing time from origin.

Queue lists (Basic)

A queue list q example with relative durations.



Queue lists (Basic)

Constructors and getters

Constructors

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

with associated syntactical normal form.

Getters

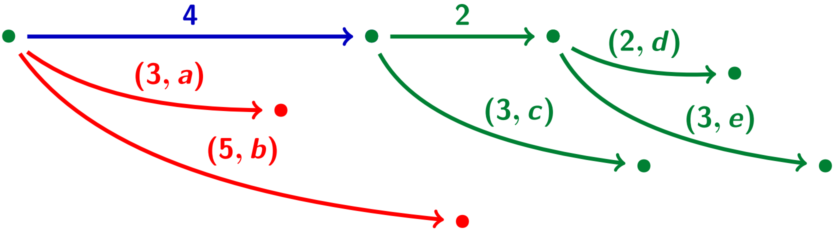
$$\left.\begin{array}{l} \blacktriangleright\ atomsQ :: QList\ d\ a \to \mathcal{P}(a) \\ \blacktriangleright\ delayToTailQ :: QList\ d\ a \to d^* \end{array}\right\} headQ$$

- $tailQ :: QList\ d\ a \to QList\ d\ a$

with a list flavor.

Queue lists (Basic)

A queue list q example with relative durations with **atoms**, **delay to tail** and **tail**.



Queue lists (Basic)

with some (quick checkable) invariants

Head/tail invariants with durations

$$atomsQ \circ fromAtomsQ\ a == a \tag{1}$$

$$\begin{aligned} &mergeQ\ (fromAtomsQ \circ atomsQ\ q) \\ &\qquad (shiftQ\ (delayToTailQ\ q)\ (tailQ\ q)) == q \end{aligned} \tag{2}$$

The meaning of delay to tail

$$\text{if} \quad delayToTailQ\ q == 0 \quad \text{then} \quad q == emptyQ \tag{3}$$

with $emptyQ = fromAtomsQ\ \emptyset$:

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

$$fmapQ :: (a \rightarrow b) \rightarrow QList\ d\ a \rightarrow QList\ d\ b$$

For classical usage

Applicative Functor

More and more merged applications

$$<*>_Q :: QList\ d\ (a \rightarrow b) \rightarrow QList\ d\ a \rightarrow QList\ d\ b$$

with $pureQ = fromAtomQ$.

A bit weird ?

Queue lists (Categorical properties)

Monad

Merging sub-queue lists into a single one

$$joinQ :: QList\ d\ (QList\ d\ a) \rightarrow QList\ d\ a$$

with $returnQ = fromAtomQ$

and substituting named time slots by queue lists

$$bindQ :: QList\ d\ a \rightarrow (a \rightarrow (QList\ d\ b)) \rightarrow QList\ d\ b$$

with $bindQ\ q\ f = joinQ\ (fmapQ\ f\ q)$.

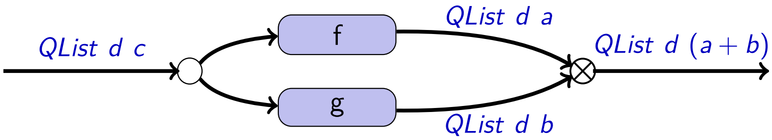
A flavor of conception by refinement ?

Queue lists (Categorical properties)

Product: $\boldsymbol{QList\ d\ (a+b)}$

Forking queue list transforms.

$$\begin{aligned} factorPQ :: (QList\ d\ c \to QList\ d\ a) \to (QList\ d\ c \to QList\ d\ b) \\ \to (QList\ d\ c \to QList\ d\ (a+b)) \end{aligned}$$



with projections

$$\begin{aligned} fromLeftQ &: QList\ d\ (a+b) \to QList\ d\ a \\ fromRightQ &: QList\ d\ (a+b) \to QList\ d\ b \end{aligned}$$

a flavor of asynchronous data-flow programming ?

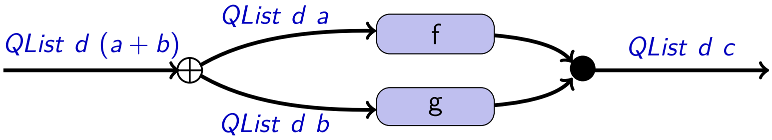
Queue lists (Categorical properties)

Restricting further to emptyQ preserving functions.

Weak sum: $\boldsymbol{QList\ d\ (a+b)}$

Joining two queue list transforms.

$$\begin{aligned} factorSQ :: (QList\ d\ a \to QList\ d\ c) &\to (QList\ d\ b \to QList\ d\ c) \\ &\to (QList\ d\ (a+b) \to QList\ d\ c) \end{aligned}$$



with injections

$$toLeftQ : QList\ d\ a \to QList\ d\ (a+b)$$
$$toRightQ : QList\ d\ b \to QList\ d\ (a+b)$$

a stronger flavor of asynchronous data-flow programming ?

Queue lists (Categorical properties)

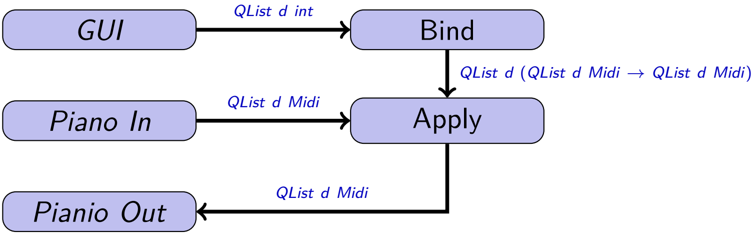
Exponent

Applying distinct transforms over time

$$applyQ :: QList\ d\ (QList\ d\ a \to QList\ d\ b) \to QList\ d\ a \to QList\ d\ b$$

a flavor of dynamic changes of transforms

An application architecture example



3. Reactive kernel

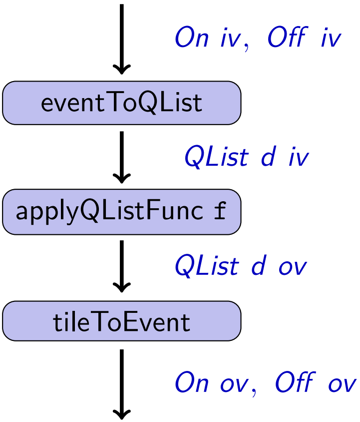
In a world where every computed object is rendered in time

object rendering is controlled by events !

Reactive kernel

Expected runtime architecture

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



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

Program : a function *f*
QList d iv → *QList d ov*

Output : well parenthesized pairs of *On ov* and *Off ov* events

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 $Updatable\ (d, a)\ t$

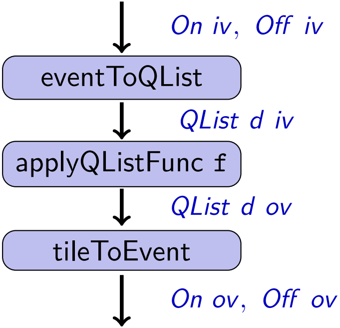
Frozen argument $p :: t$ need to be updated. An Haskell class type

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

closed under usual type constructs, provides generic update functions both for unknown durations and unknown tail.

Reactive kernel

Resulting runtime architecture



Input : pairs of *On iv* and *Off iv* events

Program : function f
$QList\ d\ iv \to QList\ d\ ov$

Output : pairs of *On ov* and *Off ov* events

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 matches durations

Temporal Tiles

Primitive temporal tiles

Temporal values : from temporal values $\boldsymbol{(d, v)}$

Values rendered from Υ to $\downarrow$.

$$\overset{\Upsilon}{\bullet} \xrightarrow{(d, v)} \underset{\downarrow}{\bullet}$$

Delays

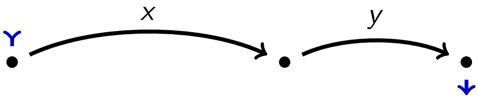
Temporal values with no value.

$$\overset{\Upsilon}{\bullet} \xrightarrow{d} \underset{\downarrow}{\bullet}$$

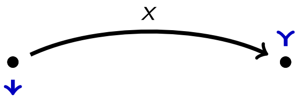
Temporal Tiles

Additive operators

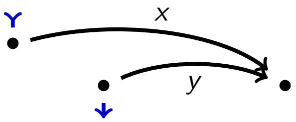
Sum (synchronisation) : $x + y$



Negation (sync. inversion) : $-x$

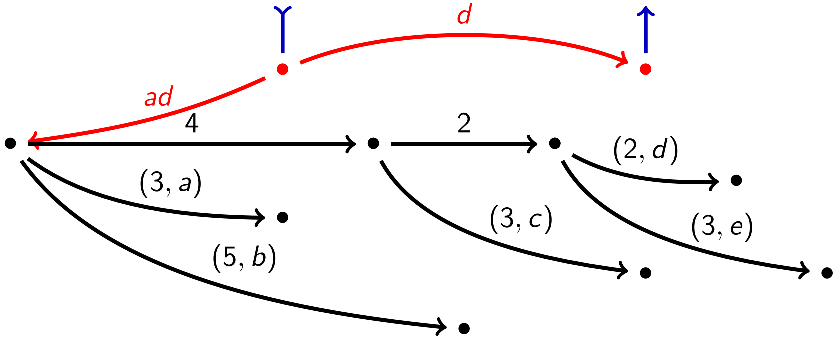


Difference (generalized sync.) : $x - y$



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

```
plusT (Tile d1 ad1 q1) (Tile d2 ad2 q2) =
    let dt = ad1 - (d1 + ad2)
        d = d1 + d2
    case (compare 0 dt) of
        LT -> Tile d ad1
                (mergeQ q1 (shiftQ dt q2))
        EQ -> Tile d ad1
                (mergeQ q1 q2)
        GT -> Tile d (ad1 - dt)
                (mergeQ (shiftQ (-dt) q1) q2
```

Temporal tiles are really just a front-end to queue lists programing !

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

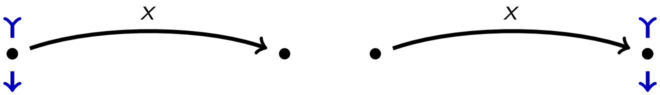
$$x + y + x = y \quad \text{and} \quad y + x + y = y$$

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

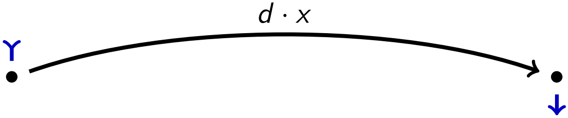
Temporal Tiles

Multiplicative operators

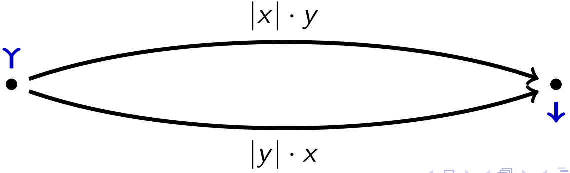
Reset and coreset : $re(x) = x - x$ and $co(x) = -x + x$



Stretch : $f \cdot (Tile\ d\ \ a\ dq) = Tile\ (f * d)\ (f * ad)\ (stretchQ\ fq)$



Product : $x * y = re(stretch(|y|, x)) + stretch(|y|, x)$



Temporal Tiles

Algebraic properties II

The multiplicative tile algebra

In the case a tile x is of non zero duration, define

$$1/x = (1/|x|^2) * x$$

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

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

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 !

Categorical properties

Categorical properties of queue lists can be lifted to tiles.

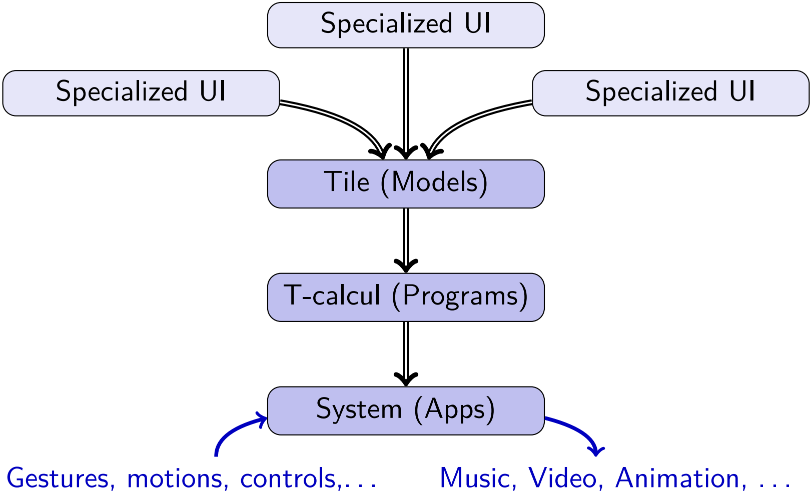
T-calculus

The resulting DSL prototype, developed in Haskell over UISF, has been released in α version.

5. Conclusion

Conclusion

Model based design



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.