Tidalcycles for Programmers: Semantics of the Pattern type

Johannes Waldmann

Leipzig, 29. Sept. 2026
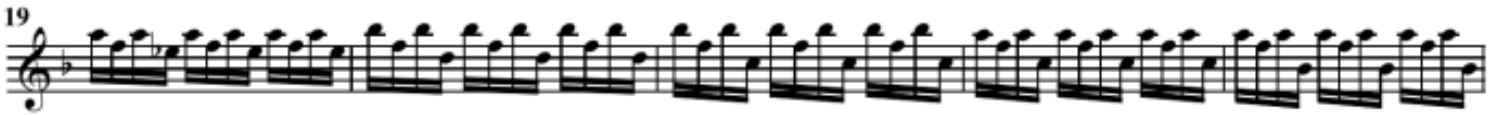
Tidalcycles (Goal, Example, Concepts)

- improvise live music with sourcecode, using the computer language as a musical environment.
  Alex McLean (2009) `https://tidalcycles.org/`
- `d1 $ fast (pure 2) $ cat [ s "bd", s "sd" ]`
  `cat :: [Pattern a] -> Pattern a`
  `type ValueMap = Data.Map.Map String Value`
  typical keys are `s`(ample name), `gain`, `speed`, `distort`
  `d1 :: Pattern ValueMap -> IO ()`
- `p :: Pattern a` represents set of events, where event has extension `:: (Time, Time)` and value `:: a`
- tidalcycles engine: continually compute events from current pattern for current time slice, send to audio engine (supercollider) for rendering to sound
- tidalcycles live coding: ghci command `d1 $ p` sets current pattern for channel 1

Patterns (Definition and basic operations)

- `p :: Pattern a` represents set of events, where event has extension `:: (Time, Time)` and value `:: a`
- describe an elementary pattern:
  - no events: `silence :: Pattern a`
  - unit pattern: `pure :: a -> Pattern a`
    `pure x` has events $\{((t, t+1), x) \mid t \in \mathbb{Z}\}$
- transform a pattern:
  - transform values (only, keep time structure)
    `fmap :: (a -> b) -> Pattern a -> Pattern b`
  - transform time (keep values) `rotL, rotR, _fast, _slow`
    `:: Time -> Pattern a -> Pattern a`
- combine several patterns (just time)
  - in parallel (union of events)
    `stack :: [Pattern a] -> Pattern a`
  - interleaving (round-robin scheduling)
    `cat :: [Pattern a] -> Pattern a`
- combine several patterns (time and values)

Interleaving of patterns

- there is *no sequential composition* (one after the other), since *patterns have no end, and no beginning*
- `cat :: [Pattern a] -> Pattern a` interleaves patterns:
  $p = \mathtt{cat}[p_0, p_1, \ldots, p_{n-1}]$ where
  $\forall k \in \mathbb{Z}, 0 \leq i < n, 0 \leq d < 1 : p(k \cdot n + i + d) = p_i(k + d)$
- 

  J. S. Bach: Partita, BWV 1006, measures 20 to 23:

```
fast 12 $ cat
 [ slow 12 $ cat [ fast 12 bf, fast 12 a ]
 , cat [ f
       , slow 3 $ cat
           [ fast 3 d,fast 3 c,fast 3 c,fast 3 bf ]]]
```

  where `bf` is unit pattern for note b-flat, etc.

Combine structure and values (at the same time)

- if we think `Pattern a = (Time -> a)`
  (a *behaviour* in functional reactive programming, FRP)
  then $p \texttt{<*>} q = \lambda t.p(t)(q(t))$
- for multi-behaviours (?), three implementations of
  `(<*>) :: Pattern (a -> b) -> Pattern a -> Pattern b`
  $p \texttt{<*>} q =$
  $$\{(c(\text{ext}l)(\text{ext}r), (\text{val}l)(\text{val}r)) \mid l \in p, r \in q, g(\text{ext}l)(\text{ext}r)\}$$
  functions c (combine) and g (guard) describe structure
  - from left (sample right value at start of event from left)
    $cab = a, gab = \text{start}a \in b$
  - from right (sample left value at start of event from right)
    $cab = b, gab = \text{start}b \in a$
  - from both $cab = a \cap b, gab = (\emptyset \neq (a \cap b))$
- Exercise: do these respect the laws for Applicative? e.g.,
  - Identity: `pure id <*> v = v`,
  - Interchange: `u <*> pure y = pure (\f -> f y) <*> u`

  (no. w.r.t. some equivalence? with a different `pure`?)

Patterns of Patterns, or: Join

- when composing music, we might have
  - global structure `p :: Pattern a`,
  - value-dependent local structure `f :: a -> Pattern b`,

  we want `(p >>= f) :: Pattern b = join (fmap f p)`
  with `join :: Pattern (Pattern b) -> Pattern b`
- `instance Monad (a -> _)` has $\texttt{join}(f) = \lambda t.f(t)(t)$
  for `Pattern`, we can do similar $\texttt{join}(p) =$
  $\{(c(\text{ext}o)(\text{ext}i), \text{val}i) \mid o \in p, i \in \text{val}o, g(\text{ext}o)(\text{ext}i)\}$
  three variants for c(ombine), g(uard), as for `(<*>)`
  - `outerJoin`: from left
  - `innerJoin`: from right
  - `unwrap`: from both

  plus one extra variant with (affine) transformed time:
  - `squeezeJoin`: the unit interval of i becomes ext$o$.
- Exercise: check Monad laws for these joins.
- use `squeezeJoin` to implement `(>>=)` (bind) to implement `(<*>)`. Use it for examples! Does this operator already exist?

Summary, Discussion

- Tidalcycles: eDSL for patterns of time-dependent values
- semantics of pattern in Tidalcycles
  - is a mixture of Behaviour and Event from functional reactive programming and extends it by ...
  - multi-Behaviours (overlapping intervals)
  - transformation of time (independent transformation per sub-pattern, Eigenzeit)
    (Time = Rational, transform needs to be invertible and continuous, affine seems the only way)
- we have ignored here: implementation ("set of event" currently implemented as function from interval to list of events contained in/intersecting with? that interval)
- we have also ignored here
  - separate surface language (mininotation) (ad-hoc syntax, no types, no names (let), no functions (lambda), no library functions)
  - pragmatics (how to use Tidalcycles to make interesting music)

Pattern – current implementation (tidal-core-1.10.2)

- "set of event" as function from interval `arc s` to list of events

```
data Pattern a =
  Pattern {query :: State -> [Event a], ... }
data State = State { arc :: Arc, ... }
type Arc = ArcF Time
data ArcF a = Arc { start :: a, stop :: a }
type Event a = EventF (ArcF Time) a
data EventF a b = Event { part :: a,  value :: b, ... }
```

- . . . contained in that interval? a specification is missing, like

```
forall (p : Pattern a) (a : Arc) (e : query p a)
  -> start (part e) `in` a
```

  and monotonicity: $a \subseteq a' \Rightarrow \text{query } p\ a \subseteq \text{query } p\ a'$

- on the other hand, invalid patterns make the back-end glitch

```
d1  $ withinArc (Arc (1/4) (3/4))
   (withResultTime (/  2) . withQueryTime (/ 2))
  $ s "superpiano" + note (scale "major" $ run 16 - 12)
```