TidalCycles: Continuous and Discrete Patterns of Functional Reactive Programming

ANONYMOUS AUTHOR(S)*

Abstract goes here.

CCS Concepts: • **Applied computing** → **Performing arts**; **Sound and music computing**; • **Theory of computation** → **Models of computation**; • **Software and its engineering** → **Domain specific languages**; **Functional languages**.

Additional Key Words and Phrases: pure functional reactive programming, FRP, music representation, algorithmic pattern

ACM Reference Format:
Anonymous Author(s). 2024. TidalCycles: Continuous and Discrete Patterns of Functional Reactive Programming. In . ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

1 REPRESENTING PATTERNS IN FUNCTIONA REACTIVE PROGRAMMING

Functional Reactive Programming (FRP) was first implemented by Conal Elliot in Fran [1], based on a formal definition of behaviour as a continuous function of time. Popular implementations of FRP (e.g. the Elm language) have followed which have instead opted for discrete rather than continuous semantics. The following introduces an approach that supports both discrete and continuous time, developed and popularised over the past ten years in the TidalCycles system, which is designed for creative, live exploration of musical (and other) patterns.

In the following I will step us through the process of building a representation, taking Elliot's definition of behaviour in Fran as a worthy starting point, and the definition of pattern in TidalCycles as an end point (with some simplifications). This is done in a practical, literate programming style, using the Haskell programming language. Example patterns are visualised, generated directly from the code in this paper.

Our starting point then, is the `Behaviour` type from Fran:

```
type Time = Double
type Behaviour a = Time -> [a]
```

This represents behaviours elegantly as functions of time, or in other words, time-varying values. There might be more than one value active at a particular point in time, hence these behaviours returns a list of values. Fran also defines an event model, but here we focus entirely on pure behaviours that have no state beyond time.

1.1 Rational time

According to Elliot, time in FRP should be *real*, therefore having arbitrary levels of precision, without building a notion of samplerate into the representation itself. In practice, Fran uses double-precision floating point numbers for representing time. Floating point numbers are certainly efficient, but

ICFP '24, September 02–07, 2024, Milan, IT

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

leave us with the problem of how to deal with associated errors in time calculations, for instance by building some tolerance into comparison between values.

To avoid such floating point headaches, here I instead use rational time as a practical alternative. Unlike floating point numbers, this will support ratios that are common in media arts, e.g. durations of 1/3 for triplets in music, and 1/24 for frame frequencies in video animation.

```
type Time = Rational
type Pattern a = Time -> [a]
```

But this raises a question – what are these numbers a ratio *of*? The answer is metric cycles, which in music have particular meaning depending on the musical tradition at play. For example in Western classical music, metric cycles are referred to as measures or bars, and in Indian classical music as the nuanced structures of Tala cycles. In TidalCycles they are simply referred as cycles, and the integer timeline (i.e., those ratios of 1) marks out the endings and beginnings of successive cycles. For example, a ratio of 4/10 would fall halfway through the third cycle, counting from zero.

I have also renamed the `Behaviour` type to `Pattern` in the above. This differentiates our type from Elliot's, and connects our type with the long history of pattern-making.

1.2 Timespans and events

Next, in order to support discrete events, I introduce time*spans* to the representation.

```
data TimeSpan = TimeSpan {begin :: Time, end :: Time}
data Event a = Event {active :: TimeSpan, value :: a}
type Pattern a = TimeSpan -> [Event a]
```

A pattern is now a function of timespans to events, and each event is active for a timespan. A pattern can then be queried with contiguous timespans, avoiding any chance of missing events that that would fall between queries of single time values.

However, we also need to take into account that an event might well not fit within the timespan of a given query. We often need to know *which* part of an event is active within the queried timespan, and whether it started before and/or continues beyond that timespan. For this reason, I add another field to our Event datatype called `whole`.

```
data Event a = Event {whole :: TimeSpan, active :: TimeSpan, value :: a}
```

The 'whole timespan of the event must either be the same as or greater than the `active` part, and always include it. We use the `active` timespan to see when an event is active during the query, but can compare it with the `whole` to check whether the event is a fragment of a larger timespan.

Now the `Pattern` type can represent discrete events, but it would be best if it could still represent continuous values. Fundamentally, a continuous value is one which does not have a discrete beginning and end. So to represent them, we simply need to make the 'whole' optional, using Haskell's standard `Maybe` type.

```
data Event a = Event {whole :: Maybe TimeSpan, active :: TimeSpan, value :: a}
```

We can now tell when an event is continuous, because its `whole` is set to `Nothing`.

The types are now complete, as follows.

```
type Time = Rational
data TimeSpan = TimeSpan {begin :: Time, end :: Time}
    deriving Show
data Event a = Event {whole :: Maybe TimeSpan, active :: TimeSpan, value :: a}
```

```
    deriving (Show, Functor)
data Pattern a = Pattern {query :: TimeSpan -> [Event a]}
    deriving (Functor)
```

2 CONSTRUCTING PATTERNS

How does this work in practice? Let's have a look at how simple patterns are defined. For continuous patterns, we simply sample a value at the halfway point of the queried timespan. For example, a sinewave with a period of one cycle:

```
sinewave :: Pattern Double
sinewave = Pattern $ \timespan ->
  [Event Nothing timespan $ sin $ (fromRational $ t timespan) * pi * 2]
    where t timespan = begin timespan + ((end timespan - begin timespan) / 2)
```

Discrete patterns need to calculate both the whole and active timespans for each event. The following `atom` function returns a pattern that repeats the given value as a discrete event, every cycle. To do this it splits the query at cycle boundaries, and sets the whole to be the beginning and end of the cycle for each event. A note on terminology - in Indian music, the `sam` is the beginning of a cycle (and end of the previous one).

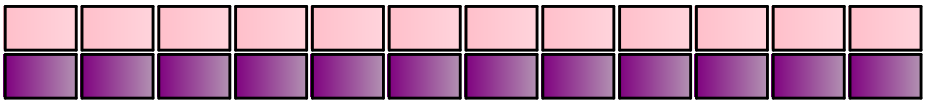
```
sam, nextSam :: Rational -> Rational
sam t = toRational (floor t :: Int)
nextSam = (1 +) . sam

splitSpan :: TimeSpan -> [TimeSpan]
splitSpan (TimeSpan s e) | sam s == sam e || n == e = [TimeSpan s e]
                         | otherwise = TimeSpan s n : splitSpan (TimeSpan n e)
    where n = nextSam s

atom :: a -> Pattern a
atom value = Pattern $ map (\timespan ->
                              Event (Just $ TimeSpan (sam $ begin timespan)
                                                     (nextSam $ begin timespan)
                                    )
                                    timespan
                                    value
                           ) . splitSpan
```

3 COMPOSING PATTERNS

Patterns are not too much use until they are composed together, and because they are functions, we of course need to do functional composition, by making a new function that calls two or more existing ones. First, here is a straightforward `stack` function for composing patterns together so that they run concurrently:

```
stack :: [Pattern a] -> Pattern a
stack pats = Pattern $ \timespan -> concatMap (`query` timespan) pats
```

```
fig1 = stack [atom pink, atom purple]
```



The above visualises twelve cycles of the pattern over time, from left to right. Combining patterns in sequence over time is a little complicated because patterns have infinite length, so we are not able to simply concatenate them. Instead, we can 'interlace' their cycles.

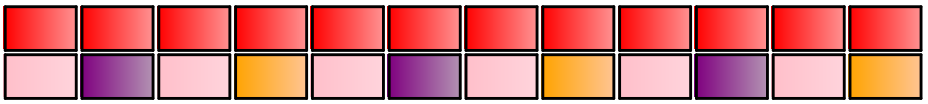
```
splitQueries :: Pattern a -> Pattern a
splitQueries pat =  Pattern $ concatMap (query pat) . splitSpan

-- Take a cycle from each pattern in turn
interlace :: [Pattern a] -> Pattern a
interlace pats = splitQueries $ Pattern f
  where f timespan = query (_late offset pat) timespan
              where n = toRational $ length pats
                    cyc = sam $ begin timespan
                    pat = pats !! floor (cyc `mod'` (n :: Rational))
                    offset = cyc - sam (begin timespan / n)

-- The same as interlace, but a cycle from each pattern is taken, interlaced
-- and squashed into a single cycle.
interlaceCycle :: [Pattern a] -> Pattern a
interlaceCycle pats = _fast (fromIntegral $ length pats) $ interlace pats
```

Let's look at a visualisation of the first six cycles of a pattern composed using both `interlace` and `stack`. In the following time is from left to right, with the vertical axis used to show simultaneously occuring events.

```
fig2 = interlace [atom pink, atom purple]
```

We can now combine stacks and interlacements:

```
fig3 = stack [atom red,
              interlace [atom pink, interlace [atom purple, atom orange]]
             ]
```



4 MANIPULATING TIME

A good thing about pure FRP is that it is possible to manipulate time by making a new function that simply adjusts the time query that is passed to another function. However our representation

has timespans in two places – the query and in the events that result. To facilitate the corrent manipulation of both, I define some utility functions for working with query and event time:

```
withSpanTime :: (Time -> Time) -> TimeSpan -> TimeSpan
withSpanTime timef (TimeSpan b e) = TimeSpan (timef b) (timef e)

withQueryTime :: (Time -> Time) -> Pattern a -> Pattern a
withQueryTime timef (Pattern q) = Pattern $ q . withSpanTime timef

withEventTime :: (Time -> Time) -> Pattern a -> Pattern a
withEventTime f = withEvent $ \e -> e {active = withSpanTime f $ active e,
                                       whole = withSpanTime f <$> whole e
                                      }
    where withEvent ef (Pattern q) = Pattern $ map ef <$> q

withTime :: (Time -> Time) -> (Time -> Time) -> Pattern a -> Pattern a
withTime fa fb pat = withEventTime fa $ withQueryTime fb pat
```

It is then straightforward to define functions for making patterned events to be faster or slower, or shifting them in time to be early or late. For example to make a pattern 'faster', query time is divided and event time is multiplied by a given factor. This queries a wider window, and 'squashes' the results back into the requested timespan.

```
_fast, _slow, _late, _early :: Time -> Pattern a -> Pattern a
_fast t  = withEventTime (/ t) . withQueryTime (* t)
_slow t  = withEventTime (* t) . withQueryTime (/ t)
_early t = withEventTime (subtract t) . withQueryTime (+ t)
_late t  = withEventTime (+ t) . withQueryTime (subtract t)
```

We can apply visualisation in understanding how events can become broken up. If `interlace` works cycle-by-cycle, what happens if an event lasts longer than a cycle?

```
fig4 = interlace [atom red, _slow 3 $ atom green]
```

The jagged edges indicate where wholes begin before, or end after, the active event timespans. From the above we can see that each green event has been broken into parts of one cycle each, and interlaced with the single-cycle red events.

4.1 Combining patterns with monadic binds

The above functions are prefixed by _, because they are considered internal functions and not part of the end-user interface. The reason for this is that TidalCycles follows the principle that *everything* is a pattern. Accordingly, we require functions with the following time signature, where the time factor is also patterned:

```
fast, slow, late, early :: Pattern Time -> Pattern a -> Pattern a
```

To implement these functions, we somehow need a way to compose patterns of time together with the patterns that are having their time structures manipulated. This is where Haskell's monadic bind (>>=) comes into view, which does what we want - it lifts functional arguments into contexts

such as patterns. Our problem is now clarified as one of how to define the `Pattern` type as an instance of Haskell's standard Monad typeclass. In particular, we need to define the bind operator `>>=` for patterns, with the type signature `Pattern a -> (a -> Pattern b) -> Pattern b`. So, what should this bind do?

Certainly, our bind will need to create a new pattern, which as we saw above, will be a function from timespans to events. This function will need to be composed of other pattern functions, in particular the 'outer' pattern given as the first argument, and 'inner' pattern resulting from the second argument. The big remaining question is, how do we deal with the event timespans? The two active timespans are straightforwardly combined, as the intersection. There is ambiguity, however, in how the two 'whole' timespans should be combined.

//: <> The events returned from those inner pattern queries are then collated and returned, with one caveat – there is ambiguity about what the resulting event's 'whole' timespan should be. It could from from the 'outer' or 'inner' pattern, or be the intersection of the two.

This ambiguity comes down to where the pattern *structure* should come from - should we preserve the structure of the outer pattern, the inner pattern, or a combination of the two? In the case of the `fast` function (and its `slow`, `late` and `early` friends), we can say that we will always want to preserve the structure of the inner patterns - the patterns of values which come second in the function arguments. This is because we only want to transform a value pattern using a time pattern, but not otherwise change the value pattern's structure. (We will see examples of functions that *do* change the structure of events later.)

The following shows how inner, outer and 'mix' binds can be implemented.

TODO - this doesn't work well with outer (?) continuous events

```
bindWith :: (Maybe TimeSpan -> Maybe TimeSpan -> Maybe TimeSpan) ->
               Pattern a -> (a -> Pattern b) -> Pattern b
bindWith chooseWhole bv f = Pattern $ concatMap match . query bv
  where match e = map (withWhole e) $ query (f $ value e) $ active e
        withWhole e e' = e' {whole = chooseWhole (whole e) (whole e')}

innerBind :: Pattern a -> (a -> Pattern b) -> Pattern b
innerBind = bindWith (flip const)

outerBind :: Pattern a -> (a -> Pattern b) -> Pattern b
outerBind = bindWith const

mixBind :: Pattern a -> (a -> Pattern b) -> Pattern b
mixBind = bindWith (liftA2 intersect)
```

In `mixBind`, the intersection of the event wholes is taken, therefore combining the time structures of the two patterns in what we term a 'mix' bind. Alternatively, `innerBind` uses the time structure of 'whole' timespans from the inner pattern, and `outerBind` from the outer pattern. The default bind in the `Monad` instance is set as a `mixBind`. I also define an `Applicative` instance based on this bind, with additional `<<*>` and `<*>>` operators based on outer and inner binds respectively:

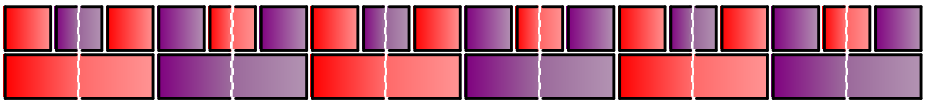
```
instance Monad Pattern where
  (>>=) = mixBind

instance Applicative Pattern where
  pure = atom
```

```
  pf <*> px = pf `mixBind` (<$> px)

-- (<<*>), (<*>>) :: Pattern (a -> b) -> Pattern a -> Pattern b
pf <<*> px = pf `outerBind` (<$> px)
pf <*>> px = pf `innerBind` (<$> px)
```

Using this, we can make a function `patternify_x` that lifts a function's first argument into a pattern, using `innerBind` to preserve the structure of the second argument. This can then be used to define our `fast`, `slow`, `late` and `early` functions.

```
patternify_x :: (a -> Pattern b -> Pattern c) ->
  (Pattern a -> Pattern b -> Pattern c)
patternify_x f ba bb = ba `innerBind` \a -> f a bb

fast, slow, late, early :: Pattern Time -> Pattern a -> Pattern a
fast = patternify_x _fast
slow = patternify_x _slow
late = patternify_x _late
early = patternify_x _early
```

```
fig5 = stack [fast (atom 1.5) p,
              slow (atom 2) p
             ]
    where p = interlace [atom red, atom purple]
```



From the above we can see that the resulting events get split at cycle boundaries, because the factor patterns (`atom 1.5` and `atom 2`) repeat every cycle. However more importantly the whole timespans are preserved, and so the overall time structure is maintained.[1]

We could avoid this fragmentation by using a continuous rather than a discrete value in our pattern of factors:

```
-- hold a value steady
hold :: a -> Pattern a
hold v = Pattern $ \ts -> [Event Nothing ts v]

fig5b = stack [fast (hold 1.5) p,
               slow (hold 2) p
              ]
    where p = interlace [atom red, atom purple]
```

[1]Note that events are filled with a gradient relative to the whole timespans, to visualise which active part of a whole timespan each event represents.

In practice we can consider the patterns to be equivalent – the overall structure is the same, whether or not contituent events have become fragmented by transformations.

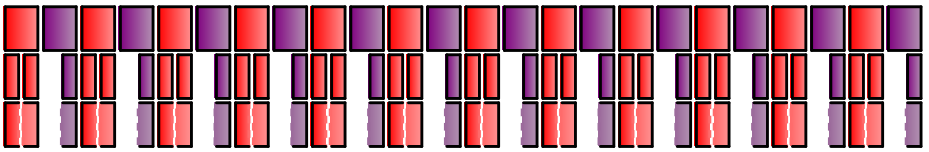
5 MASKING AND RESTRUCTURING PATTERNS

So far we have looked at the results of inner binds, where the structure of a transformed pattern is preserved. Lets now compare the use of inner and outer binds, and in particular their use in masking or restructuring pattern.

```
silence :: Pattern a
silence = Pattern $ const []

_ifpat :: Bool -> Pattern a -> Pattern a
_ifpat True p = p
_ifpat False _ = silence

mask :: Pattern Bool -> Pattern a -> Pattern a
mask bp p = bp `innerBind` \b -> _ifpat b p

struct :: Pattern Bool -> Pattern a -> Pattern a
struct bp p = bp `outerBind` \b -> _ifpat b p
```

The bind is the only difference between these two functions, but they have very different practical uses. `struct` is used to impose a new structure on a pattern, where as `mask` maintains the structure, but masks off some parts of it.

```
fig6 = stack [
    pat,
    struct (interlaceCycle [atom True, atom True, atom False, atom True]) pat,
    mask (interlaceCycle [atom True, atom True, atom False, atom True]) pat
  ]
  where pat = interlaceCycle [atom red, atom purple]
```



6 COMBINING CONTINUOUS AND DISCRETE EVENTS

A practical advantage of representing both continuous and discrete events in one type, is that they can be composed together. For example the following blends between two different discrete patterns, using a continuous sinewave:

```
fig7 = atom red `outerBind` \a
         -> atom blue `outerBind` \b
```

```
         -> _slow 12 sinewave `outerBind` \x
         -> return $ blend ((x+1)/2) a b
```

With the use of `outerBind`, the structure is taken from the first pattern, and so the events are discrete.

Unfortunately, our Pattern monad is not fully lawful, in that flipping the order of the patterns while also flipping from outer to inner bind, does not produce the same results as you might hope:

```
fig8 = _slow 12 sinewave `innerBind` \x
         -> atom blue `innerBind` \b
         -> atom red `innerBind` \a
         -> return $ blend ((x+1)/2) a b
```

The reason for this is that to produce the above visualisation, the resulting pattern is queried for twelve cycles at once. This is passed to the continuous pattern, which then produces a single event at that resolution. All is not lost, however, if we combine this resulting pattern with another discrete one, the correct result is shown again:

```
fig9 = struct (atom True) $
        _slow 12 sinewave `innerBind` \x
         -> atom blue `innerBind` \b
         -> atom red `innerBind` \a
         -> return $ blend ((x+1)/2) a b
```

The easiest workaround for this issue is to take care that continuous patterns are contained within discrete ones. In practice, this can be taken care of in the design of the combinator library, and is not something that the end-user programmer needs to worry about.

A PREAMBLE AND SUPPORTING FUNCTIONS

```
module Pattern where

import Control.Applicative
import Data.Fixed
import Data.Colour
import Data.Colour.Names

intersect :: TimeSpan -> TimeSpan -> TimeSpan
intersect (TimeSpan b e) (TimeSpan b' e') = TimeSpan (max b b') (min e e')
infixl 4 <<*>, <*>>
```

REFERENCES

[1] Elliott, C. and Hudak, P. 1997. Functional reactive animation. *Proceedings of the second ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, Aug. 1997), 263–273.