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, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 REPRESENTING PATTERNS

Functional Reactive Programming (FRP) was first implemented by Conal Elliot in Fran [Functional Reactive Animation; @cite], 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 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 (with some simplifications) as an endpoint. This is done in a practical, literate programming style, using the Haskell programming language. Example patterns are visualised, generated directly from the code in the paper. We will end up with a representation that is slightly simplified to but conceptually the same as that used in the TidalCycles programming language.

Our starting point is the following `Behaviour` type from Fran:

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

This principled start represents behaviours 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 the function returns a list of values.

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

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

1.1 Rational time

According to Elliot, `Time` in FRP should be *real*, therefore having arbitrary time 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 leave us with the problem of how to deal with floating point errors in time calculations.

To avoid floating point headaches, here I instead use rational time as a practical alternative. Unlike floating point numbers, this means ratios that are common in media arts can be properly represented, 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 music, metric cycles are referred to as measures or bars, and in Indian music as the nuanced structures of Tala cycles. In TidalCycles they are simply referred as cycles. The integer timeline (i.e., those ratios of 1) marks out the endings and beginnings of successive cycles.

I have also renamed the `Behaviour` type to `Pattern` in the above. This both differentiates our type from Elliot's, and supports later discussion relating this representation with the long history of pattern-making.

1.2 Timespans and events

Next, in order to support discrete events, I introduce the concept of events with time*spans* to our model. In the following, a timespan represents the time during which a given event is active.

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

To support this, I have also changed the `Pattern` type to be a function of timespans, rather than single time values. This is so the pattern can be queried with contiguous timespans, thereby avoiding any chance of missing events that might otherwise 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, particularly whether it started before and/or continues beyond that timespan. For this reason, I add another field to our Event datatype called `whole`, representing the whole timespan of the event, which will either be the same as or greater than the `active` part, but should always include it. We still 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.

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

Now the `Pattern` type can represent discrete events, but it would be best if it could still represent continuous values. Fundamentally, we define a continuous value as one which does not have a discrete beginning and end. So, we just 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`.

1.3 Complete pattern representation

The representation in types is now complete, with the following a basis for representing values that support both continuous and discrete time, within the same datatype.

```
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? For continuous patterns, we simply sample a value at the halfway point of the queried timespan. For example, a sinewave:

```
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)
```

For discrete patterns however, we need to supply both the whole and part. The following `atom` function returns a pattern that repeats an event with the given value every cycle.

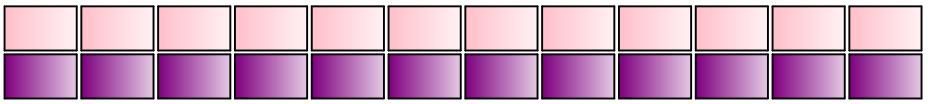
```
sam, nextSam :: Rational -> Rational
sam t = toRational (floor t :: Int)
nextSam = (1 +) . sam

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

We can compose a list of patterns into a new one, simply by passing queries on to each pattern. This assumes that we want the patterns to 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. Slightly more complicated is combining patterns in sequence, over time. Patterns have infinite length, so we are not able to simply concatenate them. Instead, we can 'interlace' their cycles.

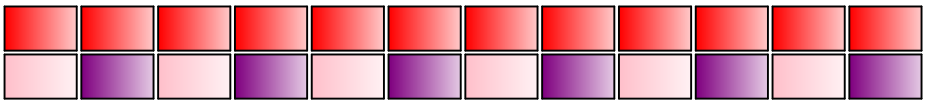
```
splitQueries :: Pattern a -> Pattern a
splitQueries pat =  Pattern $ concatMap (query pat) . splitSpan

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)
```

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", atom "purple"]]
```



4 MANIPULATING TIME

A good thing about pure FRP is that it is possible to manipulate time simply by making a new pattern function that simply adjusts the time query that is passed to an existing pattern function. However, because our representation has timespans in two places – in the query and the events that result – we must be careful to adjust both. We therefore require two functions for every time manipulation; one to adjust the query, and another to adjust the result, so that the event timespan are still active within the queried timespan. To facilitate this, I first define some utility functions for working with query and event time:

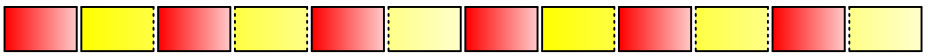
```
withSpanTime :: (Time -> Time) -> TimeSpan -> TimeSpan
withSpanTime timef (TimeSpan b e) = TimeSpan (timef b) (timef e)

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 faster/slower, or early/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 "yellow"]
```



The dashed lines indicate where wholes begin before, or end after, the active event timespans. From the above we can see that the three cycles of each yellow 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

On these foundations, a domain specific language can be built. 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:

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

How do we implement these functions? We somehow need a way to combine patterns of time 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. This clarifies our problem as one of how to define patterns 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 biggest question here is deciding how to deal with the timespans of events when composing pairs of patterns together. The two active timespans are straightforwardly combined, as the intersection. There is however ambiguity 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 above `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.

```
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.

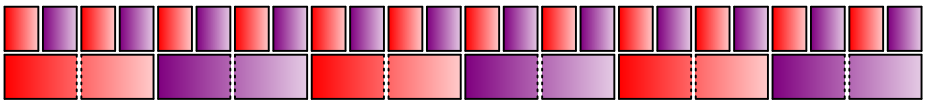
Alternatively, `innerBind` uses the time structure of 'whole' timespans from the inner pattern, and `outerBind` from the outer pattern. The default bind in the `Monad` instance is set as a `mixBind`. I also define an `Applicative` instance based on this bind.

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

instance Applicative Pattern where
  pure = atom
  pf <*> px = pf >>= (<$> 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 2) p,
              slow (atom 2) p
             ]
    where p = interlace [atom "red", atom "purple"]
```



From the above we can see that for the second pattern in the stack transformed with `slow`, because the `atom 2` repeats every cycle, the resulting events get split at cycle boundaries. However they keep their whole timespans with a duration of two cycles, and so the overall time structure is maintained. In the visualisations, events are filled with a gradient relative to the whole timespans, to make this a little clearer.

5 MASKING AND RESTRUCTURING PATTERNS

So far we have manipulated time, but not structure.

```
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 `outerBind` \b -> _ifpat b p

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

A PREAMBLE AND SUPPORTING FUNCTIONS

```
module Pattern where
```

```haskell
import Control.Applicative
import Data.Fixed

intersect :: TimeSpan -> TimeSpan -> TimeSpan
intersect (TimeSpan b e) (TimeSpan b' e') = TimeSpan (max b b') (min e e')
```