FRP for you and me

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. FRP for you and me. In . ACM, New York, NY, USA, **??** pages. https://doi.org/XXXXXXX.XXXXXXX

1 TIDALCYCLES: CONTINUOUS AND DISCRETE PATTERNS OF FUNCTIONAL REACTIVE PROGRAMMING

Functional Reactive Programming (FRP) was first introduced by Conal Elliot as Fran [Functional Reactive Animation; @cite], centering on a definition of behaviour as a continuous function of time. Popular implementations of FRP, such as in 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, that has been developed and popularised over the past ten years in the TidalCycles system, designed for creative, live exploration of musical and other patterns.

1.1 Representing patterns

In the following we willi step through the process of building a representation for musical pattern, taking Elliot's definition of behaviour in Fran as a worthy starting point. This is done in a practical, literate programming style, using the Haskell programming language. We will end up with a that is representation slightly simplified to but conceptually the same as that used in the TidalCycles programming language.

Our starting point is the following `Behaviour` type, based on Fran's behaviour type:

```
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

1.1.1 Rational time. According to Elliot, `Time` in FRP should be considered as *real*, therefore having arbitrary time precision, without building a notion of samplerate into the representation itself. In practice though, 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 we instead use rational time as a practical alternative. This fulfills a need that is very common in computational media: to accurately represent ratios, 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 your musical tradition. For example in Western music, metric cycles are referred to as measures or bars, and in Indian music the more nuanced Tala cycles. Taking inspiration from the latter, in (the eponymous) TidalCycles they are simply referred as cycles. The integer timeline (i.e., the ratios with denominator of 1) marks out the endings and beginnings of successive cycles.

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

1.1.2 Timespans and events. Next, in order to support discrete events, we 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}
```

To support this, we also change our `Pattern` type to be a function of time spans, rather than single time values. This is so we can query the pattern with contiguous time spans, thereby avoiding any chance of missing events that might otherwise fall between queries.

```
type Pattern a = TimeSpan -> [Event a]
```

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 what part of an event is active within a timespan, particularly whether it started before and/or continues beyond that timespan. For this reason, we add another field to our Event datatype called `whole`, representing the whole timespan of the event, which might be greater than the `active` part, but should always include it.

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

Now we turned the `Pattern` type into one that can represent discrete events, but it would be best if it could still represent continuous values. Here, the difference between a discrete and continuous value is that the latter is never part of a 'whole' event. So, we just need to make that optional.

```
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`.

1.2 Complete pattern representation

Our representation 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)
```

1.3 Constructing patterns

How does this work in practice? For continuous patterns, we simply sample a value at the halfway point of the given timespan. For example, a sinewave:

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

For discrete patterns, we need to supply both the whole and part. For example, to repeat a discrete value with a duration of one 'cycle', we can use the `atom` function:

```
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 v = Pattern $ map (\timespan -> Event (Just $ TimeSpan (sam $ begin timespan) (nextS
```

1.4 Manipulating time

One good thing about pure FRP is that it is possible to manipulate time, simply by making a new pattern function that calls the old one, adjusting time values passed to its query on the way. Because the above 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 timespans of events are still active within the queried timespan. To make that easy, here are 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
```

We can now make functions for making events faster/slower, or early/late.

```
_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)
```

For example in the above, to make a pattern 'faster', we both divide query time by a given factor, and multiply event time by that same given factor. In so doing we query a wider window for more events, that then 'squash' back into requested timespan.

1.5 Combining patterns

We can most combine a list of patterns concurrently, by simply applying the same query to them all:

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

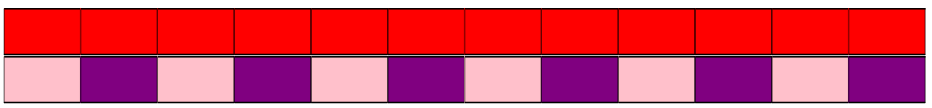
Slightly more complicated is combining them over time. Because patterns have infinite length, we are not able to concatenate them. Instead, we can 'interlace' cycles, which correspond to the integer timeline.

```
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)
```

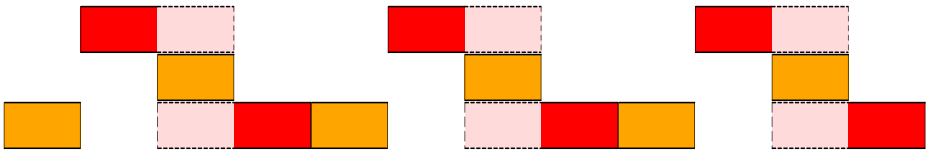
Let's look at a visualisation of the first six cycles of a pattern composed using both `interlace` and `stack`. In the following time is from left to right, with the vertical axis used to show simultaneously occuring events.

```
fig1 = stack [atom "red", interlace [atom "pink", atom "purple"]]
```



We can apply this 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?

```
fig2 = interlace [atom "orange", _slow 2 $ atom "red"]
```



From the above we can see that the 'whole' red events last two cycles, but each is broken in two 'active' parts lasting one cycle each. The red portion represents the active part, and the lighter red (pink) portion represents the rest of the 'whole'.

1.6 Combining patterns with monadic binds

On these foundations, we can begin building a domain specific language. 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 in TidalCycles, *everything* is a pattern. So we require functions with the following time signature:

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

This is where Haskell's monadic bind (>>=) comes into view, which supports the lifting of functional arguments into contexts like our patterns. In other words, defining our Pattern type as an instance of Haskell's standard Monad typeclass will solve this problem (and many others to come). However we do need to define the bind operator >>= for patterns, with the type signature `Pattern a -> (a -> Pattern b) -> Pattern b`. The question is, 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 patterns, in particular the 'outer' function given as the first argument, and 'inner' functions resulting from the second argument. The biggest question here is deciding how to deal with the timespans of events when composing these patterns together.

In the bind, the resulting pattern function will pass on its query to the 'outer' pattern function, then for each event returned by it, pass the event's value to the binding function, then use the event's active timespan to query the 'inner' pattern returned by the binding function. The events returned by the 'inner' pattern should then fall within the timespan of the outer events.

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. In practice, the question is about where pattern *structure* shoul 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 it's `slow`, `late` and `early` friends, we can say that we want to preserve the structure of the inner patterns - the patterns of values which come second in the functions' 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 -> Pa
bindWith chooseWhole bv f = Pattern $ concatMap match . query bv
  where match event = map (withWhole event) $ query (f $ value event) $ active event
        withWhole event event' = event' {whole = chooseWhole (whole event) (whole event')

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)
```

So by taking the intersection of the event wholes, we are combining the time structures of the two patterns in what we term a 'mix' bind. Alternatively we could do an 'inner' bind by using the time structure of the inner pattern, or an 'outer' bind by using the outer structure. The default bind in the `Monad` instance is set as a `mixBind`. We also define straightforward `Monoid` and `Applicative` instances.

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

instance Applicative Pattern where
  pure = atom
  pf <*> px = pf >>= (<$> px)
```

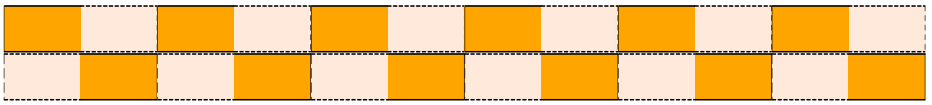
Using this, we can make a function `patternify_x` that lifts a function's first argument into a pattern, using `innerBind` to preserve the structure of the second argument. This can then be used to define our `fast`, `slow`, `late` and `early` functions.

```
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
```

```
fig3 = fast (atom 2) $ atom "orange"
```

```
fig4 = slow (atom 2) $ atom "orange"
```



1.6.1 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

import Control.Applicative
import Data.Fixed

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