morecycles

Quint Guvernator, Debdutta Guha Roy,
Kaustav Kumar Choudhury, Jay Warnock,
Gideon Pol, Milan La Rivière

Sunday 30th March, 2025

Abstract

This paper describes the design and implementation of six new musically relevant pattern manipulation functions for the music live-coding environment *TidalCycles*.

Contents

1 How to use this?

The source code of this project simultaneously defines a report as a LaTeX document; as well as a Haskell library with an included test suite. The latter can itself be used in two ways: as a Haskell library purely for pattern manipulation, or as a set of function definitions for the TidalCycles environment.

1.1 Compiling the report

To generate the PDF: clone the repository, open `report.tex` in your favorite LaTeX editor, and compile. You can also generate the report in a terminal by running `pdflatex report; bibtex report; pdflatex report; pdflatex report`

If you don't have a LaTeX environment installed locally, you can also upload the *entire* repository to a service like Overleaf, set the main file to `report.tex`, and compile it there.

1.2 Compiling and testing the library

You should have stack installed (see `https://haskellstack.org/`) and open a terminal in the same folder.

- To compile everything: `stack build`.
- To open ghci and play with the code: `stack ghci`
- To run the tests: `stack clean && stack test --coverage`

1.3 Using the functions in TidalCycles compositions

Getting the code to work in a live TidalCycles environment can be tricky. Doing it properly involves following the following steps, which are quite general because the particulars will depend on which platform you are using:

- Create a new stack project that pulls in both this library and TidalCycles as dependencies
- Install SuperCollider and SuperDirt locally
- Follow the rest of the installation instructions on the TidalCycles website, but use the local stack project instead of installing the package globally
- Create a `BootTidal.hs` file somewhere which imports the `morecycles` functions you want to use and binds them to the appropriate names
- Configure you reditor to read the `BootTidal.hs` file

However, experience in our group has shown that this can be a time-consuming and error-prone process on some platforms (particularly: the commercial operating systems). If the above steps are not working or take too long, we offer the following as a quick-and-dirty alternative:

- Install TidalCycles, SuperCollider, and SuperDirt exactly as described on the TidalCycles website
- Select the Haskell source of the function(s) you want to use
- Remove all newlines and comments
- Paste these into the editor you configured for TidalCycles in the first step
- Press the keyboard shortcut you typically use for executing a line to execute this code and add the name binding to the current interpreter session

2 About this project

TidalCycles[1] is an environment for music livecoding: a type of musical performance where an audience watches an artist write code which generates sound. TidalCycles is based on Haskell and offers a "mini-language" for users to define cyclic patterns of "events". Events, in turn, are time spans where a particular value is active. In a musical context, these values can be the names of samples, notes, or chords; but as there are no additional constraints put on event values, they can also represent other parameters that vary over the course of the musical performance, such as the cutoff frequency of a filter, a boolean mask selecting certain notes and ignoring others, etc.

In addition to the pattern mini-language, TidalCycles also provides a large library of functions which can modify patterns or apply digital signal processing (DSP) effects to the generated audio. In this project, each contributor was responsible for proposing, designing, implementing, and testing a pattern manipulation function: one which takes (at least) one pattern as input and produces a pattern as output. This structure allowed us to work more or less independently, exploring each of our musical interests, while still all learning how TidalCycles leverages the power of Haskell's inherant laziness to work with infinite patterns and constrained randomness.

3 Functions

3.1 Jumble (Quint)

In Western music, beats are often arranged hierarchically into a metrical structure. Depending on the rhythm of a pattern, some beats are more important to the pattern's structure than others which may just be present to add variety.

It would be nice if musicians could specify which beats are important, allowing the rest to vary automatically. We call this function `jumble`.

`jumble` takes two patterns as input: a content pattern ('Pattern a') and a mask pattern ('Pattern Bool'). This is similar to the signature of the 'mask' function.

The function produces as output a pattern where any event which overlaps with a 1 in the mask pattern is passed through undisturbed, and the values at the other events are rotated. The smallest "empty space" becomes the granularity at which the unmasked events are chopped.

[1] `https://tidalcycles.org/`

To implement this non-deterministic function, we first create a deterministic version where the permutation index is given explicitly as a parameter:

```
-- deterministic version of jumble, which takes a permutation index
jumble' :: Int -> Pattern Bool -> Pattern a -> Pattern a
jumble' 0 _ p = p
jumble' i mp p = stack [static, variable] where
  static = mask mp p
  variable = rotateValues $ mask (inv mp) p

  -- rotate a list (head to last) a certain number of times
  rot8 :: Int -> [a] -> [a]
  rot8 _ [] = []
  rot8 n' (x:xs) = if n == 0 then x:xs else rot8 (n-1) (xs ++ [x]) where
    n = n' `mod` length (x:xs)

  -- extract values from an event
  getValue :: Event a -> a
  getValue (Event _ _ _ v) = v

  -- rotate the values of an event
  rotateValues = cyclewise f where
    f events = inject (rot8 i values) events where
      values = getValue <$> events

  -- swap out the value in each event with values from a list
  inject :: [a] -> [Event b] -> [Event a]
  inject (v:vs) (e:es) = e{value=v} : inject vs es
  inject _ _ = []

  cyclewise f Pattern{query=oldQuery} = splitQueries Pattern{query=newQuery} where
    newQuery (State (Arc t0 t1) c) = (narrow . f) expandedEvents where
      expandedStart = fromInteger $ floor t0
      expandedEnd = fromInteger $ ceiling t1
      expandedArc = Arc expandedStart $ expandedEnd + (if expandedStart == expandedEnd then
          1 else 0)
      expandedState = State expandedArc c
      expandedEvents = sortOn (\Event{part=Arc{start=t}} -> t) (oldQuery expandedState)
      narrow es = [e{part=Arc (max t0 p0) (min t1 p1)} | e@Event{part=Arc p0 p1} <- es,
          isIn (Arc t0 t1) (wholeStart e)]
```

Using this definition, we can create a non-deterministic version by choosing a permutation index randomly:

```
-- non-deterministic version of jumble, which randomly chooses a new permutation each cycle
jumble :: Pattern Bool -> Pattern a -> Pattern a
jumble _mp _p = Pattern{query=newQuery} where
  newQuery _state = undefined -- choose [jumble' i mp p | i <- [0..length (query p state)
      -1]] -- TODO
```

Jumble tests

First, we need to describe how to create arbitrary `Pattern` instances:

```
instance (Arbitrary a) => Arbitrary (Pattern a) where
  arbitrary = sized m where
    m n | n < 4 = listToPat . (:[]) <$> arbitrary
    m n = fastCat <$> oneof [ sequence [resize (n `div` 2) arbitrary, resize (n `div` 2)
        arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary, resize (n `div` 2) arbitrary] ]

instance (Fractional a, Arbitrary a, Eq a) => Arbitrary (ArcF a) where
  arbitrary = sized m where
    m i = Arc 0 . notZero <$> x where
      x = resize (i `div` 2) arbitrary
      notZero n = if n == 0 then 1 else n
```

We can now define our tests:

```
main :: IO ()
main = hspec $ do
  describe "Jumble" $ do

    it "should change the pattern if nothing is masked" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[0]") (parseBP_E "[a b]")) (
          parseBP_E "[b a]" :: Pattern String)

    it "should change the pattern with a complex mask 1" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[1 0 1 0]") (parseBP_E "[a b c d]"
          )) (parseBP_E "[a d c b]" :: Pattern String)

    it "should change the pattern with a complex mask 2" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[1 0]") (parseBP_E "[a b c d]")) (
          parseBP_E "[a b d c]" :: Pattern String)

    it "should change the pattern with a complex mask 3" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[0 [1 0]]") (parseBP_E "[a b c d]"
          )) (parseBP_E "[b d c a]" :: Pattern String)

    it "should change the pattern with a complex mask 4" $
      property $ \a -> compareP a (jumble' 1 (parseBP_E "[1 0 1 0]") (parseBP_E "[bd [hh cp
          ] sd cp]")) (parseBP_E "[bd [cp cp] sd hh]" :: Pattern String)

    it "shouldn't change the pattern when the permutation index is zero" $
      property $ \a mp p -> compareP a (jumble' 0 mp p) (p :: Pattern Int)

    it "shouldn't change the pattern when the whole pattern is masked" $
      property $ \a i p -> compareP a (jumble' i (parseBP_E "[1]") p) (p :: Pattern Int)

    it "shouldn't change the pattern when the permutation index loops around" $
      property $ \a -> compareP a (jumble' 2 (listToPat [True, False, True, False]) (
          listToPat [1, 2, 3, 4])) (listToPat [1, 2, 3, 4 :: Int])
```

3.2 Gracenotes (Milan & Gideon)

In Western music, grace notes are quick, ornamental notes that precede a main note, adding expressiveness and variation. These notes do not affect the rhythm but add embellishment to a melody.

It would be nice if musicians could specify where grace notes should be added, allowing for dynamic and varied performances. We call this function `gracenotes`.

`gracenotes` takes two patterns as input: a content pattern (Pattern a) and a mask pattern (Pattern Bool). The function produces an output pattern where any event that overlaps with a 'True' in the mask pattern is duplicated with an additional grace note occurring just before it.

the function `gracenotes` takes as input a Time variable, a Pattern Bool, and an original Pattern. The output is the resulting pattern with grace notes added. The grace notes are added to the notes that match the mask pattern. The Double parameter specifies how early the grace note starts, relative to the main note. So a grace note with offset 0.125 starts 1/8th of a cycle before the main note, and ends when the main note starts. The pitch of the grace note is always the same as the next note in the original pattern. This, will wrap around to the next note in the pattern if the end of the pattern is reached.

```
gracenotes' :: Time -> Pattern Bool -> Pattern a -> Pattern a
gracenotes' offset mp p = stack [original, graceNotes] where
  original = p
  graceNotes = Pattern{query=newQuery}
  newQuery state =
```

```
    let
      -- Get events that match the mask
      maskedEvents = query (mask mp p) state
      -- Get the pattern for a single cycle
      referenceState = State {arc = Arc 0 1, controls = controls state}
      referenceCycle = query p referenceState
      -- Sort reference events by start time to establish sequence
      sortedReference = sortBy (\e1 e2 -> compare (cyclePos $ start $ part e1) (cyclePos $
          start $ part e2)) referenceCycle
      -- Create a circular list of values from the reference pattern
      referenceValues = cycle $ map value sortedReference
      -- For each masked event, find its position in the cycle and get the next value
      createGraceNote e =
        let
          -- Get normalized position in cycle (0-1)
          cyclePosTime = cyclePos $ start $ part e
          -- Find the closest event in the reference cycle
          findPosition [] _ = 0
          findPosition [_] _ = 0
          findPosition (x:xs) t =
            if abs (cyclePos (start (part x)) - t) < 0.0001
            then 0  -- Found the event
            else 1 + findPosition xs t
          eventIndex = findPosition sortedReference cyclePosTime
          -- Get next value, respecting the cycle structure
          nextValue = referenceValues !! (eventIndex + 1)
          -- Create the grace note
          t0 = start $ part e
          graceStart = t0 - offset
          graceEnd = t0
          gracePart = Arc graceStart graceEnd
          graceEvent = e {part = gracePart, whole=Just gracePart, value = nextValue}
        in
          graceEvent
    in
      map createGraceNote maskedEvents
```

For the non-deterministic version, we need to generate a random Boolean pattern. So we first define an instance of Arbitrary for Pattern.

To test this functionality manually, you can use the following commands; In the ghci terminal when purely working with patterns:

```
-- p2e $ gracenotes' 0.125 (s2p "[1 0 1 0]" :: Pattern Bool) (s2p "[a b c d]" :: Pattern
    String)
-- p2e $ gracenotes 0.125 (s2p "[a b c d]" :: Pattern String)
```

When working with tidal, producing sounds:

```
-- d1 $ gracenotes' 0.125 ("1 0 1 0" :: Pattern Bool) (n "c a f e" # sound "supermandolin")
```

Tests

First, we need to describe how to create arbitrary `Pattern` instances:

```
instance (Arbitrary a) => Arbitrary (Pattern a) where
  arbitrary = sized m where
    m n | n < 4 = listToPat . (:[]) <$> arbitrary
    m n = fastCat <$> oneof [ sequence [resize (n `div` 2) arbitrary, resize (n `div` 2)
        arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary, resize (n `div` 2) arbitrary] ]

instance (Fractional a, Arbitrary a, Eq a) => Arbitrary (ArcF a) where
  arbitrary = sized m where
```

```
    m i = Arc 0 . notZero <$> x where
      x = resize (i `div` 2) arbitrary
      notZero n = if n == 0 then 1 else n
```

We can now define our tests:

```
main :: IO ()
main = hspec $ do
  describe "GraceNotes" $ do
```

The first test checks that the function doesn't change the pattern when the mask is all zeros. When the mask is all zeros, the function should return the original pattern since no grace notes are added.

```
    it "shouldn't change the pattern when the mask is all zeros" $
      property $ \a -> compareP a
        (gracenotes' 0.125 (parseBP_E "[0 0 0 0]") (parseBP_E "[a b c d]"))
        (parseBP_E "[a b c d]" :: Pattern String)
```

The second test checks that the function doesn't make a difference what the gracenote length is when the mask is all zeroes. When the mask is all zeroes, the function should return the original pattern since no grace notes are added.

```
    it "shouldn't make a difference what the gracenote length is when the mask is all
        zeroes" $
      property $ \a -> compareP a
        (gracenotes' 0.25 (parseBP_E "[0 0 0 0]") (parseBP_E "[a b c d]" :: Pattern String)
            )
        (gracenotes' 0.5 (parseBP_E "[0 0 0 0]") (parseBP_E "[a b c d]"))
```

The third test checks that the function adds grace notes for all events when the mask is all ones.. When the mask is all ones, the function should add grace notes for all events in the pattern. This test tests them for a length of 1/8.

```
    it "should add grace notes for all events when the mask is all ones and the length is
        1/8" $
      property $ \a -> counterexample
        ("Actual (floating-point):\n" ++ printPattern a (gracenotes' 0.125 (parseBP_E "[1 1
            1 1]") (parseBP_E "[a b c d]") :: Pattern String) ++
        "\nExpected (floating-point):\n" ++ printPattern a correctPatternTest3)
        (compareP a
          (gracenotes' 0.125 (parseBP_E "[1 1 1 1]") (parseBP_E "[a b c d]"))
          correctPatternTest3)
```

The fourth test checks that the function adds grace notes for all events that overlap when the length is 1/4. This test tests them for a length of 1/4.

```
    it "should add grace notes for all events that overlap when the length is 1/4" $
      property $ \a -> counterexample
        ("Actual (floating-point):\n" ++ printPattern a (gracenotes' 0.25 (parseBP_E "[1 1
            1 1]") (parseBP_E "[a b c d]") :: Pattern String) ++
        "\nExpected (floating-point):\n" ++ printPattern a correctPatternTest4)
        (compareP a
          (gracenotes' 0.25 (parseBP_E "[1 1 1 1]") (parseBP_E "[a b c d]"))
          correctPatternTest4)
```

The fifth test checks the functionality of the mask. It creates random masks and tests if the grace notes are added correctly.

```
  it "should selectively add grace notes if the mask is not uniformly ones" $
    property $ \a m ->
    let
      mask = take 4 (m ++ repeat False) :: [Bool]
      maskPattern = listToPat mask :: Pattern Bool
    in counterexample
      ("Actual (floating-point):\n" ++ printPattern a (gracenotes' 0.25 (parseBP_E "[1 1
          1 1]") (parseBP_E "[a b c d]") :: Pattern String) ++
      "\nExpected (floating-point):\n" ++ printPattern a (correctPatternTest5 mask))
      (compareP a
        (gracenotes' 0.125 maskPattern (parseBP_E "[a b c d]"))
        (correctPatternTest5 mask))
```

A helper function to print the patterns:

```
printPattern :: (Show a) => Arc -> Pattern a -> String
printPattern arcRange pat = unlines $ map showEvent $ queryArc pat arcRange
  where
    showEvent (Event {part = Arc s e, value = v}) =
      "(" ++ show (realToFrac s :: Double) ++ ">" ++ show (realToFrac e :: Double) ++ ")|"
          ++ show v
```

The correct patterns are excluded from the report because they are too long.

3.3 Jitter (Debdutta)

In live-coded music, perfect quantization can sometimes sound mechanical and rigid. Human musicians naturally introduce slight variations in timing, creating a sense of groove, swing, or expressiveness. It would be useful if musicians could introduce controlled randomness into their patterns, allowing each event to slightly vary in timing while still maintaining the overall rhythmic structure. This function, which we call `jitter`, enables such organic fluctuations by introducing small, randomized shifts to event start times.

The function `myModifyTime` precisely modifies event timing by applying a transformation function to each event's start time. It queries all events within a time span, retrieves their start time, converts it to a Double for the transformation, applies the function, converts the modified time back to Rational, and updates the event's arc while preserving the original stop time.

This mechanism allows controlled alterations in rhythmic feel, making patterns more flexible and human-like. It forms the basis for higher-level functions like jitter effects, which introduce randomness or other time-based modifications into a performance.

```
myModifyTime :: Pattern a -> (Double -> Double) -> Pattern a
myModifyTime pat f = Pattern $ \timeSpan ->
  map updateEvent (query pat timeSpan)
  where
    -- updateEvent takes an event e and modifies its 'part' field.
    updateEvent e =
      let eventArc = part e          -- eventArc :: ArcF Time, where Time is Rational
          currentStart = start eventArc -- get the start time (a Rational)
          -- Convert the start time to Double, apply f, and convert back to Rational.
          newStart = toRational (f (realToFrac currentStart))
          -- Build a new arc with the updated start, keeping the original stop.
          newArc = eventArc { start = newStart }
      in e { part = newArc }
```

The `jitterWith` function introduces controlled timing variations by applying an offset function to the start time of each event in a pattern. It simultaneously queries the input pattern and

a continuously generated random pattern using the built-in `rand` function. For each event, it takes a random value from the random pattern, applies the offset function, and adds the result to the event's start time. This process creates systematic deviations from strict timing, enabling effects like swing, groove, or subtle rhythmic fluctuations while preserving the overall pattern structure.

Example: `d1 $ jitterWith (*0.05) (sound "bd sn cp hh")`. This example shifts each event's start time by 0.05 times a random value, adding controlled, random variation to the rhythm.

```
jitterWith :: (Double -> Double) -> Pattern a -> Pattern a
jitterWith offsetFunc pat = Pattern $ \timeSpan ->
  let events = query pat timeSpan
      randomOffsets = query (rand :: Pattern Double) timeSpan
      updateEvent (e, r) =
        let eventArc = part e
            currentStart = start eventArc
            timeOffset = offsetFunc r -- Apply the random value to the offset function
            newStart = toRational (realToFrac currentStart + timeOffset)
            newArc = eventArc { start = newStart }
        in e { part = newArc }
  in zipWith (curry updateEvent) events (map value randomOffsets)
```

The function `jitter` introduces natural-sounding randomness by applying a small, unpredictable time shift to the start time of each event in a pattern. It creates a more dynamic, human-like feel in rhythmic sequences by varying event timing within a controlled range. When the provided maximum jitter is zero, the function simply returns the original pattern unaltered. Otherwise, it leverages `jitterWith` along with a helper function (`randomOffset`) that generates a random value between $-$`maxJitter` and `maxJitter`. This random offset is added to each event's start time, ensuring that every execution produces slightly different timing variations.

```
jitter :: Pattern a -> Double -> Pattern a
jitter pat maxJitter
  | maxJitter == 0 = pat -- No jitter when max jitter is 0
  | otherwise = jitterWith randomOffset pat
  where
    -- Generate a random offset between -maxJitter and maxJitter
    randomOffset :: Double -> Double
    randomOffset _ = unsafePerformIO (randomRIO (-maxJitter, maxJitter))
```

The function `jitterP` introduces random timing variations to a pattern. It dynamically determines the maximum jitter applied to each event based on a separate pattern (`maxJitterPat`). For each event, it identifies the corresponding event in `maxJitterPat` based on overlapping time cycles and uses its value as the upper bound for a random offset. A random value is generated uniformly in the range $[-m, m]$, where m is the extracted jitter value, and added to the event's start time. This dynamic variation creates nuanced and human-like timing fluctuations while preserving the overall pattern structure.

```
jitterP :: Pattern a -> Pattern Double -> Pattern a
jitterP pat maxJitterPat = Pattern $ \timespan ->
    let contentEvents = query pat timespan
        maxEvents     = query maxJitterPat timespan
        getMaxForEvent e =
            let t = start (part e)  -- current event start time (Rational)
                matching = filter (\e' ->
                        let p = part e'
                            pStart = start p
                            pStop  = stop p
                        in t >= pStart && t < pStop)
                    maxEvents
            in case matching of
```

```
            (m:_) -> value m
            []    -> 0
    updateEvent e =
        let currentStart = start (part e)
            m   = getMaxForEvent e
            timeoffset       = unsafePerformIO (randomRIO (-m, m))
            newStart     = toRational (realToFrac currentStart + timeoffset)
            newArc       = (part e) { start = newStart }
        in e { part = newArc }
in map updateEvent contentEvents
```

Implementation

To test and see how `jitter` and `jitterP` work, one can follow these steps:

- **Load the Module:** In your TidalCycles session, ensure that your module is in the search path and load it by running: `:set -i"/morecycles/lib"`
  `:m + Jitter`
- **Test the Fixed Jitter Function:** Apply the `jitter` function to a pattern with a fixed maximum jitter value. For example: `d1 $ jitter (sound "bd sn cp hh") 0.02`
  This command will randomly shift the start time of each event by up to ± 0.02 cycles.
- **Test the Variable Jitter Function:** Apply the `jitterP` function to a pattern using a dynamic maximum jitter value derived from a pattern. For example: `d1 $ jitterP (sound "bd sn cp hh") (range 0.01 0.05 sine)`
  Here, the maximum jitter value varies between 0.01 and 0.05 cycles following a sine wave, so the random shift applied to each event is determined by the corresponding value in this pattern.
- **Observing the Effects:** Listen carefully to the output of each command. The first test should produce a consistent, fixed range of timing variations, while the second test will yield dynamically changing variations in timing, resulting in a more expressive and humanized rhythmic feel.

Deterministic vs. Non-Deterministic Functions

A function is deterministic if it always produces the same output for the same input. Non-deterministic functions may produce different outputs for the same input due to randomness, external state, or time-dependent behavior.

In our implementation, the behavior of the functions falls into two categories:

- **Deterministic Functions:** A deterministic function always produces the same output given the same input and transformation. In our code, the function `myModifyTime` is deterministic. It applies a user-supplied transformation function to the start time of every event in a pattern. For example, if you apply

  ```
  myModifyTime pat (+0.1)
  ```

every event's start time will be shifted exactly by 0.1 cycles every time. There is no external randomness, so the output pattern is predictable.

- **Non-Deterministic Functions:** Non-deterministic functions, on the other hand, can produce different outputs on each execution even with the same input. In our code, the functions `jitterWith`, `jitter`, and `jitterP` are non-deterministic because they incorporate randomness in their calculations.

  - `jitterWith` is non-deterministic when its offset function involves randomness. In our implementation, we use a function that retrieves random values (from the built-in `rand` pattern or via `unsafePerformIO (randomRIO ...)`), so each time the function is applied, different random offsets may be added to the event start times.
  - `jitter` uses `jitterWith` with a helper function that generates a random offset in the range $[-\texttt{maxJitter}, \texttt{maxJitter}]$ via `randomRIO`. For example, executing

    ```
    d1 $ jitter (sound "bd sn cp hh") 0.02
    ```

    will randomly shift each event's start time by a value between -0.02 and +0.02 cycles. Even if you run the same command repeatedly, the exact timing of the events will vary because of the random values.
  - `jitterP` further extends the concept by determining the maximum jitter for each event from a provided pattern (e.g., `(range 0.01 0.05 sine)`). Although the maximum allowed jitter for each event is derived from a deterministic pattern, the actual offset applied is still chosen randomly within the calculated bounds. Thus, executing

    ```
    d1 $ jitterP (sound "bd sn cp hh") (range 0.01 0.05 sine)
    ```

    yields event timings that change from cycle to cycle in a non-predictable manner.

In conclusion, while `myModifyTime` deterministically applies a specified transformation to event timings, the higher-level functions `jitterWith`, `jitter`, and `jitterP` incorporate randomness to produce non-deterministic, dynamic variations in timing. This non-determinism is essential for creating a more humanized, less mechanical rhythmic feel.

Jitter Test Cases

To verify the functionality of the `jitter` function, we will create test cases that ensure the jitter function doesnt completely disrupt the rhythm and that the overall structure of the pattern remains intact.

```
testPattern :: Pattern String
testPattern = fromList ["a", "b", "c"]
```

This test verifies that jitterWith, when given a constant offset function (e.g., always adding 0.1 cycles), properly increases each event's start time by 0.1.

```
debugEvent :: Show a => EventF (ArcF Time) a -> String
debugEvent e =
  "Event { start = " ++ show (start (part e)) ++
  ", stop = " ++ show (stop (part e)) ++
  ", value = " ++ show (value e) ++ " }"
```

```
debugEvents :: Show a => [EventF (ArcF Time) a] -> String
debugEvents events = "[" ++ unlines (map debugEvent events) ++ "]"

testJitterWith :: Spec
testJitterWith = describe "jitterWith" $ do
  it "shifts each event's start time within the expected range" $ do
    let maxJitter = 0.1
        jitteredPattern = jitterWith (const maxJitter) testPattern
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        differences = zipWith (\e1 e2 -> abs ((start . part) e2 - (start . part) e1))
                              originalEvents jitteredEvents
        maxJitterRational = toRational maxJitter
    differences `shouldSatisfy` all (<= maxJitterRational)
```

This test checks that applying the jitter function with a maximum jitter of 0 results in an unchanged pattern (i.e. no timing shifts occur).

```
testJitterZero :: Spec
testJitterZero = describe "jitter (fixed max jitter)" $ do
  it "leaves event timings unchanged when max jitter is 0" $ do
    let jitteredPattern = jitter testPattern 0
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        originalStarts = map (start . part) originalEvents
        jitteredStarts = map (start . part) jitteredEvents
        originalVals   = map value originalEvents
        jitteredVals   = map value jitteredEvents
    jitteredStarts `shouldBe` originalStarts
    jitteredVals   `shouldBe` originalVals
```

Tests for jitterP:

- When the maximum jitter pattern is pure 0, the output should equal the input.
- When a constant max jitter is provided (e.g., 0.1), each event's timing shift should be within the bound [-0.1, 0.1].

```
testJitterPNoJitter :: Spec
testJitterPNoJitter = describe "jitterP (zero max jitter pattern)" $ do
  it "does not change event timings when max jitter pattern is 0" $ do
    let maxJitterPat = pure 0 :: Pattern Double
        jitteredPattern = jitterP testPattern maxJitterPat
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        originalStarts = map (start . part) originalEvents
        jitteredStarts = map (start . part) jitteredEvents
    jitteredStarts `shouldBe` originalStarts

testJitterPWithinBounds :: Spec
testJitterPWithinBounds = describe "jitterP (constant max jitter pattern)" $ do
  it "ensures each event's timing shift is within the specified bound" $ do
    let maxJitter = 0.1
        maxJitterPat = pure maxJitter :: Pattern Double
        jitteredPattern = jitterP testPattern maxJitterPat
        state = State (Arc 0 3) 0
        originalEvents = query testPattern state
        jitteredEvents = query jitteredPattern state
        differences = zipWith (\e1 e2 -> abs ((start . part) e2 - (start . part) e1))
                              originalEvents jitteredEvents
        maxJitterRational = toRational maxJitter
    differences `shouldSatisfy` all (<= maxJitterRational)
```

The main function runs the test suite.

```
main :: IO ()
main = hspec $ do
  testJitterWith
  testJitterZero
  testJitterPNoJitter
  testJitterPWithinBounds
```

The following test cases verify the correctness of the jitter functions:

- **testJitterWith:** Ensures `jitterWith` shifts event start times by a fixed offset when provided with a constant function. Confirms deterministic behavior.
- **testJitterZero:** Verifies that applying `jitter` with a maximum jitter of 0 leaves event timings unchanged, ensuring correct handling of the boundary condition.
- **testJitterPNoJitter:** Confirms that when the jitter pattern is constant at 0 (`pure 0`), `jitterP` produces an output identical to the input pattern.
- **testJitterPWithinBounds:** Ensures that when a maximum jitter pattern (e.g., `pure 0.1`) is applied, event shifts remain within the specified bound $[-0.1, 0.1]$, verifying controlled randomness.

The main function runs all test cases using Hspec to validate deterministic and non-deterministic behavior.

3.4 Arrhythmia (Jay)

This function takes a `ControlPattern` and a `Time` r and returns another pattern that plays normally during its first its first cycle and then inserts a rest of length r at the beginning of the next cycle so that whatever events occur from the time $1 - r$ to 1 in the original cycle are delayed until the third cycle. This causes the originally patern to become progressively less in sync with its time signature and creates interesting polyrhythms when combined with other patterns with the same signature.

First we provide a function that queries a pattern during its first cycle to determine if splitting the pattern into n pieces (where n is the denominator of our time input) is possible. Otherwise, we have events that begin in one cycle and end in another which Tidal treats as two separate events and plays them twice.

```
chopper :: Integer -> Rational -> Integer -> [(Rational,Rational)]
chopper 0 _ _ = []
chopper j m k = (m,m+(1%k)) : chopper (j-1) (m+(1%k)) k

safe2Divide :: Pattern a -> Integer -> Bool
safe2Divide p k = and [and [fromJust (whole x) == eventPart x |x <- queryArc p $ uncurry
    Arc tpl] | tpl <- chopper k 0 k]
```

Now, a simple function that takes a pattern, some number of cycles j and some cycle index k and overlays j cycles of silence with the pattern but only on cycles divisible by k.

```
playOnly :: Int -> Int -> Pattern ValueMap -> Pattern ValueMap
playOnly j k p = every' (pure j) (pure k) (<> p) $ s $ parseBP_E "~"
```

This function queries a pattern at a specific time interval (an arc) and returns a list of tuples of times and patterns where the time is what portion of a cycle that pattern occupies, including the silences. This output can be fed into the native `timeCat` function to create a new pattern easily.

```
hlpr :: Time -> [Event ValueMap] -> [(Time,ControlPattern)]
hlpr r [] = [(1-r,s $ parseBP_E "~")]
hlpr r ((Event _ _ (Arc st ft) x) : evs) = [(st-r,s $ parseBP_E "~"),(ft-st,pure x)] ++
    hlpr ft evs

intervals :: ControlPattern -> Arc -> [(Time,ControlPattern)]
intervals pat ar = filter (\x -> fst x /= 0) $ hlpr (start ar) $ queryArc pat ar
```

Here, we have a function that, when given an integer n and the delay time t, returns the nth shifted cycle. For example, if shifting by one third of a cycle, the third cycle will begin with the last third of the original cycle, followed by one third of a cycle's rest, followed by the first third of the original cycle.

```
cycleCycle :: Integer -> Time -> ControlPattern -> ControlPattern
cycleCycle 0 _ pat = pat
cycleCycle j r pat = timeCat $ frontend ++ (r,s $ parseBP_E "~")
  : filter (\x -> fst x /= 0) (init backend
  ++ [((fst . last) backend - r*(j%1),(snd . last) backend)]) where
  frontend = intervals pat (Arc (1-r*((j%1)-1)) 1)
  backend  = intervals pat $ Arc 0 (1-(j%1)*r)
```

Finally, we put it all together with arrhythmia. The `<>` operator that we feed into `foldl` plays two patterns simulaneously. So we begin with the orignal pattern, and then we fold in a pattern that plays one of the fragmented sections only on the corresponding cycles. Again, this only requires `denominator r` steps before we loop around to the original pattern.

```
arrhythmia :: Time -> ControlPattern -> Either String ControlPattern
arrhythmia r pat
  | not $ safe2Divide pat (denominator r) = Left "pattern cannot be subdivided"
  | otherwise = Right $ foldl (<>) (playOnly count 0 pat) [playOnly count j (cycleCycle (
      toInteger j) r pat) | j <- take (count-1) [1..]] where
      count = fromIntegral (1 + denominator r)

arrhythmiaUnsafe :: Time -> ControlPattern -> ControlPattern
arrhythmiaUnsafe r pat = foldl (<>) (playOnly count 0 pat) [playOnly count j (cycleCycle (
    toInteger j) r pat) | j <- take (count-1) [1..]] where
      count = fromIntegral (1 + denominator r)

arcStart :: Arc -> Time
arcStart (Arc t _) = t

arcEnd :: Arc -> Time
arcEnd (Arc _ t) = t
```

Tests

```
instance (Arbitrary a) => Arbitrary (Pattern a) where
  arbitrary = sized m where
    m n | n < 4 = listToPat . (:[]) <$> arbitrary
    m n = fastCat <$> oneof [ sequence [resize (n `div` 2) arbitrary, resize (n `div` 2)
        arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary]
      , sequence [resize (n `div` 2) arbitrary, resize (n `div` 2) arbitrary, resize (n `
          div` 2) arbitrary, resize (n `div` 2) arbitrary] ]

instance (Fractional a, Arbitrary a, Eq a) => Arbitrary (ArcF a) where
```

```
  arbitrary = sized m where
    m i = Arc 0 . notZero <$> x where
      x = resize (i `div` 2) arbitrary
      notZero n = if n == 0 then 1 else n
```

Here, we define two tests for arrhythmia. The first checks that shift the time by 0 returns the same pattern, and the second tests if the pattern eventually repeats the same cycle.

```
main :: IO ()
main = hspec $ do
  describe "Arrhythmia" $ do

    it "should not change the pattern if the time shifts by 0" (
      queryArc (arrhythmiaUnsafe 0 $ s $ parseBP_E "bd bd hh") (Arc 0 1)
      ==
      queryArc (s $ parseBP_E "bd bd hh") (Arc 0 1)
      )

    it "should repeat a cycle every k cycles" $
      property $ \a nt ->
        queryArc (arrhythmiaUnsafe nt $ s $ parseBP_E "bd bd hh") a ==
        queryArc (arrhythmiaUnsafe nt $ s $ parseBP_E "bd bd hh") (newArc a nt)
        where
          newArc :: Arc -> Time -> Arc
          newArc a nu = Arc (arcStart a * (denominator nu % 1)) ((arcStart a * (denominator
               nu % 1)) + (arcEnd a - arcStart a))
```

3.5 RhythmMask (Kaustav)

"RhythMask": Probability-Based Masking

The idea behind RhythMask is to create a rhythmic effect where some beats are probabilistically dropped or kept each cycle, rather than being strictly fixed by a binary mask.

Theoretical implementation idea

Input:

- A content pattern (Pattern a): The original rhythmic sequence.
- A probability pattern (Pattern Double): A probability value (between 0.0 and 1.0) that determines the likelihood of each beat being played.

Output: A modified version of the input pattern with beats dropped probabilistically.

Steps of implementation:

- Extract Events: The function first queries the input pattern to get a list of its events.
- Apply Probability Filtering: Each event is evaluated against the corresponding probability value.
- Random Decision Making: A random number is generated for each event, and if it is below the probability threshold, the event passes on to the final output, otherwise it is removed.
- Reconstruction of the Pattern: The remaining beats are reconstructed into a new TidalCycles pattern.

The main function that handles this is `rhythmaskProb`. It applies a probabilistic mask to a series of events (musical notes) that is passed as a string to Tidal. A probabilistic mask is a list of values like [0.0, 0.29, 0.17, 1.0], which defines a list of probabilities (the highest value is 1 and the lowest is 0). Each element in this list corresponds to a single event from a string that is passed to the function such as: "bd sn arpy hh". If the value is 1, then the beat (event) is always kept, if it 0, then it is always dropped, and if it is between 0 and 1, then based on the number (the probabilistic value), the beat may or may not be kept in each cycle. This introduces a little anarchy into the infinite musical loop in Tidal, wherein a beat that is played in the first cycle may not be played in the second one, but may be played again in another subsequent cycle.

The steps and the function described above define one of the main functionalities of the RhythMask function. However, this is relatively easy to do in Tidal using `fastcat` and `fmap`. So, in order to expand on this, I added addtional features to the code that makes sense, while staying true to the core idea of RhythMask. These functions are `rhythmaskProbWith`, `rhythmask` and `rhythmaskWith`.

The probabilistic mask is defined by the function `probMaskPattern`. The `fmap` here applies the function ($< p$) to each random value in rand, resulting in True with probability p (since a random number between 0 and 1 is $< p$ with probability p). The `fastcat` method takes a list [Pattern a] and returns Pattern a. It takes a list of patterns and concatenates them in time, cycling through them faster than `cat`. It divides one cycle equally among the patterns, so if you pass 4 patterns, each one gets 1/4 of a cycle.

The `rhythmaskProb` is the main function that takes a probabilistic mask and applies it to a sequence of events in Tidal.

```
-- | Applies a probabilistic mask to a pattern
rhythmaskProb :: Pattern a -> [Double] -> Pattern a
rhythmaskProb pat probs = myFilterEvents pat (probMaskPattern probs)
```

A sample input for this function can be defined as follows:

```
--d1 $ rhythmaskProb (sound "hh arpy bd sn") [0.9, 0.2, 0.8, 0.1]
```

`rhythmaskProbWith` uses a probabilistic mask generated by the function `probMaskPattern` to apply a transformation to an event instead of dropping it. If the value in the probabilistic mask is 1, then the specified transformation is always applied, if it 0, then it is never applied, and if it is between 0 and 1, then based on the number (the probabilistic value), a transformation may or may not be applied in each cycle of the infinite music loop playing in Tidal.

```
-- | Applies a transformation to masked-out events using a probabilistic mask
rhythmaskProbWith :: Pattern a -> [Double] -> (Pattern a -> Pattern a) -> Pattern a
rhythmaskProbWith pat probs transform =
  let maskP    = probMaskPattern probs
      notMaskP = fmap not maskP
      kept         = myFilterEvents pat maskP
      transformed  = myFilterEvents (transform pat) notMaskP
  in stack [kept, transformed]
```

Sample input(The "gain" transformation in Tidal controls the amplitude of an event using a power function with a default value of 1. Values lower than 1 make the event (sound) quieter.):

```
--d1 $ rhythmaskProbWith (sound "bd sn bd sn") [0.6, 0.4, 0.7, 0.2] (# gain 0.3)
```

`parseMask` converts a binary mask string (e.g. "1 0 1 0") into a Pattern Bool. It splits the string into words, maps "1" to True and any other word to False, cycles the resulting list, and then limits it to one cycle using take.

`myFilterEvents` takes a pattern 'pat' and a boolean mask pattern 'maskPat' and produces a new pattern that only keeps events where the mask is True. (It extracts the events from both patterns at a given time, zips them, and keeps an event if its corresponding mask event (extracted via value) is True.)

`rhythmask` applies a mask (given as a string like "1 0 1 0") to a pattern. Only events corresponding to a True (or "1") in the mask will be kept.

```
rhythmask :: Pattern a -> String -> Pattern a
rhythmask pat maskStr = myFilterEvents pat (parseMask maskStr)
```

The `randomMaskString` function is used to generate a random list of 1s and 0s, however, it is also possible to manually define the list. A sample input:

```
--d1 $ rhythmask (sound (parseBP_E "hh arpy bd sn bd hh arpy sn hh bd")) (randomMaskString
    10)
--d1 $ rhythmask (sound (parseBP_E "hh arpy bd sn bd hh arpy sn hh bd")) [1 1 0 1 1 0 0 0 1
     0]
```

`rhythmaskWith` applies a transformation to the events that are masked out. It splits the pattern into two parts: Events that are kept when the mask is True and transformed when the mask is False. This is rather similar to the `rhythmaskProbWith` function, but it does not depend on a list of probabilities. It takes a binary mask instead, where 1 is True and 0 is False.

```
rhythmaskWith :: Pattern a -> String -> (Pattern a -> Pattern a) -> Pattern a
rhythmaskWith pat maskStr transform =
  let maskP    = parseMask maskStr
      notMaskP = fmap not maskP
      kept          = myFilterEvents pat maskP
      transformed   = myFilterEvents (transform pat) notMaskP
  in stack [kept, transformed]
```

A sample input for this function for both cases can be defined as follows :

```
--d1 $ rhythmaskWith (sound "hh arpy bd sn") "1 0 1 0" (# gain 0.5)
```

Tests

The test suite validates the behavior of RhythMask, testing the performance of not only the main event manipulation functions, but also the helper functions that are responsible for filtering events and generating masks (Boolean and probabilistic both). The tests are written using `hspec` and `QuickCheck`.

```
-- Helpers
stateFor :: Int -> State
stateFor numEvents = State (Arc 0 (fromIntegral numEvents)) 0
randomMaskStringIO :: Int -> IO String
randomMaskStringIO n = unwords <$> replicateM n (randomRIO (0 :: Int, 1) >>= \b -> return (
    show b))
-- Generator: list of length n with alternating True/False
genAlternatingMask :: Int -> Gen [Bool]
genAlternatingMask n = return $ take n (cycle [True, False])
-- Generator: list of length n with alternating 1.0 and 0.0
genAlternatingProbs :: Int -> Gen [Double]
genAlternatingProbs n = return $ take n (cycle [1.0, 0.0])
```

`parseMask` validates that `parseMask` correctly converts a string like "1 0 1 0" into a Pattern Bool with the correct True/False values for one cycle.

```
-- Test: parseMask (static)
testParseMask :: Spec
```

```
testParseMask = describe "parseMask" $
  it "parses '1 0 1 0' into [True, False, True, False]" $ do
    let maskPat = parseMask "1 0 1 0"
        events  = take 4 (query maskPat (stateFor 4))
        values  = map value events
    values `shouldBe` [True, False, True, False]
```

`myFilterEvents` checks that the `myFilterEvents` function filters a pattern correctly based on a boolean mask. Uses `QuickCheck` to test many input patterns and masks.

```
-- Test: myFilterEvents (QuickCheck with NonEmpty)
prop_myFilterEvents :: NonEmptyList String -> Property
prop_myFilterEvents (NonEmpty xs) = forAll (genAlternatingMask (length xs)) $ \mask ->
  let pat      = fromList xs
      maskPat  = fromList mask
      filtered = myFilterEvents pat maskPat
      result   = map value (query filtered (stateFor (length xs)))
      expected = map fst . filter snd $ zip xs mask
  in result === expected
testMyFilterEvents :: Spec
testMyFilterEvents = describe "myFilterEvents" $
  it "filters events based on boolean mask" $
    property prop_myFilterEvents
```

`testRhythmask` tests `rhythmask` which applies a binary(1/0) string based mask. The test ensures that only "1"-marked events are retained.

```
-- Test: rhythmask (QuickCheck)
prop_rhythmask :: NonEmptyList String -> Property
prop_rhythmask (NonEmpty xs) = forAll (genAlternatingMask (length xs)) $ \mask ->
  let maskStr = unwords (map (\b -> if b then "1" else "0") mask)
      pat     = fromList xs
      result  = map value (query (rhythmask pat maskStr) (stateFor (length xs)))
      expected = map fst . filter snd $ zip xs mask
  in result === expected
testRhythmask :: Spec
testRhythmask = describe "rhythmask" $
  it "keeps only events where mask is '1'" $
    property prop_rhythmask
```

`testRhythmaskWith` extends `rhythmask` by testing `rhythmaskWith`, which also applies a transformation (like `crush` or `gain`) to events corresponding to 1 (True). For testing purposes, we append a "!" symbol instead of musical transformations, since the testing happens completely in Haskell.

```
-- Test: rhythmaskWith (QuickCheck)
prop_rhythmaskWith :: NonEmptyList String -> Property
prop_rhythmaskWith (NonEmpty xs) = forAll (genAlternatingMask (length xs)) $ \mask ->
  let maskStr   = unwords (map (\b -> if b then "1" else "0") mask)
      pat       = fromList xs
      transform = fmap (++ "!")
      result    = map value (query (rhythmaskWith pat maskStr transform) (stateFor (length
          xs)))
      kept      = map fst . filter snd $ zip xs mask
      dropped   = map ((++ "!") . fst) . filter (not . snd) $ zip xs mask
  in result === (kept ++ dropped)
testRhythmaskWith :: Spec
testRhythmaskWith = describe "rhythmaskWith" $
  it "keeps some events and transforms the rest" $
    property prop_rhythmaskWith
```

`testProbMaskPattern` validates that `probMaskPattern` produces correct boolean patterns. If the probabilities are all 0.0, no events should pass. If the probabilities are all 1.0, all events should pass.

```
-- Test: probMaskPattern (QuickCheck)
prop_probMaskPattern_allFalse :: Positive Int -> Bool
```

```
prop_probMaskPattern_allFalse (Test.QuickCheck.Positive n) =
  let pat = probMaskPattern (replicate n 0.0)
  in all (== False) (map value $ take n $ query pat (stateFor n))
prop_probMaskPattern_allTrue :: Positive Int -> Bool
prop_probMaskPattern_allTrue (Test.QuickCheck.Positive n) =
  let pat = probMaskPattern (replicate n 1.0)
  in all (== True) (map value $ take n $ query pat (stateFor n))
testProbMaskPattern :: Spec
testProbMaskPattern = describe "probMaskPattern" $ do
  it "produces all False when probability is 0" $ property prop_probMaskPattern_allFalse
  it "produces all True when probability is 1" $ property prop_probMaskPattern_allTrue
```

`testRhythmaskProb` tests `rhythmaskProb`, which uses a probability list (like [0.81, 0.25, 1.0, 0.0]) to probabilistically include or exclude events. This test verifies that events corresponding to 0.0 probability values get filtered out.

```
-- Test: rhythmaskProb (QuickCheck)
prop_rhythmaskProb_allFiltered :: NonEmptyList String -> Bool
prop_rhythmaskProb_allFiltered (NonEmpty xs) =
  let probs = replicate (length xs) 0.0
      pat   = fromList xs
      out   = rhythmaskProb pat probs
  in null (query out (stateFor (length xs)))
testRhythmaskProb :: Spec
testRhythmaskProb = describe "rhythmaskProb" $
  it "filters out all events when probability is 0" $ property
      prop_rhythmaskProb_allFiltered
```

`testRhythmaskProbWith` tests `rhythmaskProbWith`, similar to `rhythmaskWith` but for probabilistic masks. It makes sure that events with prob = 1.0 are kept and events with prob = 0.0 are transformed.

```
-- Test: rhythmaskProbWith (QuickCheck)
prop_rhythmaskProbWith :: NonEmptyList String -> Property
prop_rhythmaskProbWith (NonEmpty xs) = forAll (genAlternatingProbs (length xs)) $ \probs ->
  let pat       = fromList xs
      transform = fmap (++ "!")
      result    = map value (query (rhythmaskProbWith pat probs transform) (stateFor (
          length xs)))
      kept      = map fst . filter ((== 1.0) . snd) $ zip xs probs
      dropped   = map ((++ "!") . fst) . filter ((== 0.0) . snd) $ zip xs probs
  in result === (kept ++ dropped)
testRhythmaskProbWith :: Spec
testRhythmaskProbWith = describe "rhythmaskProbWith" $
  it "keeps events with prob=1.0 and transforms others" $
    property prop_rhythmaskProbWith
```

`testRandomMaskString` checks that `randomMaskStringIO` generates a string of exactly n binary digits (1/0), useful for random but deterministic masking in a real time scenario.

```
-- Test: randomMaskString (QuickCheck IO)
prop_randomMaskString_valid :: Positive Int -> Property
prop_randomMaskString_valid (Test.QuickCheck.Positive n) = ioProperty $ do
  s <- randomMaskStringIO n
  let bits = words s
  return $ length bits == n && all (`elem` ["0", "1"]) bits
testRandomMaskString :: Spec
testRandomMaskString = describe "randomMaskString" $
  it "generates a mask string with the specified number of bits" $
    property prop_randomMaskString_valid
```

3.6 SwingTime (Milan & Gideon)

In Western music, particularly in jazz and blues, swing refers to a rhythmic feel where alternate beats are slightly delayed, creating a "long-short" pattern instead of an evenly spaced beat structure. This gives the rhythm a more dynamic, groovy feel.

The `swing` function allows musicians to apply this effect by defining which beats should be shifted and by how much. This is controlled using a mask pattern, which determines which beats remain steady and which are swung.

`swing` takes a rational number, a mask pattern and a pattern to apply the swing to as arguments. The rational number is the swing amount, which is the amount of time to shift the beat. The mask pattern is a pattern of booleans that determines which beats should be shifted. The pattern to apply the swing to is the pattern that will be changed.

```
-- deterministic swing function with a given swing amount
swing :: Rational -> Pattern Bool -> Pattern a -> Pattern a
swing amt mp p = p {query = \st -> concatMap (applySwing mp st) (query p st)}
  where
    applySwing :: Pattern Bool -> State -> Event a -> [Event a]
    applySwing maskPattern st ev =
      case whole ev of
        Nothing -> [ev]  -- If there's no whole, return the event unchanged
        Just a ->
          let startTime = start a
              endTime = stop a
              swingAtSt = query maskPattern (st {arc = Arc startTime endTime})
              shouldSwing = not (null swingAtSt) && any (isTrue . value) swingAtSt
              isTrue True = True
              isTrue _ = False
              swingShift = if shouldSwing then amt else 0
              newArc = Arc (startTime + swingShift) (endTime + swingShift)
          in [ev {part = newArc, whole = Just newArc}]
```

To test swing manually, the following commands can be used:

```
-- p2e $ swing 0.125 (s2p "[1 0 1 0]" :: Pattern Bool) (s2p "[a b c d]" :: Pattern String)
```

When working with tidal, producing sounds:

```
-- d1 $ swing 0.125 ("1 0 1 0" :: Pattern Bool) (n "c a f e" # sound "supermandolin")
```

4 Conclusion

TODO