Tidal: Domain specific language for live coding of pattern

Homepage and mailing list: http://yaxu.org/tidal/

Tidal is a language for live coding pattern, embedded in the Haskell language. You don't really have to learn Haskell to use Tidal, but it might help to pick up an introduction. You could try Graham Hutton's "Programming in Haskell" or Miran Lipovača's "Learn you a Haskell for Great Good" (which has a free online version). Or, you could just try learning enough syntax just by playing around with Tidal.

Installation

Linux installation: https://github.com/yaxu/Tidal/blob/master/doc/install-linux.md

Mac OS X installation: https://github.com/yaxu/Tidal/blob/master/doc/install-osx.md

Feel free to ask questions and share problems and success stories on the mailing list.

Tidal is currently only usable within the emacs editor, as installed via the above instructions. Emacs is a long-lived and rather complex beast. If you're new to emacs, you can bring up a tutorial by pressing `ctrl-h`, and then `t`.

Sequences

Tidal starts with nine connections to the dirt synthesiser, named from `d1` to `d9`. Here's a minimal example, that plays a bass drum every loop:

```
d1 $ sound "bd"
```

In the above, `sound` tells us we're making a pattern of sounds, and `"bd"` is a pattern that contains a single sound. `bd` is a sample of a bass drum. To run the code, use `Ctrl-C` then `Ctrl-C`.

In case you're wondering, the $ character in the above is Haskell syntax, which just means "give the result of the right to the function on the left". An alternative would have been to do without the $ by wrapping everything on the right in parenthesis: `d1 (sound "bd")`

We can pick variations of a sound by adding a colon (`:`) then a number, for example this picks the fourth bass drum (it counts from 0, so :3 gives you the fourth sound):

```
d1 $ sound "bd:3"
```

Putting things in quotes actually defines a sequence. For example, the following gives you a pattern of bass drum then snare:

```
d1 $ sound "bd sn"
```

When you do `Ctrl-C Ctrl-C` on the above, you are replacing the previous pattern with another one on-the-fly. Congratulations, you're live coding.

The `sound` function in the above is just one possible parameter that we can send to the synth. Below show a couple more, `pan` and `vowel`:

```
d1 $ sound "bd sn sn"
   |+| vowel "a o e"
   |+| pan "0 0.5 1"
```

NOTE: `Ctrl-C Ctrl-C` won't work on the above, because it goes over more than one line. Instead, do `Ctrl-C Ctrl-E` to run the whole block. However, note that there must be blank lines surrounding the block (which also means that patterns cannot contain blank lines).

Note that for `pan`, when working in stereo, that `0` means hard left, `1` means hard right, and `0.5` means centre.

When specifying a sequence you can group together several events to play inside a single event by using square brackets:

```
d1 $ sound "[bd sn sn] sn"
```

This is good for creating compound time signatures (sn = snare, cp = clap):

```
d1 $ sound "[bd sn sn] [cp cp]"
```

And you put events inside events to create any level of detail:

```
d1 $ sound "[bd bd] [bd [sn [sn sn] sn] sn]"
```

You can also layer up several loops, by using commas to separate the different parts:

```
d1 $ sound "[bd bd bd, sn cp sn cp]"
```

This would play the sequence `bd bd bd` at the same time as `sn cp sn cp`. Note that the first sequence only has three events, and the second one has four. Because tidal ensures both loops fit inside same duration, you end up with a polyrhythm.

Try replacing the square brackets with curly brackets:

```
d1 $ sound "{bd ht lt, sn cp}"
```

This is a different way of specifying a polyrhythm. Instead of both parts taking up the same amount of time, each event within the second part takes up the same amount of time as the second part. You can embed these different forms inside each other:

```
d1 $ sound "{bd [ht sn, lt mt ht] lt, sn cp}"
```

You can make parts of patterns repeat by using ∗, for example the following expressions produce the same pattern:

```
d1 $ sound "[bd bd bd, sn cp sn cp]"

d1 $ sound "[bd*3, [sn cp]*2]"
```

Conversely, you can slow down patterns by using /, the following pattern plays part of each subpattern each cycle:

```
d1 $ sound "[bd sn sn*3]/2 [bd sn*3 bd*4]/3"
```

Peace and quiet with silence and hush

An empty pattern is defined as `silence`, so if you want to 'switch off' a pattern, you can just set it to that:

```
d1 silence
```

If you want to set all the connections (from `d1` to `d9`) to silence at once, there's a single-word shortcut for that:

```
hush
```

You can also isolate a single connection and silence all others with the `solo` function:

```
solo $ d1 $ sound "bd sn"
```

Beats per second

You can change the beats per second (bps) like this:

```
bps 1
```

If you prefer to think in beats per minute, simply divide by 60

```
bps (140 / 60)
```

Samples

All the samples can be found in the `samples` subfolder of the Dirt distribution. Here's some you could try:

```
flick sid can metal future gabba sn mouth co gretsch mt arp h cp
cr newnotes bass crow hc tabla bass0 hh bass1 bass2 oc bass3 ho
odx diphone2 house off ht tink perc bd industrial pluck trump
printshort jazz voodoo birds3 procshort blip drum jvbass psr
wobble drumtraks koy rave bottle kurt latibro rm sax lighter lt
```

Each one is a folder containing one or more wav files. For example when you put `bd:1` in a sequence, you're picking up the second wav file in the `bd` folder. If you ask for the ninth sample and there are only seven in the folder, it'll wrap around and play the second one.

If you want to add your own samples, just create a new folder in the samples director, and put `wav` files in it.

Continuous patterns

As well as making patterns as sequences, we can also use continuous patterns. This makes particular sense for parameters such as `pan` (for panning sounds between speakers) and `shape` (for adding distortion) which are patterns of numbers.

```
d1 $ sound "[bd bd] [bd [sn [sn sn] sn] sn]"
   |+| pan sinewave1
   |+| shape sinewave1
```

The above uses the pattern `sinewave1` to continuously pan between the left and right speaker. You could also try out `triwave1` and `squarewave1`. The functions `sinewave`, `triwave` and `squarewave` also exist, but they go between `-1` and `1`, which is often not what you want.

Transforming patterns

Tidal comes into its own when you start building things up with functions which transform the patterns in various ways.

For example, `rev` reverses a pattern:

```
d1 $ rev (sound "[bd bd] [bd [sn [sn sn] sn] sn]")
```

That's not so exciting, but things get more interesting when this is used in combination another function. For example `every` takes two parameters, a number, a function and a pattern to apply the function to. The number specifies how often the function is applied to the pattern. For example, the following reverses the pattern every fourth repetition:

```
d1 $ every 4 (rev) (sound "bd*2 [bd [sn sn*2 sn] sn]")
```

You can also slow down or speed up the playback of a pattern, this makes it a quarter of the speed:

```
d1 $ slow 4 $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

And this four times the speed:

```
d1 $ density 4 $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

Note that `slow 0.25` would do exactly the same as `density 4`.

Again, this can be applied selectively:

```
d1 $ every 4 (density 4) $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

Note the use of parenthesis around `(density 4)`, this is needed, to group together the function `density` with its parameter `4`, before being passed as a parameter to the function `every`.

Instead of putting transformations up front, separated by the pattern by the `$` symbol, you can put them inside the pattern, for example:

```
d1 $ sound (every 4 (density 4) "bd*2 [bd [sn sn*2 sn] sn]")
   |+| pan sinewave1
```

In the above example the transformation is applied inside the `sound` parameter to d1, and therefore has no effect on the `pan` parameter. Again, parenthesis is required to both group together `(density 4)` before passing as a parameter to `every`, and also around `every` and its parameters before passing to its function `sound`.

```
d1 $ sound (every 4 (density 4) "bd*2 [bd [sn sn*2 sn] sn]")
   |+| pan (slow 16 sinewave1)
```

In the above, the sinewave pan has been slowed down, so that the transition between speakers happens over 16 loops.

Mapping over patterns

Sometimes you want to transform all the events inside a pattern, and not the time structure of the pattern itself. For example, if you wanted to pass a sinewave to `shape`, but wanted the sinewave to go from `0` to `0.5` rather than from `0` to `1`, you could do this:

```
d1 $ sound "bd*2 [bd [sn sn*2 sn] sn]"))
   |+| shape ((/ 2) <$> sinewave1)
```

The above applies the function `(/ 2)` (which simply means divide by two), to all the values inside the `sinewave1` pattern.

Parameters

These are the synthesis parameters you can use

- `sound` - a pattern of strings representing sound sample names (required)
- `pan` - a pattern of numbers between 0 and 1, from left to right (assuming stereo)
- `shape` - wave shaping distortion, a pattern of numbers from 0 for no distortion up to 1 for loads of distortion

- `vowel` - formant filter to make things sound like vowels, a pattern of either `a`, `e`, `i`, `o` or `u`. Use a rest (~) for no effect.
- `cutoff` - a pattern of numbers from 0 to 1
- `resonance` - a pattern of numbers from 0 to 1
- `speed` - a pattern of numbers from 0 to 1, which changes the speed of sample playback, i.e. a cheap way of changing pitch
- `accelerate` - a pattern of numbers that speed up (or slow down) samples while they play.
- `end` - a pattern of numbers from 0 to 1. Truncates samples before they finish playing.
- `delay` - a pattern of numbers from 0 to 1. Sets the level of the delay signal.
- `delayfeedback` - a pattern of numbers from 0 to 1. Sets the amount of delay feedback.
- `delaytime` - a pattern of numbers from 0 to 1. Sets the length of the delay.

Pattern transformers

In the following, functions are shown with their Haskell type and a short description of how they work.

brak

```
brak :: Pattern a → Pattern a
```

(The above means that `brak` is a function from patterns of any type, to a pattern of the same type.)

Make a pattern sound a bit like a breakbeat

Example:

```
d1 $ sound (brak "bd sn kurt")
```

Reversal

```
rev :: Pattern a → Pattern a
```

Reverse a pattern

Examples:

```
d1 $ every 3 (rev) $ sound (density 2 "bd sn kurt")
```

Beat rotation

```
(<~) :: Time → Pattern a → Pattern a
```

or

```
(~>) :: Time → Pattern a → Pattern a
```

(The above means that <~ and ~> are functions that are given a time value and a pattern of any type, and returns a pattern of the same type.)

Rotate a loop either to the left or the right.

Example:

```
d1 $ every 4 (0.25 <~) $ sound (density 2 "bd sn kurt")
```

Increase or decrease density

```
density :: Time → Pattern a → Pattern a
```

or

```
slow :: Time → Pattern a → Pattern a
```

Speed up or slow down a pattern.

Example:

```
d1 $ sound (density 2 "bd sn kurt")
   |+| slow 3 (vowel "a e o")
```

Every nth repetition, do this

```
every :: Int → (Pattern a → Pattern a) → Pattern a → Pattern a
```

(The above means `every` is a function that is given an integer number, a function which transforms a pattern, and an actual pattern, and returns a pattern of the same type.)

Transform the given pattern using the given function, but only every given number of repetitions.

Example:

```
d1 $ sound (every 3 (density 2) "bd sn kurt")
```

```
whenmod :: Int → Int → (Pattern a → Pattern a) → Pattern a → Pattern a
```

(The above has a similar form to `every`, but requires an additional number.)

Similar to `every`, but applies the function to the pattern, when the remainder of the current loop number divided by the first parameter, is less than the second parameter.

For example the following makes every other block of four loops twice as dense:

```
d1 $ whenmod 8 4 (density 2) (sound "bd sn kurt")
```

Interlace

```
interlace :: OscPattern → OscPattern → OscPattern
```

(A function that takes two OscPatterns, and blends them together into a new OscPattern. An OscPattern is basically a pattern of messages to a synthesiser.)

Shifts between the two given patterns, using distortion.

Example:

```
d1 $ interlace (sound  "bd sn kurt") (every 3 rev $ sound  "bd sn:2")
```

Spread

```
spread :: (a → t → Pattern b) → [a] → t → Pattern b
```

(The above is difficult to describe, if you don't understand Haskell, just read the description and examples..)

The `spread` function allows you to take a pattern transformation which takes a parameter, such as `slow`, and provide several parameters which are switched between. In other words it 'spreads' a function across several values.

Taking a simple high hat loop as an example:

```
d1 $ sound "ho ho:2 ho:3 hc"
```

We can slow it down by different amounts, such as by a half:

```
  d1 $ slow 2 $ sound "ho ho:2 ho:3 hc"
```

Or by four thirds (i.e. speeding it up by a third; `4%3` means four over three):

```
  d1 $ slow (4%3) $ sound "ho ho:2 ho:3 hc"
```

But if we use `spread`, we can make a pattern which alternates between the two speeds:

```
d1 $ spread slow [2,4%3] $ sound "ho ho:2 ho:3 hc"
```

There's a version of this function, `spread'` (pronounced "spread prime"), which takes a *pattern* of parameters, instead of a list:

```
d1 $ spread' slow "2 4%3" $ sound "ho ho:2 ho:3 hc"
```

This is quite a messy area of Tidal - due to a slight difference of implementation this sounds completely different! One advantage of using `spread'` though is that you can provide polyphonic parameters, e.g.:

```
d1 $ spread' slow "[2 4%3, 3]" $ sound "ho ho:2 ho:3 hc"
```

Striate

```
striate :: Int → OscPattern → OscPattern
```

Striate is a kind of granulator, for example:

```
d1 $ striate 3 $ sound "ho ho:2 ho:3 hc"
```

This plays the loop the given number of times, but triggering progressive portions of each sample. So in this case it plays the loop three times, the first time playing the first third of each sample, then the second time playing the second third of each sample, etc.. With the highhat samples in the above example it sounds a bit like reverb, but it isn't really.

You can also use striate with very long samples, to cut it into short chunks and pattern those chunks. This is where things get towards granular synthesis. The following cuts a sample into 128 parts, plays it over 8 cycles and manipulates those parts by reversing and rotating the loops.

```
d1 $  slow 8 $ striate 128 $ sound "bev"
```

The `striate'` function is a variant of `striate` with an extra parameter, which specifies the length of each part. The `striate'` function still scans across the sample over a single cycle, but if each bit is longer, it creates a sort of stuttering effect. For example the following will cut the bev sample into 32 parts, but each will be 1/16th of a sample long:

```
d1 $ slow 32 $ striate' 32 (1/16) $ sound "bev"
```

Smash

```
smash :: Int → [Time] → OscPattern → OscPattern
```

Smash is a combination of `spread` and `striate` - it cuts the samples into the given number of bits, and then cuts between playing the loop at different speeds according to the values in the list.

So this:

```
  d1 $ smash 3 [2,3,4] $ sound "ho ho:2 ho:3 hc"
```

Is a bit like this:

```
  d1 $ spread (slow) [2,3,4] $ striate 3 $ sound "ho ho:2 ho:3 hc"
```

This is quite dancehall:

```
d1 $ (spread' slow "1%4 2 1 3" $ spread (striate) [2,3,4,1] $ sound
"sn:2 sid:3 cp sid:4")
  |+| speed "[1 2 1 1]/2"
```

Combining patterns

Because Tidal patterns are defined as something called an "applicative functor", it's easy to combine them. For example, if you have two patterns of numbers, you can combine the patterns by, for example, multiplying the numbers inside them together, like this:

```
d1 $ (brak (sound "bd sn:2 bd sn"))
   |+| pan ((*) <$> sinewave1 <*> (slow 8 $ "0 0.25 0.75"))
```

In the above, the `sinewave1` and the `(slow 8 $ "0 0.25 0.75")` pattern are multiplied together. Using the <$> and the <*> in this way turns the * operator, which normally works with two numbers, into a function that instead works on two *patterns* of numbers.

Here's another example of this technique:

```
d1 $ sound (pick <$> "kurt mouth can*3 sn" <*> slow 7 "0 1 2 3 4")
```

The `pick` function normally just takes the name of a set of samples (such as `kurt`), and a number, and returns a sample with that number. Again, using <$> and <*> turns `pick` into a function that operates on patterns, rather than simple values. In practice, this means you can pattern sample numbers separately from sample sets. Because the sample numbers have been slowed down in the above, an interesting texture results.

By the way, "0 1 2 3 4" in the above could be replaced with the pattern generator `run 5`.

Juxtapositions

The `jux` function creates strange stereo effects, by applying a function to a pattern, but only in the right-hand channel. For example, the following reverses the pattern on the righthand side:

```
d1 $ slow 32 $ jux (rev) $ striate' 32 (1/16) $ sound "bev"
```

When passing pattern transforms to functions like `jux` and `every`, it's possible to chain multiple transforms together with `.`, for example this both reverses and halves the playback speed of the pattern in the righthand channel:

```
d1 $ slow 32 $ jux ((|+| speed "0.5") ∘ rev) $ striate' 32 (1/16) $ sound "bev"
```

Plus more to be discovered!

You can find a stream of minimal cycles written in Tidal in the following twitter feed: http://twitter.com/tidalcycles/

Acknowledgments

Special thanks to l'ull cec (http://lullcec.org) and hangar (http://hangar.org) for supporting the documentation and release of tidal as part of the ADDICTED2RANDOM project.