A Domain Specific Language For Drum Beat Programming

André Rauber Du Bois[1] **, Rodrigo Ribeiro**[1 2]

[1]PPGC - Universidade Federal de Pelotas, Pelotas - RS

[2]PPGCC - Universidade Federal de Ouro Preto, Ouro Preto - MG

dubois@inf.ufpel.edu.br, rodrigo@decsi.ufop.br

Abstract

This paper describes HDrum, a Domain Specific Language for writing drum patterns. Programs written in HDrum look similar to the grids, available in sequencers and drum machines, used to program drum beats, but as the language has an inductive definition we can write abstractions to manipulate drum patterns. HDrum is embedded in the Haskell functional programming language, hence it is possible to implement Haskell functions that manipulate patterns generating new patterns. The paper also presents a case study using HDrum, an implementation of *The Clapping Music*, a minimalistic music written by Steve Reich in 1972. The HDrum language is currently compiled into midi files.

1. Introduction

This paper presents the HDrum language, a domain specific language for drum pattern programming embedded in the Haskell functional programming language. Simple drum pattern programming in HDrum looks like an ASCII version of of the grids for drum beat programming in sequencers and drum machines. But the constructors used to describe patterns have a well defined inductive semantics which leads to interesting properties and allows the definition of functions over patterns and multi-tracks. HDrum provides two abstractions, patterns and tracks, the first is used do describe a drum beat and the second to associate an instrument to a pattern. HDrum is an algebra that defines the sets of patterns and tracks and also the functions that operate on these sets. Mainly HDrum provides operators for repetition, sequencing and parallel composition of patterns and tracks.

As the DSL is embedded in Haskell, it is possible to use all the power of functional programming in our benefit to define new abstractions over drum patterns. To understand the paper the reader needs no previous knowledge of Haskell, although some knowledge of functional programming and recursive definitions would help. We try to introduce the concepts and syntax of Haskell needed to understand the paper as we go along.

Although HDrum and its prototype implementation are only designed for drum beat programming the abstractions provided by the language can be applied for general music programming as well.

The paper is organized as follows. First we describe the main constructors for patterns (Section 2.1) and tracks (Section 2.2) design and their basic operations. Next, we examine the important abstraction of track composition (Section 2.3). Section 2.4, provides a discussion on simple algebraic properties of the language, e.g., the notion of equality, associativity, etc. The compilation of HDrum into midi files is explained in Section 3. A case study, the implementation of the minimalistic music *The Clapping Music*, is presented in Section 4. Finally, related work, conclusions and future work are discussed.

2. HDrum: Drum patterns for Haskell

2.1. Simple Drum Patterns

HDrum is an algebra (i.e., a set and the respective functions on this set) for drum pattern programming. The set of all drum patterns can be described inductively as an algebraic data type in Haskell:

```
data DrumPattern = X | O |
     DrumPattern :|  DrumPattern
```

The word `data` creates a new data type, in this case, `DrumPattern`. This definition says

that a drum pattern can be either a hit (X), a rest (O), or a sequential composition of patterns using the operator (:|), that takes as arguments two drum patterns and returns a new drum pattern.

As an example, we can define two 4/4 drum patterns, one with a hit in the 1st beat called kick and another that hits in the 3rd called snare.

```
kick :: DrumPattern
kick =  X :| O :| O :| O

snare :: DrumPattern
snare = O :| O :| X :| O
```

The symbol (::) is used for type definition in Haskell, and can be read as *has type*, e.g. kick has type DrumPattern.

As DrumPattern is a recursive data type, it is possible to write recursive Haskell functions that operate on drum patterns. For example, usually a certain pattern is repeated many times in a song, and a repeat operator (.*) for patterns can be defined as follows:

```
(.*) :: Int -> DrumPattern
            -> DrumPattern
1 .* p = p
n .* p = p :| (n-1) .* p
```

The repeat operator takes as arguments an integer n and a drum pattern p, and returns a drum pattern that is a composition of n times the pattern p. As can be seen in the previous example, the composition operator can combine drum patterns of any size and shape, e.g.:

```
hihatVerse :: DrumPattern
hihatVerse = 8 .* (X :| O :| X :| O)

hihatChorus :: DrumPattern
hihatChorus = 4 .* (X :| X :| X :| X)

hihatSong :: DrumPattern
hihatSong = hihatVerse :|
            hihatChorus :|
            hihatVerse :|
            hihatChorus
```

or simply:

```
hihatSong :: DrumPattern
hihatSong = 2 .* (hihatVerse :|
                  hihatChorus)
```

In order to make any sound, a drum pattern must be associated to an instrument hence generating a Track, as explained in the next section.

2.2. Tracks

A track is the HDrum abstraction that associates an instrument to a pattern. The Track data type is also defined as an algebraic type in Haskell:

```
data Track =
   MakeTrack Instrument DrumPattern
   | Track :|| Track
```

A simple track can be created with the MakeTrack constructor, which associates an Instrument to a DrumPattern. A Track can also be the *parallel* composition of two tracks, which can be obtained with the :|| operator. In the current implementation of the language, the instruments available are the different drum sounds of the midi protocol [1]. Instruments is also defined as an algebraic data type listing all possible instruments:

```
data Instrument =  AcousticBassDrum
    | BassDrum | SideStick
    | AcousticSnare | HandClap
    |  (...)
```

Now, we can use the previously defined patterns kick and snare to create tracks:

```
kickTrack :: Track
kickTrack = MakeTrack BassDrum kick

snareTrack :: Track
snareTrack =
   MakeTrack AcousticSnare snare
```

and also multi-tracks:

```
rockMTrack :: Track
rockMTrack =
 kickTrack :||
 snareTrack :||
 MakeTrack ClosedHiHat (X:|X:|X:|X)
```

2.3. Composing Tracks

The :|| operator allows the parallel composition of Tracks, i.e., adding an extra track to a multi-track song. But what if we want to compose tracks in sequence, e.g., we have different multi tracks for the introduction, verse and chorus, and want to combine them in sequence to form a complete song?

One problem that we need to deal with are the different sizes of patterns in a multi-track. The size of a multi-track, is the size of its largest pattern. It is important to notice that when composing tracks, we assume that smaller patterns have rest beats at their

```
track1 =
  MakeTrack BassDrum          (X)
  :|| MakeTrack AcousticSnare (O :| O :| X)
  :|| MakeTrack ClosedHiHat   (X :| X :| X :| X)

track2 =  MakeTrack BassDrum  (X :| O :| O :| O)
  :|| MakeTrack AcousticSnare (O :| O :| X :| O)
  :|| MakeTrack ClosedHiHat   (X :| O :| X )
  :|| MakeTrack Cowbell       (X)

track1track2 =
 MakeTrack BassDrum          (X :| O :| O :| O :| X :| O :| O :| O)
 :|| MakeTrack AcousticSnare (O :| O :| X :| O :| O :| O :| X :| O)
 :|| MakeTrack ClosedHiHat   (X :| X :| X :| X :| X :| O :| X )
 :|| MakeTrack Cowbell       (O :| O :| O :| O :| X )

track2track1 =
 MakeTrack BassDrum          (X :| O :| O :| O :| X)
 :|| MakeTrack AcousticSnare (O :| O :| X :| O :| O :| O :| X :| O)
 :|| MakeTrack ClosedHiHat   (X :| O :| X :| O :| X :| X :| X :| X)
 :|| MakeTrack Cowbell       (X)

track1twice =
  MakeTrack BassDrum          (X :| O :| O :| O :| X)
  :|| MakeTrack AcousticSnare (O :| O :| X :| O :| O :| O :| X )
  :|| MakeTrack ClosedHiHat   (X :| X :| X :| X :| X :| X :| X :| X)

track2twice =
  MakeTrack BassDrum          (X :| O :| O :| O :| X :| O :| O :| O)
  :|| MakeTrack AcousticSnare (O :| O :| X :| O :| O :| O :| X :| O)
  :|| MakeTrack ClosedHiHat   (X :| O :| X :| O :| X :| O :| X)
  :|| MakeTrack Cowbell       (X :| O :| O :| O :| X)
```

Figure 1: Composing tracks

end, hence all patterns are assumed to have the size of the largest pattern in a multi-track. We can define this concepts formally with the following recursive functions:

```
lengthDP :: DrumPattern -> Int
lengthDP O = 1
lengthDP X = 1
lengthDP (X:|p) = 1 + lengthDP p
lengthDP (O:|p) = 1 + lengthDP p
lengthDP (x:|y) = lengthDP x +
                  lengthDP y


lengthTrack :: Track -> Int
lengthTrack (MakeTrack _ dp) =
  lengthDP dp
lengthTrack ((MakeTrack _ dp):||t) =
   max (lengthDP dp) (lengthTrack t)
```

Where `lengthDP` recursively calculates the size of a drum pattern, and `lenghtTrack` finds out the size of the largest pattern in a track, i.e., the size of the track.

HDrum provides two constructs for composing tracks in sequence, a repetition operators `|*` and a sequecing operator `|+`. The repetition operator is similar to `.*` but operates on all patterns of a muti-track:

```
|* :: Int -> Track -> Track
```

It that takes an integer `n` and a multi-track `t` and repeats all patterns in all tracks `n` times adding the needed rest beats at the end of smaller tracks.

The semantics of composing two multi-tracks `t1` and `t2`, i.e., `t1 |+ t2` is as follows:

- First we add rest beats to the end of each track in `t1` that has matching instruments with tracks in t2, so that all those tracks have the same size as the largest pattern in `t1`
- Then, for all patterns `p1` in `t1` and `p2` in `t2` that have the same instrument `i`, we generate a new track `MakeTrack i (p1:|p2)`
- Finally, we add a pattern of rests the size of t1, to the beginning of all tracks in `t2` that were not composed with tracks in `t1` in the previous step

Hence the size of the composition of two tracks `t1` and `t2` is sum of the size of the largest pattern in `t1` with the largest pattern in `t2`.

In Figure 1 it is possible to see two tracks with different sizes of patterns inside (`track1` and `track2`) and their compositions, `track1track2` is the same as `track1 |+ track2` and `track2track1` is the same as `track2 |+ track1`. The `track1twice` track is equivalent to `2 |* track1` and `track2twice` is equivalent to `2 |* track2`.

2.4. Algebraic Properties

In this section we discuss some algebraic properties of drum patterns and tracks. Data types `DrumPattern` and `Track` provide a syntactic representation of music data, which allows different sound renderings. As an example, let `v1`,`v2` and `v3` be any value of type `DrumPattern`. Then, one should expect that `v1 :| (v2 :| v3)` will have the same meaning as `(v1 :| v2) :| v3`, i.e. sequential composition is an associative operation. From a semanticist point of view, such values are different, since they represent distinct syntactic entities, but they can have the same meaning using an appropriate notion of equality.

We consider two `DrumPatterns` or `Tracks` equal if they produce the same music. In order to define such equality precisely, we will need an algebra of drum music. Formally, an *algebraic structure* $\langle S, op_1, op_2, ...op_{n-1}\rangle$ is a n-uple formed by a non-empty carrier set S and operations over it. The algebraic structure of drum sounds is formed by a set $\mathcal{D}$ of drum music values with a distinct value ϵ to denote rest, functions to sequential and parallel composition, $\oplus$ and $\|$ respectively, a function for translating hits of a given instrument to its correspondent music value, namely $[\![\_]\!]_I$: `Instrument` $\rightarrow \mathcal{D}$ and a equivalence relation between $\mathcal{D}$ elements denoted as $v \equiv v'$, for some $v, v' \in \mathcal{D}$. We assume that $\|$ is commutative and both $\oplus$ and $\|$ are associative operators, with identity ϵ^0 for $\oplus$, ϵ^n for $\|$, where t^n denotes the parallel composition of n copies of t. When $n = 0$, t^0 denotes a rest of duration 0. We let $\mathcal{P}$ denote the set of pairs formed by a value of type `Instrument` and a value of type `DrumPattern`.

Translation of patterns and tracks is easily defined by recursion as follows as in Figure 2.

Using the semantics, we can define an equivalence relation for HDrum values. Let v_1 and v_2 be two drum patterns or tracks. We say that they are equivalent, $v_1 \approx v_2$, if and only if $[\![v_1]\!] \equiv [\![v_2]\!]$, where $[\![v_1]\!]$ denotes the semantics for patterns or tracks, respectively. Using this equivalence relation, we can check

$$
\begin{array}{lcl}
[\![\_,\_]\!]_D & : & \mathcal{P} \to \mathcal{D} \\
[\![i, \mathtt{X}]\!]_D & = & [\![i]\!]_I \\
[\![i, \mathtt{O}]\!]_D & = & \epsilon \\
[\![i, d_1 \mathtt{:|}\, d_2]\!]_D & = & [\![i, d_1]\!]_D \oplus [\![i, d_2]\!]_D
\end{array}
$$

$$
\begin{array}{lcl}
[\![\_]\!]_T : \mathtt{Track} \to \mathcal{D} & & \\
[\![\mathtt{MakeTrack\ i\ d}]\!]_T & = & [\![\mathtt{i,d}]\!]_D \\
[\![t_1 \mathtt{:||} t_2]\!]_T & = & [\![t_1]\!]_T \parallel [\![t_2]\!]_T
\end{array}
$$

Figure 2: Semantics of HDrum.

that HDrum patterns enjoys some algebraic properties. We list some of them below:

- Associativity: sequential and parallel composition are associative: For all `p1`, `p2` and `p3` of type `DrumPattern`, we have `p1 :| (p2 :| p3)` $\approx$ `(p1 :| p2) :| p3`. For all `t1`, `t2` and `t3` of type `Track`, we have `t1 :| (t2 :| t3)` $\approx$ `(t1 :| t2) :| t3`.
- Identity: $\mathtt{O}^0$ is the identity for sequential composition.
- Commutativity of parallel composition: for all tracks `t1` and `t2`, we have that `t1 :|| t2` $\approx$ `t2 :|| t1`.

Such properties are easily proved by induction over the structure of `DrumPatterns` and `Tracks`, using the respective properties of $\oplus$ and $\parallel$ operators.

3. Compiling HDrum into midi files

Midi is a standardized protocol to transmit real time information for the playback of a piece of music. The protocol defines a collection of messages that can be used to play music and for communication between Midi devices. Midi files contain the description of a piece of music, i.e., information such as what notes are played, when they are played, for how long and how loud. Specifically for the implementation of HDrum the important messages are the NOTE ON and NOTE OFF messages which tell when to start and stop playing a sound. Basically, the HDRum compiler traverses the data structure of patterns and tracks generating the appropriate NOTE ON and NOTE OFF messages for the drum instruments specified at tracks. We used Haskell's `Codec.Midi` library to generate the files. This library provides an algebraic data type defining all midi messages and handles the generation of binary files from a list of midi messages.

4. Case study: The clapping music

Minimalistic music is a type of art music, created in the early sixties, that uses minimal musical material. A famous piece created in this style is *The Clapping Music*, written by Steve Reich in 1972. The song was written to be performed by two people clapping the pattern in Figure 3. After 8 bars, one of the players shifts the pattern one eight note to the right. The player keeps shifting every eight bars until his pattern meets again the pattern of the first player. A video of Steve Reich himself playing the piece can be seen in [2].

The interesting thing about this song is that shifting bits is a very common operation in computer science, so me can actually program the song in HDrum.

Figure 3: The clapping music

The initial pattern of the song, can be implemented in HDrum as follows:

```
clappingPat :: DrumPattern
clappingPat = X :| X :| X :| O :|
              X :| X :| O :| X :|
              O :| X :| X :| O
```

We can shift a pattern using the following function:

```
shiftPat :: DrumPattern -> DrumPattern
shiftPat O = O
shiftPat X = X
shiftPat (O:|p) = p :| O
shiftPat (X:|p) = p :| X
shiftPat (p1:|p2) = (tailDP p1) :|
                    p2:|(headDP p1)
```

If the pattern to be shifted is just a simple hit or rest, `shiftPat` does nothing. If the first element of the pattern is a hit or rest, then we simply move them to the end. If the composed pattern is formed by two more complex patterns (`p1` and `p2`), we remove the first beat of `p1` and add it to the end of the pattern using `headDP` to get the first element of the pattern.

Now, it is possible to write a function that creates a pattern by shifting the beats of an initial pattern:

```
shiftMany :: Int -> Int
                 -> DrumPattern
                 -> DrumPattern
shiftMany 1 t p = t .* (shiftPat p)
shiftMany n t p =
   t .* shifted
   :| shiftMany (n-1) t shifted
         where shifted = shiftPat p
```

The `shiftMany` function takes two integers (`n` and `t`) and a pattern `p` as arguments and returns a new pattern that shifts `p n` times, and each shifted version of `p` is repeated `t` times.

Now it is possible to implement the song. The first pattern, the one that is not shifted, is just a repetition of `clappingPat`. The second pattern, starts with the basic pattern and then is shifted every eight bars. It must shift 12 times in order to become the initial pattern again. Hence we have:

```
fstPatCS :: DrumPattern
fstPatCS = 104 .* clappingPat

sndPatCS :: DrumPattern
sndPatCS = 8.* clappingPat
   :| shiftMany 12 8 clappingPat
```

and the final clapping song becomes:

```
clappingSong :: Track
clappingSong =
  MakeTrack HandClap fstPatCS
  :|| MakeTrack HandClap sndPatCS
```

5. Related Works

There has been a lot of work on designing programming languages for computer music. Due to lack of space, we discuss here the ones that are closer to HDrum. There are many DSLs for computer music based on functional languages, e.g. [3, 4, 5, 6, 7]. These languages usually provide means for playing notes and composing the sounds generated in sequence and in parallel. In these languages the programmer can write a sequence of notes and rests, and these sequences can also be combined in parallel. In HDrum, instead of having different notes in the same track, each track indicates when a single note is played, i.e., It is the repetition pattern of a single note, similar to what happens in grids of a drum machine. Although the symbols used in HDrum have semantic meaning, visually programs look like an ASCII version of the grids for writing drum beats available in modern sequencers. We believe that this approach makes it easier for someone that is used with sequencer tools to write simple tracks in HDrum with little knowledge of functional programming. Furthermore, as patterns are no associated with notes, patterns can be reused with different instruments when needed.

6. Conclusions and Future Work

This paper has described the HDrum language for drum beat programming and its abstractions. Although HDrum was designed for percussion instruments, the ideas presented here can be easily adapted for general music programming. Our ultimate goal is to design a full language for live music programming. The current implementation of HDrum can be found in [8].

References

[1] Midi Instruments. https://www.midi.org/specifications/item/gm-level-1-sound-set, May 2017.

[2] The Clapping Music. https://www.youtube.com/watch?v=hH1j06bMH-DQ&t=115s, May 2017.

[3] Alex McLean. Making programming languages to dance to: Live coding with tidal. In *FARM 2014*. ACM, 2014.

[4] Paul Hudak, Tom Makucevich, Syam Gadde, and Bo Whong. Haskore music notation: An algebra of music. *J. of Functional Programming*, 6(3), May 1996.

[5] H. Thielemann. Audio processing using haskell. In *DAFx'04*, 2004.

[6] Paul Hudak and David Janin. Tiled polymorphic temporal media. In *FARM 2014*. ACM, 2014.

[7] Paul Hudak. An algebraic theory of polymorphic temporal media. In *PADL*, 2004.

[8] HDrum Language. https://sites.google.com/site/hdrumlanguage/, May 2017.