Zachary Tom

CS 252

12-9-19

EDMDSL: A Domain Specific Language for Electronic Dance Music Production

Abstract

This paper presents EDMDSL, a domain specific language embedded in the Haskell functional programming language that can be used for music production. EDMDSL has two core data types- patterns and tracks. Patterns represent the concatenation of notes and rests, and tracks represent the association of patterns with samples that when aligned in parallel form a song. The language is compiled into Sonic Pi, which is an educational programming language designed to create music.

Introduction

EDM, or Electronic Dance Music, is a broad range of electronic music genres ranging from house and techno to dubstep and trap [5]. It has been widely produced on personal computers using software applications like Garageband, FL Studio, and Ableton. Consequently, a link between coding/software and music creation has been seen as the popularity of computer-based music production has grown with time. With that growing popularity however has also come a growing desire for a traditional approach to music production on a computer. Music composition software products enable instant feedback on a song in development, as an artist can throw together drums and synths, play the resulting song, and quickly change in/out any tracks as they see fit. This instant feedback can encourage a trial-and-error production process that many seek to avoid [4]. Thus, audio programming languages aim to emulate the traditional approach to music composition, the kind one would picture Mozart would do when sitting down at his piano to write a symphony. A number of audio programming languages already exist, including CSound, Pure Data, and Chuck among others [1]. EDMDSL is an addition to the family that focuses on the production of electronic dance music. Specifically, it focuses on the production of house music because house as a genre is simplistic and repetitive, therefore making it easier for inexperienced programmers to pick up and start composing.

Tech Stack

Before diving into the expressions of EDMDSL, the technologies used to build it will be discussed. EDMDSL is an embedded domain specific language, meaning it is implemented within a host language [3]. It is embedded in the Haskell functional programming language,

thus enabling EDMDSL to take full advantage of functional programming capabilities, such as the ability to write recursive functions that operate on patterns. This will be explored further in following sections. EDMDSL code is compiled into Sonic Pi. Sonic Pi is a live coding environment originally designed as an educational tool used for teaching coding and music composition in schools [2]. It is now used for many other applications including EDMDSL. Sonic Pi has a large library of electronic samples ranging from saws to plucks, and thus is a good tool for EDMDSL to use.

Expressions

EDMDSL has two core expressions that define the language. The first is patterns.

data Pattern = X | O | Pattern :| Pattern

The X represents a note, the O a rest, and the Pattern :| Pattern represents a concatenation of two patterns. Thus, via the concatenation operator, a pattern can be produced consisting of a multitude of notes and rests. In EDMDSL code, this looks like:

supersawP = X :| O :| X :| O

which indicates a pattern with a note on the first and third beats and rests on the second and fourth beats. Since the point of patterns is to allow for repetition, an operator is provided that facilitates recursive operations on patterns:

(∗ =) :: Int -> Pattern Pattern

This operator takes in an integer and a pattern. The integer indicates the number of repeats of the input pattern, and the function returns the composite patterns. Example expressions using the repeat operator are as follows:

pluckP = 10 ∗ = O :| O :| O :| X

This indicates 10 concatenations of a pattern of 3 rests and a note on the fourth beat. Thus, the ∗ = operator saves the work of having to write out a pattern multiple times, and is an example of why the functional programming capabilities of Haskell are useful for this specific project.

The other core expression of EDMDSL is tracks.

data Track = CreateTrack Sound Pattern | Track := Track

Where ‘CreateTrack’ is the constructor and ‘Sound’ is a string representing the sample in the Sonic Pi library that will be played. In EDMDSL code, this looks like:

supersawTrack = Supersaw supersawP

The (:=) operator enables the concurrent production of Tracks to produce a song. A coder can put together a bass drum track, snare track, and cymbals track to produce a beat, for

example, like so:

beatSong = := bassTrack := snareTrack := cymbalTrack

Implementation

EDMDSL is implemented via a parser, evaluation functions, and a backend that compiles the evaluated code into Sonic Pi code. The first step is the parsing of a .edmdsl file. A coder will write their song into a .edmdsl file and it may look something like the following:

drumbass = 10 ∗ = X :| O :| X :| O;

drumsnare = 10 ∗ = X :| X :| O :| X;

drumcymbal = 10 ∗ = O :| O :| X :| O;

bassTrack = drum_bass_hard drumbass;

snareTrack = drum_snare_hard drumsnare;

cymbalTrack = drum_cymbal_closed drumcymbal;

beatSong = := bassTrack := snareTrack := cymbalTrack

The above song is an example of a beat consisting of a bassdrum, snare drum, and cymbals, played for a total of 40 notes. The program parses the specified file into expressions. The expressions are then evaluated by a set of functions that convert them into a track. The track is then converted into Sonic Pi code via a series of backend functions.

trackToSonicPiHelper :: Float -> [[Sound]] -> Bool -> String

trackToSonicPiHelper restLength [] b = ""

trackToSonicPiHelper restLength (x:xs) b = produceNotes x b ++ "\ tsleep " ++ show restLength ++ "\ n" ++ trackToSonicPiHelper restLength xs b

produceNotes :: [Sound] -> Bool -> String

produceNotes [] b = ""

produceNotes (x:xs) b = if (b) then "\ tsample :" ++ x ++ ", rate: 1\ n" ++ produceNotes xs b else "\ tsynth :" ++ x ++ "\ n" ++ produceNotes xs b

The backend functions operate by first parsing out the sounds (the string variables representing samples in the Sonic Pi library) into a list format indicative of parsed pattern notes

and rests. This list is then passed to the functions above, which parse it directly into Sonic Pi code. Sonic Pi code is written to a text file in primitives including "sleep", "sample", and "synth". These values indicate the length of time to wait between notes, the drum samples of the Sonic Pi library, and the synth samples of the library respectively.

Once the Sonic Pi code is ready, the Sonic Pi server is accessed via the Sonic Pi Tool, which is a command line utility that enables messages to be sent to the Sonic Pi Server without using its GUI interface. The Sonic Pi Tool can be used to start the server, stop it, and even send code to be processed in real time. The primitives mentioned above are used in the backend of EDMDSL to dictate the sounds that should be played for every X and the rests for every O. A beats per minute parameter is passed to Sonic Pi as well to indicate the tempo at which the song should be played.

```
play :: Float -> Track -> Bool -> IO ()
play bpm track b = do
 writeFile "SonicPi_Code.txt" $ trackToSonicPi (60/bpm) track b
 v <- system $ sPTPath++"sonic-pi-tool eval-file SonicPi_Code.txt"
    print $ show v
```

Play is the main function that initiates compilation of EDMDSL code into Sonic Pi code. As can be seen in the code sample above, the function writes the compiled Sonic Pi code to a text file that is then passed to the Sonic Pi Tool for evaluation. The play function takes a boolean indicating whether the file is a beats file or not.

Currently, beats and synths are separated into separate .edmdsl files to be processed. This is due to the infrastructure of Sonic Pi. Sonic Pi has separate primitives for synths and samples (samples being the library that contains beats). EDMDSL in this version does not automatically check which kind of sample is being requested from Sonic Pi and thus beats and synths need to be separated.

Conclusion

In conclusion, EDMDSL is a domain specific language embedded in Haskell, created for music production. It is a simple language that compiles into Sonic Pi, allowing access to a diverse library of electronic samples and beats. Future directions for EDMDSL include implementing automatic type checking so that beats and synths can be written to the same .edmdsl file. Furthermore, EDMDSL does not currently support pitch specification. This would require pitch options to be associated with every note, and while difficult to implement in concise syntax, various approaches will be considered for further development. Finally, Sonic Pi supports looping and real time coding-to-music generation. Thus, EDMDSL will seek to facilitate live coding of music via the Haskell interpreter in the command line.

References

[1] Bois, André Rauber Du and Rodrigo Geraldo Ribeiro. "HMusic: A domain specific language for music programming and live coding." *NIME (2019).*

[2] "DROPS - Collaboration and learning through live coding (Dagstuhl Seminar 13382)". drops.dagstu Retrieved 2015-05-02.

[3] Fowler, Martin, and Rebecca Parsons. *Domain-Specific Languages. Addison-Wesley, 2011.*

[4] Layton, Julia. "Composing in Code: Musician-Programmers Are Changing the Way Music Is Made." *HowStuffWorks, HowStuffWorks, 3 Feb. 2016, entertainment.howstuffworks.com/con code-musicianprogrammers-are-changing-the-way-music-is-made.htm.*

[5] Mcleod, Kembrew. "Genres, Subgenres, Sub-Subgenres and More: Musical and Social Differentiation Within Electronic/Dance Music Communities." *Electronica, Dance and Club Music, 2017, pp. 289–305., doi:10.4324/9781315094588-19.*