HMusic: A domain specific language for music programming and live coding

André Rauber Du Bois
Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas
Pelotas - RS - Brazil
dubois@inf.ufpel.edu.br

Rodrigo Geraldo Ribeiro
Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto
Ouro Preto - MG - Brazil
rodrigo@decsi.ufop.br

ABSTRACT

This paper presents HMusic, a domain specific language based on music patterns that can be used to write music and live coding. The main abstractions provided by the language are patterns and tracks. Code written in HMusic looks like patterns and multi-tracks available in music sequencers, drum machines and DAWs. HMusic provides primitives to design and compose patterns generating new patterns. The basic abstractions provided by the language have an inductive definition and HMusic is embedded in the Haskell functional programming language, hence programmers can design functions to manipulate music on the fly. The current implementation of the language is compiled into Sonic Pi [10] and can be downloaded from [9].

Author Keywords

Live coding, Functional Programming, Haskell

CCS Concepts

•Applied computing → Sound and music computing; *Performing arts;* **•Software and its engineering → Functional languages;**

1. INTRODUCTION

Computers are generic abstract machines that can be programmed with different goals in a variety of domains, including arts in general, and music. Computer music is usually associated with the use of software applications to create music, but on the other hand, there is a growing interest in programming languages that let artists write software as an expression of art. There are a number of programming languages that allow artists to write music, e.g., CSound [2], Max [13, 28], Pure Data [23], Supercollider [19], Chuck [27], FAUST [22], to name a few. Besides writing songs, all these languages also allow the live coding of music. Live coding is the idea of writing programs that represent music while these programs are still running, and changes in the program affect the music being played without breaks in the output [21].

This paper presents HMusic, a Domain Specific language for music programming and live coding. HMusic is based on the abstraction of patterns and tracks where the code looks very similar to the grids available in sequencers, drum machines and DAWs. The difference is that these abstractions have an inductive definition, hence programmers can write functions that manipulate these tracks in real time. 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 patterns of songs. 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.

The contributions of this paper are as follows:

- We describe the design and implementation of HMusic, a DSL for music programming that provides the abstractions of patterns and tracks, together with a set of functions to manipulate and combine these abstractions. The interesting aspect of the language is that basic programs look like the grids available in drum machines and sequencers, which is a concept familiar to music composers.
- We describe a simple interface for live coding based on looping tracks and function application to modify tracks in real time.

In the current implementation of HMusic, tracks can load pre-recorded samples. As it is currently compiled into Sonic Pi [10], any sample accessible by the Sonic Pi environment can be loaded and manipulated in tracks. The current implementation of the HMusic language can be downloaded from [9].

The paper is organized as follows. First we describe the main constructors for pattern (Section 2.1) and track (Section 2.2) design and their basic operations. Next, we examine the important abstraction of track composition, i.e., combining different multi-tracks to form a new track (Section 2.3). The abstraction provided by HMusic for live coding is presented in Section 3. The compilation of HMusic into Sonic Pi is explained in Section 4. Finally, related work, conclusions and future work are discussed.

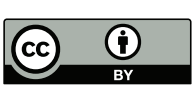
2. HMUSIC

2.1 HMusic Patterns

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

```
data MPattern = X | O | MPattern :|  MPattern
```

The word `data` creates a new data type, in this case, `MPattern`. This definition says that a pattern can be either a



NIME'19, June 3-6, 2019, Federal University of Rio Grande do Sul, Porto Alegre, Brazil.

playing a sample (`X`), a rest (`O`), or a sequential composition of patterns using the operator (`:|`), that takes as arguments two music patterns and returns a new 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 :: MPattern
kick =  X :| O :| O :| O

snare :: MPattern
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 `MPattern`.

As `MPattern` is a recursive data type, it is possible to write recursive Haskell functions that operate on 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 -> MPattern
            -> MPattern
1 .* p = p
n .* p = p :| (n-1) .* p
```

The repeat operator takes as arguments an integer `n` and a pattern `p`, and returns a 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 :: MPattern
hihatVerse = 8 .* (X :| O :| X :| O)

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

hihatSong :: MPattern
hihatSong = hihatVerse :|
            hihatChorus :|
            hihatVerse :|
            hihatChorus
```

or simply:

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

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

2.2 HMusic Tracks

A track is the HMusic 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 MPattern
   | Track :|| Track

type Instrument = String
```

A simple track can be created with the `MakeTrack` constructor, which associates an `Instrument` to a `MPattern`. A `Track` can also be the *parallel* composition of two tracks, which can be obtained with the `:||` operator. `Instrument` is a type synonym for `String`s. An instrument can be any audio file accessible by the Sonic Pi environment (see Section 4).

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) :||
 MakeTrack "GuitarSample" 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 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:

```
lengthMP :: MPattern -> Int
lengthMP O = 1
lengthMP X = 1
lengthMP (x:|y) = lengthMP x +
                  lengthMP y


lengthTrack :: Track -> Int
lengthTrack (MakeTrack _ dp) =
  lengthMP dp
lengthTrack (t1 :|| t2) =
   max (lengthTrack t1) (lengthTrack t2)
```

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

HMusic provides two constructs for composing tracks in sequence, a repetition operator `|*` and a sequencing 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`

```
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 "GuitarSample"        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 "GuitarSample"       (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 "GuitarSample"          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 "GuitarSample"  (X :| O :| O :| O :| X)
```

Figure 1: Composing tracks

- 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 equivalent to `track1 |+ track2` and `track2track1` is equivalent to `track2 |+ track1`. The `track1twice` track is equivalent to `2 |* track1` and `track2twice` is equivalent to `2 |* track2`.

```
play :: Float -> Track -> IO ()
loop :: Float -> Track  -> IO ()
applyToMusic :: (Track -> Track) -> IO ()
```

Figure 2: HMusic primitives for live coding

3. LIVE CODING WITH HMUSIC

HMusic provides a set of primitives for playing tracks and live coding. These primitives allow programmers to play songs written in HMusic, loop tracks, and to modify tracks on the fly, i.e., while they are being played. These primitives can be seen in Figure 2.

The first primitive, `play`, takes two arguments: a `Float`, which is the BPM (Beats per Minute) of the song and a track, and simply plays this track in the BPM provided. The `loop` function also takes the same arguments but will loop the track in the BPM provided. If a loop is already being played, it will be substituted by the new one. The `applyToMusic` function can be used to modify the current pattern being played. It takes as argument a function from `Track` to `Track` and applies it to the pattern being looped.

These functions can be called in the Haskell interpreter (GHCi [3]) to live code music. Here is a simple example of a live code session. We start by looping a simple multi-track that contains only snare and kick:

```
*HMusic> loop 120 (kickTrack :|| snareTrack)
```

This call will start looping at 120 BPM a parallel composition of the `kickTrack` and `snareTrack` defined priviously in Section 2.2. Next, we can add to the loop being played another track with a hi-hat:

```
*HMusic> applyToMusic  (:|| MakeTrack "ClosedHiHat" (X:|
X:|X:|X))
```

In this example, we are using partial application to transform the parallel composition operator (that has type `Track -> Track -> Track`) into a function that takes only one argument, i.e., `Track -> Track`.

Next, we can add a guitar sample in the beginning of the loop:

```
*HMusic> applyToMusic  (:|| MakeTrack "guitarSample" X)
```

Instead of adding new tracks, we can also substitute tracks. For example, we can write a Haskell function that substitute an instrument that is being played in one track by a different instrument:

```
subsInstrument :: Instrument -> Instrument ->
                  Track -> Track
subsInstrument i1 i2 (MakeTrack i p)
   | i == i1 = MakeTrack i2 p
```

```
   | otherwise = (MakeTrack i p)
subsInstrument i1 i2 (t1 :|| t2) =
     subsInstrument i1 i2 t1
     :|| subsInstrument i1 i2 t2
```

The `subsInstrument` function takes two instruments and a multi-track as arguments and substitute any occurrence of the the first instrument by the second in the argument track generating a new track.

For example, we can use `subsIntrument` to substitute the guitar sample for an organ sample:

```
applyToMusic (subsInstrument "guitarSample" "orgSample")
```

3.1 Algebraic Properties

In this Section we discuss some algebraic properties of drum patterns and tracks. Data types `MPattern` 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 `MPattern`. 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 `MPattern`s or `Track`s equal if they produce the same music. In order to define such equality precisely, we will need an algebra of music values. 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 musical sounds is formed by a set $\mathcal{D}$ of music values with a distinct value ϵ to denote silence, functions to sequential and parallel composition, $\oplus$ and $\|$ respectively, a function for translating samples 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. We let $\mathcal{P}$ denote the set of pairs formed by a value of type `Instrument` and a value of type `MPattern`.

Translation of patterns and tracks is easily defined by recursion as follows as in Figure 3.

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

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

Figure 3: Semantics of HMusic.

Using the semantics, we can define an equivalence relation for HMusic 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 that HMusic 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 `MPattern`, 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`.
- 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 `MPattern`s and `Track`s, using the respective properties of $\oplus$ and $\|$ operators.

4. COMPILING HMUSIC INTO SONIC PI

The current implementation of HMusic compiles patterns and tracks into Sonic Pi [10] code. Sonic Pi is an educational programming language created with the objective of teaching programming to kids through the creation of music. It is an open source tool originally developed for the Raspberry Pi processor but it is also available for different platforms such as Windows, Linux and macOS. Although the tool has been initially designed for pedagogical purposes, it is currently being used by a variety of musicians for live coding performances.

To compile the abstractions provided by HMusic into Sonic Pi code, we use a small set of primitives provided by the language, such as loops, rests and playing sound samples:

- `live_loop`: loops the sound generated by a set of Sonic Pi instructions
- `sleep n`: makes the current thread wait for n seconds (or a fraction) before playing the sound generated by the next instructions
- `sample :audofile`: plays the `audiofile`

We use the BPM parameter of functions `loop` and `play` to calculate the time for each beat, e.g., for 100 BPM, 100 beats will be played in 60 seconds. For example, if we choose to loop the following multi-track

```
track =
 MakeTrack "kick"       X
 :|| MakeTrack "snare" (O :| O :| X)
 :|| MakeTrack "hihat" (X :| X :| X :| X)
 :|| MakeTrack "Guitar" X
```

at 100 BPM, the generated Sonic Pi code will be

```
live_loop :hmusic do
        sample :kick
        sample :hihat
        sample :Guitar
        sleep 0.6
        sample :hihat
        sleep 0.6
        sample :snare
        sample :hihat
        sleep 0.6
        sample :hihat
        sleep 0.6
end
```

In order to interact with the Sonic Pi server, we used the Sonic Pi Tool [5], which is a command line utility that allows to send messages to the Sonic Pi server without using its GUI interface. It provides commands to start the Sonic Pi server, stop it, and also allows to send code to be processed in real time. The Haskell's `System.Cmd` interface [6], which is a simple library to call external commands in Haskell, was used to implement the functions `play`, `loop` and `applyToMusic`. These functions transform HMusic tracks into Sonic Pi code, and use the `System.Cmd` library to access the sonic server tool and execute the generated music. When looping, the code for the current track being played is held

in a global `IORef` [17], which is basically a pointer to the track being played. The `loop` function substitutes the track being played, and `applyToMusic` will modify it, compile it again, and send it to the Sonic Pi server through the Sonic Pi tool.

5. RELATED WORKS

There has been a lot of work on designing programming languages for computer music and live coding. Most of these languages, e.g., CSound [2], Max [13, 28], Pure Data [23], Supercollider [19], Chuck [27], FAUST [22] etc, are based on the idea of dataflow programming, where signal generators and processors can be connected either visually or through code, providing the abstraction of streams of data/sound that can be combined and processed. Some languages for music programming e.g., Gibber [24] and IXI Lang [18], and music notation languages e.g., LilyPond [4] and abc notation [1], also provide ways of describing patterns and/or tracks, but do not focus on their composition/combination.

There are many DSLs for computer music based on functional languages, e.g. [20, 16, 25, 15, 14]. 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 HMusic, instead of having different sounds in the same track, each track indicates when a single sound is played, i.e., It is the repetition pattern of a single sound, similar to what happens in grids of a drum machine and sequencers. Although the symbols used in HMusic 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 HMusic with little knowledge of functional programming. Furthermore, as patterns are not associated with sounds, patterns can be reused with different instruments when needed.

Some of the abstractions provided here are based on previous work by the authors [12]. The language presented in the previous work allows only drum beat programming, and was compiled into midi files. No loading of samples or live coding was supported.

The formal semantics of a language with support for live coding is the subject of Aaron et. al. work [11]. The authors discuss some problems with the semantics of Sonic-Pi `sleep` function and propose a formalization to fix the problem while being compatible with Sonic-Pi previous versions. The work introduces the notion of *time-safety* and shows that Sonic-Pi's new semantics is time-safe. Time safety is an important notion when programs consists of multiple threads that need to cooperate to produce a music. Since HMusic semantics is compiled to Sonic Pi, it enjoys the time safety property. We let the formalization of HMusic compilation process and its extension to support multi-thread programs, like Sonic-Pi, to future work.

6. CONCLUSIONS

This paper described HMusic, a Domain Specific Language for music programming and live coding, that is embedded in the Haskell functional programming language. The main abstractions of the language are patterns and tracks, that look similar to the grids available in sequencers and DAWs. The language also supports primitives for composing and manipulating patterns and tracks, e.g., concatenation, repetition, and parallel composition. Besides, these abstractions have an inductive definition, hence programmers can write Haskell functions to manipulate these tracks generating new abstractions for the language. HMusic also supports live coding through a very simple interface based on the idea of looping and function application. Programmers can manipulate and modify music in real time using the basic abstractions of the language or create new functions for this purpose.

The language is still in its early stages of development. We believe that there are a number of lines for future work. Development of music and live coding in HMusic would be much easier with a special editor that could, either visually or with options in a menu, generate automatically empty tracks of a desired size, with the programmer being responsible for filling the hits. One simple way of obtaining such a feature is using Emacs macros [8]. The system implemented to support HMusic could be easily extended for collaborative live coding, where different programmers interact with the music at the same time. HMusic tracks can be converted into strings of text using Haskell's `Read` and `Show` type classes [26], hence a simple interface for collaborative live coding can be obtained with a socket server that receives code, which is processed locally in the clients, and sends to be run on a central Sonic PI server. Elm [7], is a functional programming language with syntax and many features similar to Haskell. It is compiled into JavaScript, and used to create web browser-based graphical user interfaces. We believe that HMusic could easily be ported to Elm which would allow web-based music performances. We are currently extending the language to support a new type of track in which effects can be added:

```
MakeTrackE Instrument [Effect] MPattern
```

Besides `Instrument`s these tracks can take as argument a list of effects that are applied in order. Effects available in Sonic Pi will be able to be loaded in tracks, like changing the rate of samples, reverb, amp, etc. For the semantics of track composition, only tracks that have the same instruments and effects in the same order are considered the same. The reviewers of this paper mentioned the necessity of operators for track composition that do not add rest beats to the end of tracks. Such operators are easier to implement than the ones described here, and will be available in the next version of the language.

HMusic is a new language, and It is difficult to know if the abstractions proposed here would be of any help for real live coders. We plan in the future to spread the word about the language, initially in the Computer Science Department and Music Departments of our University, through local talks and performances, in order to get feedback with the objective of improving the language.

7. ACKNOWLEDGMENTS

This work was supported by CAPES/Brasil (Programa Nacional de Cooperação Acadêmica da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

8. REFERENCES

[1] ABC Notation. http://abcnotation.com/, January 2019.
[2] CSound. http://csound.com, January 2019.
[3] Glasgow Haskell Compiler. https://www.haskell.org/ghc/, January 2019.
[4] LilyPond. http://lilypond.org/, January 2019.
[5] Sonic Pi Tool. https://github.com/lpil/sonic-pi-tool, January 2019.

[6] System.Cmd. http://hackage.haskell.org/package/process-1.6.5.0/docs/System-Cmd.html, January 2019.

[7] The Elm Programming Language. https://elm-lang.org/, January 2019.

[8] The Emacs editor. https://www.gnu.org/software/emacs/, January 2019.

[9] The HMusic DSL. https://github.com/hmusiclanguage/hmusic, January 2019.

[10] S. Aaron, A. F. Blackwell, and P. Burnard. The development of sonic pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming. *Journal of Music, Technology and Education*, 9:75–94, 05 2016.

[11] S. Aaron, D. Orchard, and A. F. Blackwell. Temporal semantics for a live coding language. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, FARM '14, pages 37–47, New York, NY, USA, 2014. ACM.

[12] A. R. D. Bois and R. G. Ribeiro. A domain specific language for drum beat programming. In *Proceedings of the Brazilian Symposium on Computer Music*, 2017.

[13] E. Favreau, M. Fingerhut, O. Koechlin, P. Potacsek, M. Puckette, and R. Rowe. Software developments for the 4x real-time system. In *International Computer Music Conference*, 1986.

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

[15] P. Hudak and D. Janin. Tiled polymorphic temporal media. In *FARM 2014*. ACM, 2014.

[16] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation: An algebra of music. *J. of Functional Programming*, 6(3), May 1996.

[17] S. P. Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. Press, 2002.

[18] T. Magnusson. The ixi lang: A supercollider parasite for live coding. In *International Computer Music Conference*, 2011.

[19] J. McCartney. Supercollider, a new real time synthesis language. In *International Computer Music Conference*, 1996.

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

[21] C. A. McLean. *Artist-Programmers and Programing Languages for the Arts*. PhD thesis, University of London, 2011.

[22] Y. Orlarey, D. Fober, and S. Letz. Faust : an efficient functional approach to dsp programming. In *New Computational Paradigms for Computer Music*, 2009.

[23] M. Puckette. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference*, pages 37–41, 1996.

[24] C. Roberts, M. K. Wright, and J. Kuchera-Morin. Music programming in gibber. In *ICMC*, 2015.

[25] H. Thielemann. Audio Processing Using Haskell. In *DAFx04*, 2004.

[26] S. Thompson. *The Craft of Functional Programming*. Addison-Wesley, 2011.

[27] G. Wang and P. Cook. Chuck: A programming language for on-the-fly, real-time audio synthesis and multimedia. pages 812–815, 01 2004.

[28] D. Zicarelli. How i learned to love a program that does nothing. *Computer Music Journal*, 26(4):44–51, 2002.