Combining Effects in a Music Programming Language based on Patterns

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

[1]Programa de Pós-Graduação em Computação
Universidade Federal de Pelotas, Pelotas-RS, Brazil

[2]Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Ouro Preto, Ouro Preto - MG, Brazil

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

Abstract. *HMusic is 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 combine patterns generating new patterns. The objective of this paper is to extend the original design of HMusic to allow effects on tracks. We describe new abstractions to add effects on individual tracks and in groups of tracks, and how they influence the combinators for track composition and multiplication. HMusic allows the live coding of music and, as it is embedded in the Haskell functional programming language, programmers can write functions to manipulate effects on the fly. The current implementation of the language is compiled into Sonic Pi [1], and we describe how the compiler's back-end was modified to support the new abstractions for effects. HMusic can be and can be downloaded from [2].*

1 Introduction

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 [3], Max [4, 5], Pure Data [6], Supercollider [7], Chuck [8], FAUST [9], 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 [10].

HMusic [11] is 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.

This paper discusses new abstractions for HMusic to deal with effects. More precisely, the contributions of this paper are as follows:

- We extend the abstractions of HMusic to incorporate effects. Basically, two new types of tracks are added: a track that takes a list of effects that are applied in order to the track's pattern, and a master track that applies a set of effects to a multi-track (Section 3.1)
- HMusic provides two operators for combining multi-tracks, a sum operator that takes two multi-tracks and generates a new track that plays the two multi-tracks one after the other, and a multiplication operator that takes an integer n and a multi-track t and generates a track that is n times t. We extend the behaviour of these operations to deal with effects and explain the semantics of track composition in the presence of effects (Section 3.3)
- We show how the new abstractions for effects can be used during a live coding session (Section 4)
- We describe how the new abstractions presented in this paper can be compiled into Sonic Pi code (Section 4)

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 paper is organized as follows. First we describe HMusic and the main constructors for pattern (Section 2.1) and track (Section 2.2) design and their basic operations. Next, the extensions for effects are explained (Section 3.1). In Section 3.3, we examine the semantics of track composition, i.e., combining different multi-tracks to form a new track, in the presence of effects. Live coding with effects is explained in Section 4. The compilation of HMusic with effects into Sonic Pi is described in Section 5. 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

*Supported by CAPES.

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 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 5).

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
```

3 Effects in HMusic

In this paper, the abstractions of HMusic are extended to incorporate effects. The new abstractions allow to add effects in individual tracks (Section 3.1) and in a group of tracks (Section 3.2). The use of effects in live coding is discussed in Section 4.

3.1 Effects on Tracks

To incorporate effects on tracks, the `MTrack` data type was extend with a new type of track:

```
data MTrack = (...)
  | MakeTrackE Instrument [Effect] MPattern
```

Besides `Instrument`s these tracks can take as argument a list of effects that are applied in order. In the current implementation, effects available in Sonic Pi can be loaded in tracks (see Section 5), like changing the rate of samples, reverb, amp, etc:

```
data Effect = Reverb Float | Amp Float
     | Attack Float | Rate Float
     | Sustain Float | (...)
```

For example, we can now write a drum multitrack which adds a bit of reverb on the snare:

```
drums :: MTrack
drums =
 MakeTrackE "snare" [Reverb 0.3] snare :||
 MakeTrack "kick" kick :||
 MakeTrack "hihat" (X:|X:|X:|X)
```

As the `MTrack` data type has an inductive definition, we can write recursive functions that manipulate effects and tracks (e.g., add, remove, modify) while music is being played, as described in Section 4.

3.2 Effects on Groups of Tracks

HMusic was also extend to support the addition of effects in a group of tracks:

```
data MTrack = (...)
  | Master  [Effect] MTrack
```

The `Master` track takes two arguments, a list of effects and an `MTrack`, which is possibly a multi-track, and ads these effects in order to the `MTrack`.

For example, we can now add effects to the whole `drums` track defined previously:

```
drumsE :: MTrack
drumsE = Master [Amp 0.2, Sustain 0.4] drums
```

Since the snare track already has a reverb applied to it, when the `Master` track is added, the effects applied to the snare are now `Amp`, followed by `Sustain`, followed by `Reverb`.

3.3 Effects and Track Composition

The `:||` operator, discussed previously, allows the parallel composition of `Tracks`, i.e., adding an extra track to a multi-track song. But what if we want to combine 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?

If we reason that different multi-tracks represent different parts of a song, when we combine two multi-tracks, we want the second multi-track to start playing only when the first one finished. Hence, when combining a multi-track, its size is always the size of its largest pattern. Thus, when combining tracks we assume that smaller patterns have rest beats at their end, meaning that 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 (x:|y) = lengthMP x +
                  lengthMP y
lengthMP _ = 1


lengthTrack :: Track -> Int
lengthTrack (MakeTrack _ dp) =
  lengthMP dp
lengthTrack (MakeTrackE _ _ dp) =
  lengthMP dp
lengthTrack (Master _ t) = lengthTrack t
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 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.

An operator for combining two multi-tracks `t1` and `t2`, generating a new multi-track is also provided:

```
|+ :: Track -> Track -> Track
```

When combining two multi-tracks, tracks that use the same instruments and effects are merged. 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 and effects 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` and effects `e`, we generate a new track `MakeTrack i (p1:|p2)` for simple tracks, and `MakeTrackE i e (p1 :| p2)` for tracks with effects. Master tracks are only merged if their internal tracks have the same instruments and effects. Internal tracks of a master track are merged using the rules stated above.
- 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 some examples of track combination are presented, where track `te1te2` is equivalent to `t1 |+ te2`, and `te3te4` is equivalent to `te3 |+ te4`, and `te3twice` is equivalent to `2 |* te3`.

4 Effects and Live Coding

4.1 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

```
te1 =
  MakeTrack "bassDrum"                          (X :| O :| O)
  :|| MakeTrackE "snare"   [Amp 0.5]            (O :| O :| X)
  :|| MakeTrackE "cymbal"  [Reverb 0.3]         (X :| X :| X :| X)

te2 =  MakeTrack "bassDrum"           (X :| O :| O :| O)
  :|| MakeTrackE "snare" [Amp 0.5]    (O :| O :| X :| O)
  :|| MakeTrack "HiHat"               (X :| O :| X )
  :|| MakeTrack "GuitarSample"         X


te3 = Master [Reverb 1.0] te1
      :|| MakeTrack "Cowbell"        (X :| O :| X )

te4 = Master [Reverb 1.0] te1
   :|| MakeTrack "GuitarSample"       X

te1te2 =  MakeTrack "bassDrum"              (X :| O :| O :| O :| X :| O :| O :| O)
  :|| MakeTrackE "snare" [Amp 0.5]          (O :| O :| X :| O :| O :| O :| X :| O)
  :|| MakeTrackE "cymbal" [Reverb 0.3]      (X :| X :| X :| X)
  :|| MakeTrack "HiHat"                     (O :| O :| O :| O :| X :| O :| X)
  :|| MakeTrack "GuitarSample"              (O :| O :| O :| O :| X)

te3te4 = Master [Reverb 1.0]
         (MakeTrack "bassDrum               (X :| O :| O :| O :| X)
       :|| MakeTrackE "snare" [Amp 0.5]     (O :| O :| X :| O :| O :| O :| X)
       :|| MakeTrackE "cymbal" [Reverb 0.3] (X :| X :| X :| X :| X :| X :| X :| X))
 :|| MakeTrack "GuitarSample"               (O :| O :| O :| O :| X)
 :|| MakeTrack "Cowbell"                    (X :| O :| X)

te3twice =
  Master [Reverb 1.0]
     (MakeTrack "bassDrum"                 (X :| O :| O :| O :| X :| O :| O :| O)
      :|| MakeTrackE "snare" [Amp 0.5]     (O :| O :| X :| O :| O :| O :| X :| O)
      :|| MakeTrackE "cymbal" [Reverb 0.3] (X :| X :| X :| X :| X :| X :| X :| X))
 :|| MakeTrack "Cowbell"                   (X :| O :| X :| O :| X :| O :| X :| O)
```

Figure 1: Combining tracks

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

Figure 2: HMusic primitives for live coding

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 [12]) 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)
```

The `map` function is omnipresent in functional programming languages and it is used to apply a function to all elements of a list. We can easily define a similar function for tracks:

```
mapTrack :: (Track -> Track) -> Track
            -> Track
mapTrack f (t1 :|| t2) =
     mapTrack f t1 :|| mapTrack f t2
mapTrack f t = f t
```

The `mapTrack` function can be used to modify tracks while they are being played. For example, we could write a function to substitute instruments of simple tracks:

```
subsInstr i1 i2 t@(MakeTrack i p)
  | i == i1 = MakeTrack i2 p
  | otherwise = t
subsInstr i1 i2 t = t

*HMusic> applyToMusic (mapTrack
(subsInstr "ClosedHiHat" "OpenHiHat"))
```

4.2 Effects in Live Coding

With HMusic, it is possible to dynamically add and remove effects from tracks being played. For example, it is possible to add effects to the multi-track being played:

```
*HMusic> applyToMusic
(\track -> Master [Amp 1.0] track)
```

Here we used a lambda abstraction to create a function that takes a track as an argument (the current track being played), and creates a master track around it.

As another example, we can write a function that substitutes a set of effects in a track by another set of effects:

```
subsEffects :: [Effect] -> [Effect]
               -> Track -> Track
subsEffects e1 e2 t@(MakeTrackE i e f)
  | e1 == e = MakeTrackE i e2 f
  | otherwise = t
subsEffects e1 e2 t = t
```

and use it to modify the parameter of an effect:

```
*HMusic> applyToMusic
(subsEffects [Reverb 1.0] [Reverb 0.3])
```

or to substitute a set of effects by another:

```
*HMusic> applyToMusic
(subsEffects [Reverb 1.0, Amp 1.0]
             [Attack 0.75, Release 0.75])
```

It is also possible to add effects to simple tracks by turning them into an effect track:

```
changeTrack :: Instrument -> [Effect]
               -> Track -> Track
changeTrack i e t@(MakeTrack it p)
  | i == it = MakeTrackE i e p
  | otherwise = t
changeTrack i e t = t
```

and use it to add an effect to a snare track:

```
*HMusic> applyToMusic
(changeTrack "snare" [Reverb 1.0])
```

We also provide a named master track:

```
data MTrack = (...)
  | MasterN  String [Effect] MTrack
```

Naming master tracks helps programmers to easily modify Master tracks (e.g., modify the effects being used, modify internal tracks by adding effects) while live coding. For example, we can define a named track for drums using the priviously definded `drums` track:

```
drumsN :: MTrack
drumsN = MasterN "drums" [Amp 0.2] drums
```

and while it is being played, we can modify its effects with the `changeMaster` function:

```
changeMaster :: String -> [Effect]
                -> Track -> Track
changeMaster name e t@(Master n em tm)
  | name == n = Master n e tm
  | otherwise = t
changeMaster i e t = t


*HMusic> applyToMusic
(changeMaster "drums" [Sustain 1.0])
```

```
trackcomp =
   MakeTrack "kick"                                                  X
   :|| MakeTrackE "snare" [Reverb 0.5]                            (O :| O :| X)
   :|| MakeTrackE "hihat" [Attack 0.1, Sustain 0.3, Release 0.1] (X :| X :| X :| X)
   :|| MakeTrack "guitar"                                            X

live_loop :hmusic do
    sample :kick
    sample :hihat, attack: 0.1, sustain: 0.3, release: 0.1
    sample :guitar
    sleep 0.6
    sample :hihat, attack: 0.1, sustain: 0.3, release: 0.1
    sleep 0.6
    sample :snare, reverb: 0.5
    sample :hihat, attack: 0.1, sustain: 0.3, release: 0.1
    sleep 0.6
    sample :hihat, attack: 0.1, sustain: 0.3, release: 0.1
    sleep 0.6
end
```

Figure 3: A multi-track and its code representation in Sonic Pi

5 Compiling HMusic into Sonic Pi

The current implementation of HMusic compiles patterns and tracks into Sonic Pi [1] 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, playing sound samples and effects:

- `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`
- Effects: Sonic Pi provides a set of effects that can be applied to sound samples, e.g., Amp, Reverb, Sustain, Attack, Release, etc.

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. Figure 3 shows a multi-track and the generated code in Sonic Pi for looping the track at 100 BPM.

In order to interact with the Sonic Pi server, we used the Sonic Pi Tool [13], 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 [14], 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` [15], 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.

6 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 [3], Max [4, 5], Pure Data [6], Supercollider [7], Chuck [8], FAUST [9] 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 [16] and IXI Lang [17], and music notation languages e.g., LilyPond [18] and abc notation [19], also provide ways of describing patterns and/or tracks, but do not focus on their composition/combination. In languages like Gibber and IXI Lang, patterns may contain different sounds, while in the abstraction provided by HMusic, patterns are associated with a single sound, just like in sequencers. Patterns in live coding languages can usually be routed to effects, while in HMusic effects are associated to individual tracks or multi-tracks.

There are many DSLs for computer music based on functional languages, e.g. [20, 21, 22, 23, 24]. 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 and applied to effects. 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. HMusic is an extension of a language called HDrum [25]. HDrum is a language for drum beat programming, and is compiled into midi files. No loading of samples or live coding is supported.

The formal semantics of a language with support for live coding is the subject of Aaron et. al. work [26]. 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.

7 Concluding Remarks

This paper described how to extend HMusic with abstractions for effects. Two new types of tracks were added to the language, a track that allows the application of effects on patterns and a track that allows effects on multi-tracks. The implications of track composition/combination in the presence of effects were also discussed. HMusic provides a small set of primitives for playing music and also for live coding, and we demonstrated that the new abstractions for effects can also be manipuled on the fly in a live code session.

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 [27]. 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 [28], 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 [29], 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.

Acknowledgments

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

References

[1] Samuel Aaron, Alan F. Blackwell, and Pamela 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.

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

[3] CSound. http://csound.com, January 2019.

[4] Emmanuel Favreau, Michel Fingerhut, Olivier Koechlin, Patrick Potacsek, Miller Puckette, and Robert Rowe. Software developments for the 4x real-time system. In *International Computer Music Conference*, 1986.

[5] David Zicarelli. How I Learned to Love a Program That Does Nothing. *Computer Music Journal*, 26(4):44–51, 2002.

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

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

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

[9] Yann Orlarey, Dominique Fober, and Stephane Letz. FAUST: an Efficient Functional Approach to DSP Programming. In *New Computational Paradigms for Computer Music*, 2009.

[10] Christopher Alex McLean. *Artist-Programmers and Programing Languages for the Arts*. PhD thesis, University of London, 2011.

[11] Andre Rauber Du Bois and Rodrigo Geraldo Ribeiro. Hmusic: A domain specific language for music programming and live coding. In Marcelo Queiroz and Anna Xambó Sedó, editors, *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 381–386, Porto Alegre, Brazil, June 2019. UFRGS.

[12] Glasgow Haskell Compiler. https://www.haskell.org/ghc/, January 2019.

[13] Sonic Pi Tool. https://github.com/lpil/sonic-pi-tool, January 2019.

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

[15] Simon Peyton 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.

[16] Charles Roberts, Matthew K. Wright, and JoAnn Kuchera-Morin. Music Programming in Gibber. In *ICMC*, 2015.

[17] Thor Magnusson. The IXI Lang: A SuperCollider Parasite for Live Coding. In *International Computer Music Conference*, 2011.

[18] LilyPond. http://lilypond.org/, January 2019.

[19] ABC Notation. http://abcnotation.com/, January 2019.

[20] Alex McLean. Making Programming Languages to Dance to: Live Coding with Tidal. In *FARM 2014*. ACM, 2014.

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

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

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

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

[25] André Rauber Du Bois and Rodrigo Geraldo Ribeiro. A domain specific language for drum beat programming. In *Proceedings of the Brazilian Symposium on Computer Music*, 2017.

[26] Samuel Aaron, Dominic Orchard, and Alan 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.

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

[28] Simon Thompson. *The Craft of Functional Programming*. Addison-Wesley, 2011.

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