Demo: Synthesis-Enabled Live Coding on the Web

Kat Pompermayer, Catherine Ji, Hannah Macias, Mark Santolucito
Barnard College, Columbia University
3009 Broadway
NYC, NY 10027
klp2148@barnard.edu, cmj2194@barnard.edu, hcm2142@barnard.edu, msantolu@barnard.edu

Abstract

Live Coding is a performance practice characterized by the act of programming in real-time to generate media. Live Coding allows artists to explore the intersection of their media of choice and computational structure in a performance setting. One of the main challenges in making Live Coding a more inclusive and accessible artistic practice is the overhead of learning to program, both in general, and in the language specific to the live coding environment. We present a demo of a tool for synthesis-enabled Live Coding. Our tool allows users to switch back and forth between a programming-centric live coding environment as well as a graphical interface that synthesizes code.

Introduction

Core to the ethos of Live Coding is to always "show your code" (TOPLAP 2020). However, where does this leave new prospective Live Coders with no background in programming? Is the only way to practice the art of Live Coding to first study programming independently of the Live Coding context?

To this end, we introduce our tool, a synthesis-enabled Live Coding environment on the web, which allows users to gradually ease into the practice of Live Coding. Our goal is to keep to the Live Code mantra of "show your code", such that the code for the generated sound is always visible - without demanding that the user write every line of that code themselves. To do this, we leverage *program synthesis*, the process of automatic code construction from user provided specifications.

Program synthesis is the task of the automatically generating programs based on some user provided specification. In our tool, we specifically use programming-by-example (Myers 1986), where the users provides examples of the intended behavior of the code, and we generate code that matches that pattern. Rather than manually typing examples, we allow users to provide examples through interactions with a graphical interface. As users interact with the graphical interface, code is continuously synthesized that mirrors the functionality described by the user through the interface, allowing the user to switch to a programming-focused live code style at any time.

In order to generate synthesized functions, our system relies on using queries composed in the SyGuS (Syntax Guided-Synthesis) language (Alur et al. 2013). A SyGuS solver takes specifications in the SyGuS language and produces functions that satisfy said specifications. We use SyGuS to produce our synthesized functions which project drum patterns on the GUI. The synthesis pipeline can be described through the following steps. First we gather user input from the GUI or live coding interface; the user input is then translated into array data; that array data is transformed into SyGuS constraints; a SyGuS solver uses these constraints to generate a function; and the function is then embedded into our DSL.

The contributions of this demo paper are to:

1. Provide a user-level description of the synthesis-enabled live coding tool;
2. Introduce a new set of features for synthesis-enabled live coding, including a new domains-specific language, support of duration values, and in-browser recording;
3. Present our tool, open-source, with a live demo available[1].

Related Work

There has been a number of efforts to make Live Coding more accessible. Starting with Sonic Pi (Aaron 2016) and continuing with TidalCycles (McLean and Wiggins 2010), these performance environments support domain specific-languages (DSLs), which is a critical part of making live coding more accessible to new programmers (Aaron and Blackwell 2013).

The idea of a web-based live coding environment has also been explored (Ogborn et al. 2017; Roberts et al. 2015). One of the major benefits noted in prior implementations of browser-based live coding is the ability for users to explore live coding without the need to install tools locally on their computer.

[1]https://github.com/Barnard-PL-Labs/SequencerLiveCoding

Enabling live-coding with more multimodal interfaces than simply text-editing also appears in a number of tools (Hempel, Lubin, and Chugh 2019; McNutt and Chugh 2021; Hashimoto 2021). However, these multimodal interface capabilities (e.g. projection boxes (Lerner 2020)) have remained largely syntactic - for example giving easier access to manipulate constants. Synthesis of code from user-provided specifications falls outside the scope of most existing live coding environments.

The most closely related work is that of (Santolucito 2021), off of which we build. However the prototype described in (Santolucito 2021) lacks a DSL, which is critical for the usability of the live coding language. It is also limited in its expressive capacity, as it only allows specifying the volume of a sample, but not the playback duration. There is also no built-in recording functionality, or the ability to switch interface modes between the live coding interface or the drum rack GUI.

System Overview

Our live coding environment runs on the web and uses JavaScript as the live coding language. The live coding model takes inspiration from a pure functional language setting - at each time step the state is reevaluated with the given code. The state defines what should be played at every time step. Time is quantized into 16 steps, following a common subdivision value for many step-sequencers. This allows our tool to subdivide time in sixteen indices; each index represents a 16th note in a musical measure. Given this uniform division of time, arrays are chosen to represent the intermediate form between the GUI data and text-representation. Six sample tracks are available to the user. These six tracks each represent different drums in a drum kit. Our implementation includes three different tom drums, one snare, one kick, and one hi-hat.

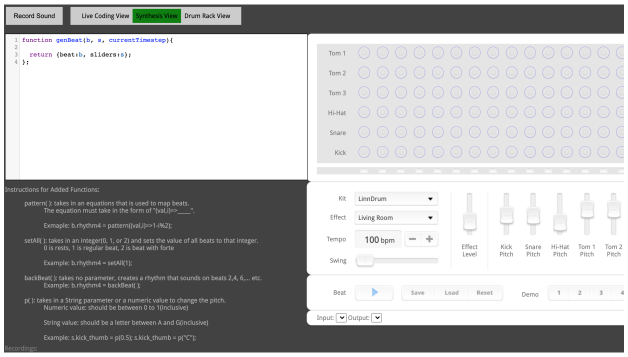
Users can write code in the live code window from scratch, or use the graphical interface to get started. By clicking on buttons on the drum rack view, initial code will be generated through program synthesis that corresponds to the recorded GUI interactions. This code can then be manually adjusted, allowing for an easier point of entry to live coding. As the manual edits to the code are made, the GUI will update as well - in this way, we have a bi-directional connection between the code and the GUI. Environments that allow for a similar interaction between code and GUI include Threnoscope (Magnusson 2014) and Glisp (Hashimoto 2021).

We believe the live-coding feature is useful in our tool because it allows the user to create beat-patterns in the GUI by leveraging the expressive power of code. This is more efficient than individually clicking each drum on the interface.

View Options in the Interface

Our tool supports three different ways to interact with the interface. We plan to use these three different views in the future to study how users leverage synthesis in a live coding environment.

The default setting is the Synthesis View, as shown in



```
function genBeat(b, s, currentTimestep){

  return {beat:b, sliders:s};
};
```

Instructions for Added Functions:

pattern( ): takes in an equations that is used to map beats.
The equation must take in the form of "(val,i)=>____".

Exmaple: b.rhythm4 = pattern((val,i)=>1-i%2);

setAll( ): takes in an integer(0, 1, or 2) and sets the value of all beats to that integer.
0 is rests, 1 is regular beat, 2 is beat with forte

Example: b.rhythm4 = setAll(1);

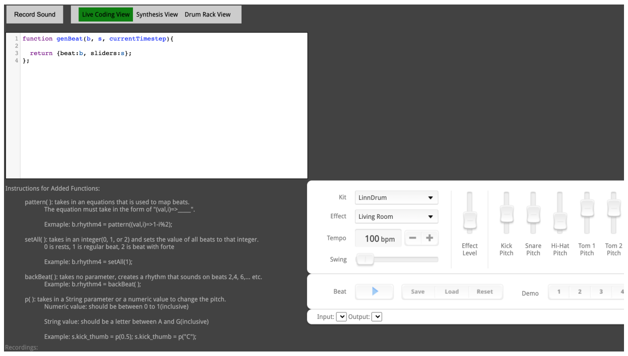
backBeat( ): takes no parameter, creates a rhythm that sounds on beats 2,4, 6,... etc.
Example: b.rhythm4 = backBeat( );

p( ): takes in a String parameter or a numeric value to change the pitch.
Numeric value: should be between 0 to 1(inclusive)

String value: should be a letter between A and G(inclusive)

Example: s.kick_thumb = p(0.5); s.kick_thumb = p("C");

Figure 1: Synthesis View



```
function genBeat(b, s, currentTimestep){

  return {beat:b, sliders:s};
};
```

Instructions for Added Functions:

pattern( ): takes in an equations that is used to map beats.
The equation must take in the form of "(val,i)=>____".

Exmaple: b.rhythm4 = pattern((val,i)=>1-i%2);

setAll( ): takes in an integer(0, 1, or 2) and sets the value of all beats to that integer.
0 is rests, 1 is regular beat, 2 is beat with forte

Example: b.rhythm4 = setAll(1);

backBeat( ): takes no parameter, creates a rhythm that sounds on beats 2,4, 6,... etc.
Example: b.rhythm4 = backBeat( );

p( ): takes in a String parameter or a numeric value to change the pitch.
Numeric value: should be between 0 to 1(inclusive)

String value: should be a letter between A and G(inclusive)

Example: s.kick_thumb = p(0.5); s.kick_thumb = p("C");

Figure 2: Live Coding View

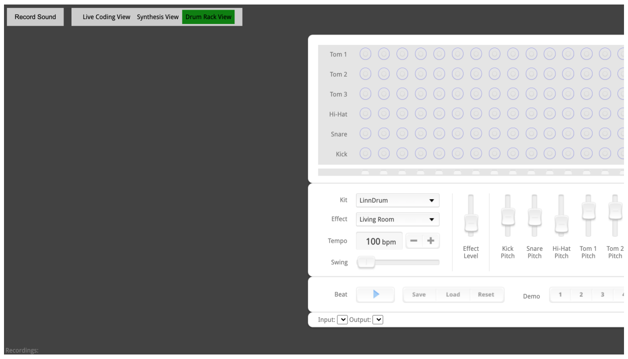


Figure 3: Drum Rack View

Fig. 1, which is the most fully featured mode. The Synthesis View contains a live-coding editor, which supports JavaScript syntax and our implemented musical library functions. Synthesis View also contains a virtual drum rack, which resembles a physical musical sequencer (a device that can record, store, distort and playback sound samples). This view allows the user to interact with both the live coding interface and the drum rack interface at the same time. The Synthesis View requires us to have implemented program synthesis to enable and maintain a semantic connection between the state of the drum rack GUI and the coding window.

A second interaction mode is the Live Coding View shown in Fig. 2, where the drum rack interface is not made available to the user. In this mode, the user must solely rely on the live coding interface to manipulate the state of the beat. Program synthesis functionality is not required here because the drum rack interface has no corresponding code representation that is visible to the user.

Finally, we have the Drum Rack View shown in Fig. 3, there is no option for live coding the beat patterns. The user manually selects the volume and duration of each beat. Synthesis is not needed in this view because there is no option to display synthesized functions in the live coding interface.

Volume Specifications

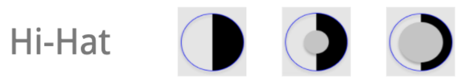


Figure 4: Volume values on the interface, showing a silent, half loudness, and full loudness note.

For each track in the interface, we have a length 16 array that defines the rhythm of the that track. At each index in the array, we can have a 0 (silence), 1 (half loudness), or 2 (full loudness). The volume values are represented on the interface using grey dots in the center of a drum beat, as shown in Fig. 4. The state of the volume array can be manipulated in one of two ways. First, the user can write code in the live coding window that manipulates the volume values. Second, the user can click directly on the drum rack GUI to change increment the volume. In the case that the user changes the state of the array through the GUI, we use program synthesis to generate code that matches the updated array state. The synthesis procedure is described in depth in (Santolucito 2021).

Duration Specifications

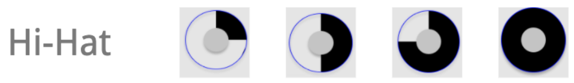


Figure 5: Duration Values on the Interface, showing a Quarter, Half, Dotted Quarter, and Whole Note

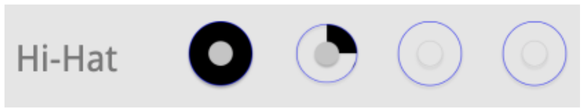


Figure 6: Overlapping Sounds: A Whole Note followed by a Quarter Note

Similar to the model for volume data, our tools allows the user to manipulate the duration of playback for each beat. The model and interaction design follows that of the volume data; for each track in the interface we have a length 16 array that corresponds to duration data. Valid duration values range from 0-4, where: 0 is silence, 1 is a quarter note, 2 is a half note, 3 is a dotted quarter note, and 4 is a whole note as shown in Fig. 5.

With the ability to specify durations for sample playback, our tool supports more complex patterns, including overlapping samples. For example, in Fig. 6, we show a 4 beat pattern. This pattern plays back the sample (Hi-Hat in this case) for a whole note duration at time index 0 (sounding the sample over 4 time indices), followed immediately by a quarter note at index 1 (which produces sound for 1 time index).

Program synthesis also supports synthesis of functions for generating duration data that matches the pattern entered through the graphical interface. This is accomplished by synthesizing two functions in a single SyGuS query. One function represents volume data, and the second represents duration data for a given track (Alur et al. 2013). This approach allows our program to run as time-efficiently as possible.

Slider Functions

In addition to the track rhythm editor, our tool also supports pitch manipulation of samples with a set of sliders. As with the track rhythms, users can manipulate pitch either through the graphical interface (moving the sliders with the mouse), or through code. When the sliders are changed through the interface, corresponding code is synthesized to match the updated state. It is important to note that the synthesis for slider values does not yet support any "automation". So far, we can only generate code that corresponds to a static value of a slider - rather than tracking the motion of a slider over time.

Internal DSL

The core of our Live Coding model is to manipulate a dictionary containing the state of the beat at every time step. At every time step, the code in the live code window is re-evaluated to generate a new beat, which is played back at that timestep. Since the dictionary containing the beat is a JavaScript object, users can simply write plain JavaScript code to live code in our language. However, to make Live Coding with our tool a more pleasant experience, we provide, additionally, a series of library functions. As these functions follow the syntax of the host language (JavaScript), we defined them as an internal DSL (a set of library functions). Our DSL is designed to facilitate the Live

Coding experience of the web application, and in the future we hope leverage these functions to implement an external DSL (adding custom syntax). In this way, users can still choose to only write vanilla JavaScript, or if they prefer, use the library functions to make the live coding experience slightly less verbose. The following section discusses the functions we provide and a short description. This documentation is also provided within the tool itself.

1. **`pattern()`** is a specialization of the `.map()` in JavaScript syntax for our context. Instead of typing out the complete syntax for mapping values of an array at each index based on the equation, the user can simply type pattern() and the equation within the parentheses. This is a special DSL function as it is supported by our synthesis algorithm. This means the user can type out this function by hand, or it can be generated by our synthesis algorithm as the user interacts with the graphical interface.
2. **`setAll()`** is a replacement for the `array.fill()` function. It instantiates an array with the same values at each index, checking that the input value is within the range of [0,2].
3. **`p()`** is a function that uses dynamic typing to accept both numeric and string values. When given a number, the function sets the pitch of the specified instrument accordingly. As for the string inputs, accepted values are the 7 notes within the letter notation for music (A - G), with each letter mapping to a numeric value. This design choice is an attempt to make the language more approachable for those with a background in Western music.
4. **`backBeat()`** We found that the alternating rhythm of the beat+rest is a common beat pattern used in our own performance practice with our tool. Thus, the backBeat function was created to assist and expedite the users' performance. In practice, the function returns an array with 1's at its even indices and 0's at its odd indices.

As of now, the library functions and generated code rely on single-letter variables `b.` and `s.`, which correspond to slider and beat control from synthesis abbreviated to increase efficiency of the Live Coding experience.

Recording Functionality

In addition to music creation functionality, our tool allows users to record their playback in-browser. When clicked, the "Record Sound" button turns red and starts the recording. To pause the recording, the user can click the button a second time and the button will turn gray again. To hear the most recently recorded sound, the user can press the "Play last recording" button. The "Save to recordings list" button saves the beat to the recordings list of the current web browser session. Once on the recordings list, the audio has full controls allowing for playing, pausing, and skipping through the recording. The three dots on the right of the recording allow the user to save the .wav file to their local disk.

The user could either choose to start recording before engaging with their interface for more of a live performance recording or they can create their beat first and then press the blue play button in the lower right side of the screen to play the beat before recording. The second option allows for the user to hear the beat they created and ensure they are satisfied before recording.

We plan to use this recording functionality to gain an understanding of which of the three interfaces (synthesis view, drum rack view, or live code view) users prefer. We plan to collect metadata analytics on which view yields more user recordings as a proxy for (one type of) preference.

Conclusions

Our demo is available online and the code is made open-source at https://github.com/Barnard-PL-Labs/SequencerLiveCoding. Our next steps are to use this infrastructure to begin user studies to gain an understanding of how synthesis impacts the live coding experience. We believe a synthesis-enabled live coding environment can invite more performers to explore live coding, even those without a strong programming background. We also hope that synthesis-enabled live coding can also give experienced live coders more freedom to explore the "the skillful extemporisation of algorithm as an expressive/impressive display of mental dexterity" and spend less time on "the glorification of the typing interface" (TOPLAP 2020).

One key open question is how we can allow the programming language to capture more complex musical patterns, while maintaining the ability for synthesis to assist new programmers. A key aspect of live coding is to use the language to explore algorithmic complexity that would not be easy to achieve with a point-and-click interface. However, our tool as it is currently implemented does not allow the programming language to exceed the expressive capabilities of the drum track GUI interface. Another area of exploration is the performance experience of synthesis-guided live coding. We are looking into ways to log the usage frequency of different interfaces, various types of clicks, and typed characters. Using collected info we will be able to gauge how users interact with the tool and determine what users find natural and intuitive about creating music through the practice of Live Coding.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant CCF-2105208, as well as partially supported the Summer Research Institute at Barnard College.

References

Aaron, S. 2016. Sonic Pi–performance in education, technology and art. *International Journal of Performance Arts and Digital Media* 12(2): 171–178.

Aaron, S.; and Blackwell, A. F. 2013. From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages. In *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling and Design*, FARM '13. New York, NY, USA: Association for Computing Machinery. ISBN 9781450323864. doi:10.1145/2505341.2505346. URL https://doi.org/10.1145/2505341.2505346.

Alur, R.; Bodik, R.; Juniwal, G.; Martin, M. M.; Raghothaman, M.; Seshia, S. A.; Singh, R.; Solar-Lezama, A.; Torlak, E.; and Udupa, A. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, 1–8. IEEE.

Hashimoto, B. 2021. Glisp: A Lisp-based Design Tool Bridging Graphic Design and Computational Arts. URL https://github.com/baku89/glisp#a-lisp-based-design-tool-bridging-graphic-design-and-computational-arts.

Hempel, B.; Lubin, J.; and Chugh, R. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, 281–292.

Lerner, S. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, 1–7. New York, NY, USA: Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376494. URL https://doi.org/10.1145/3313831.3376494.

Magnusson, T. 2014. Improvising with the Threnoscope: Integrating Code, Hardware, GUI, Network, and Graphic Scores. In *NIME*, 19–22.

McLean, A.; and Wiggins, G. 2010. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*.

McNutt, A. M.; and Chugh, R. 2021. Integrated Visualization Editing via Parameterized Declarative Templates. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21. New York, NY, USA: Association for Computing Machinery. ISBN 9781450380966. doi:10.1145/3411764.3445356. URL https://doi.org/10.1145/3411764.3445356.

Myers, B. A. 1986. Visual programming, programming by example, and program visualization: a taxonomy. *ACM sigchi bulletin* 17(4): 59–66.

Ogborn, D.; Beverley, J.; del Angel, L. N.; Tsabary, E.; and McLean, A. 2017. Estuary: Browser-based collaborative projectional live coding of musical patterns. In *International Conference on Live Coding (ICLC) 2017*.

Roberts, C.; Wakefield, G.; Wright, M.; and Kuchera-Morin, J. 2015. Designing musical instruments for the browser. *Computer Music Journal* 39(1): 27–40.

Santolucito, M. 2021. Human-in-the-loop Program Synthesis for Live Coding. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design, FARM@ICFP 2021*. ACM. To Appear.

TOPLAP. 2020. TOPLAP draft manifesto. URL https://toplap.org/wiki/ManifestoDraft.