[Demo Abstract] littleBits Synth Kit as a physically-embodied, domain specific functional programming language

James Noble Timothy Jones

Victoria University of Wellington

kjx,tim@ecs.vuw.ac.nz

Abstract

littleBits (littleBits.cc) is an open-source hardware library of pre-assembled analogue components that can be easily assembled into circuits, disassembled, reassembled, and re-used. In this demo, we consider littleBits — and the littleBits synth kit in particular — as a physically-embodied domain specific functional programming language, and how littleBits circuits can be considered as monadic programs.

1. Introduction

littleBits (littleBits.cc) is an open-source hardware library of pre-assembled analogue components that can be easily assembled into circuits, disassembled, reassembled, and re-used [1]. Designed to inspire and teach basic electrics and electronics to school-aged children (and adults without a technical background) littlebits modules clip directly onto each other. littleBits users can build a wide range circuits and devices with *"no programming, no wiring, no soldering"* [2] — even extending to a "Cloud Module" offering a connection to the internet, under the slogan "*yup. no programming here either* [sic]" [4].

The littleBits system comes packaged as a number of kits: "Base", "Premium" and "Deluxe" kits with 10, 14, and 18 modules respectively; and a series of booster kits containing lights, triggers, touch sensors, and wireless transceivers. littleBits have recently introduced special purpose kits in conjunction with third party organisations, notably a "Space Kit" designed in conjunction with NASA, and a "Synth Kit" designed in conjunction with KORG that contains the key components of an analogue modular music synthesizer.

Figure 1 shows the one of the simplest circuits in the littleBits synth kit — indeed, the simplest littleBits circuit that can actually make any sound. This circuit is composed of three simple modules — a power module on the left, an oscillator module in the centre, and a speaker module on the right. The power module accepts power from a nine volt battery (or a 9V guitar pedal mains adapter) and provides that power to "downstream" modules — as seen in the figure, littleBits circuits flow in a particular direction, and all modules are oriented so that this flow is left to right.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

FARM'14, Sep 06 - September 06 2014, Gothenburg, Sweden.

Copyright is held by the owner/author(s).

ACM 978-1-4503-3039-8/14/09.
http://dx.doi.org/10.1145/2633638.2633639

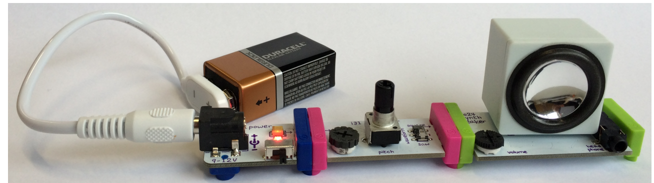

Figure 1. A simple littleBits Synth Kit circuit. From left to right, the three modules are a power source, an oscillator, and a speaker

In spite of littleBits' marketing slogans, in this demo, we analyse littleBits —- and the littleBits Synth Kit in particular — as a live physically-embodied monadic functional domain specific programming language. If building littleBits circuits is programming, then performing music with the littleBits Synth Kit (configuring modules to construct an analogue music synthesizer, and then producing sounds with that synthesizer) can be considered as a music performance by live programming — a.k.a. "livecoding" [7] — especially as the circuit construction typically occurs simultaneously with sound production.

2. SynthKit as an Embodied Functional DSL

In this section we show how building SynthKit circuits can be considered as programming in a monadic functional DSL, by modeling the littleBits SynthKit in Haskell. The linear nature of littleBits composition is mapped to the monadic bind, with secondary inputs and outputs manually managed by the programmer. The underlying monad is a standard state monad over a directed circuit graph.

The littleBits type LB is a opaque datatype with an accompanying $ClipState$ kind that tags the type, indicating whether the circuit's input on the left is open (O), or clipped into a power source (P). Like the real littleBits, the output end on the right of any circuit is always open. The partially applied type constructor is a $Functor$ for either of the two states.

$$\textbf{data } ClipState = O \mid P$$
$$\textbf{data } LB\ (s :: ClipState)\ a$$
$$\textbf{instance } Functor\ (LB\ s)$$

An open circuit can have a battery clipped into its input, which provides it with power and closes off the input.

$$battery :: LB\ O\ a \to LB\ P\ a$$

Any two littleBits sequences may be clipped together, where the sequence on the right must have its input open. The resulting sequence has the same clip state as the sequence on the left, as that is also the left end of the new circuit.

$$clip :: LB\ s\ a \to LB\ O\ b \to LB\ s\ b$$

When $s \sim O$, the type of *clip* matches the monadic operator $(\gg)$ for the monad $LB\ O$. We can implement the required functionality of $(\ggg=)$ by extracting the value in the computation on the left, applying it to the bound function, and then clipping the original and resulting circuits together.

$$\textbf{instance}\ Monad\ (LB\ O)\ \textbf{where}\ (\gg) = clip$$

$LB\ P$ is not a monad (or an applicative functor), as there is no composition between its instances to represent the bind operation.

The simplest form of littleBits module is the 'Wire', which connects two circuits without doing anything in between. In the physical embodiment, it provides some slack between circuits. As a component in the DSL, it's useful when you need a circuit but you don't want to do anything – like the real Wire component, it just gives you a little slack.

$$wire :: LB\ O\ ()$$
$$wire = return\ ()$$

The remaining modules are opaque implementations that add their corresponding circuits to the underlying state. Purely linear modules just result in (), like *wire* above, but modules with secondary outputs that branch from the linear path of the circuit take an open circuit and return a powered form of the same circuit. For instance, the keyboard module provides a secondary trigger output, so the corresponding function in the DSL takes an open circuit to plug into that output.

$$keyboard :: Key \to LB\ O\ a \to LB\ O\ (LB\ P\ a)$$

The resulting value of type $LB\ P\ a$ represents the same circuit as the parameter to *keyboard*, but this new form is now powered by the secondary output of the keyboard. When no output is desired, *wire* can be given as the parameter and the return result ignored.

The filter module takes a secondary input that represents frequency. In the DSL, the corresponding function takes a circuit to plug into this input. The value in the input circuit is threaded through into the resulting circuit.

$$filter :: Knob \to Knob \to LB\ s\ a \to LB\ O\ a$$

With physical littleBits, the input circuit need not be powered, in which case it simply has no effect. We could choose to insist that the input be powered by replacing s with P.

As secondary outputs return the newly powered circuit, that circuit can be plugged back in to another module further down.

```
example :: LB O ()
example = do
  trigger ← keyboard D wire
  oscillator 20 40
  filter 80 50 trigger
  speaker
```

The $LB\ s$ monad and accompanying functions forms a DSL for *building* littleBits circuits, not for the circuits themselves (which might be better served by Arrows of voltages). Once a circuit is built, it can be rendered and run as an IO action.

$$play :: LB\ s\ a \to IO\ a$$

Note that the circuit does not necessarily need to be powered before attempting to play it, as there may be other circuits attached as secondary inputs that have power of their own.

3. Livecoding with littleBits

If building and configuring circuits with littleBits can be considered as a form of embodied, tangible, programming, then performing music "live" with littleBits can be considered as a form of livecoding — performance programming to produce music [3, 5, 7] — or in this case both *building* and *playing* synthesizers as a live performance. In this section we describe our practice livecoding littleBits, and compare and contrast with typically textual livecoding (inasmuch as typical livecoding practice can be supposed to exist).

This section in particular draws on the first author's experience livecoding/performing littleBits with "Selective Yellow", an experimental improvisation duo of indeterminate orthography drawing on New Zealand's heritage of experimental music practice [6, 9] that seeks to recreate (electronically) all the worst excesses of free jazz with all the enthusiasm of antisocial teenagers meeting their first MOS6851. Selective Yellow is still a relatively young project, probably only Grade 2 as evaluated by Nilson [8].

Livecoding with littleBits involves two main activities that are tightly interleaved in a performance, first building the circuits by clipping modules together, and second "playing" the resulting synthesizer by turning the shafts, thumbwheels, switches, the "keys" on the keyboard module to actually generate sound. Generally a performance — or rather the portion of the performance improvised upon littleBits — starts with the smallest possible sound-generating circuit, typically the single unmodulated oscillator in figure 1. Once the littleBits are assembled (and the speaker module's output patched into the sound system) we can manipulate the oscillator's pitch and output waveform. Depending on the context of the improvisation, the possibilities of such a straightforward sound generator will be more or less quickly exhausted, at which point the performer will disassemble the circuit, insert one or more additional modules (a second oscillator, a filter, or perhaps a keyboard or sequencer module) and then continue playing the resulting circuit. In this overall pattern, littleBits livecoding is similar to some textual livecoding, where performers typically start with a single texture and build a more complex improvisation over time.

4. Conclusion

In this demo we have described the littleBits KORG Synth Kit, described how it is played (or programmed). We have argued the Synth Kit can be considered an embodied functional programming language, and presented a monadic Haskell DSL model of the Synth Kit to that end, and have begun to situate our practice performing with the Synth Kit as livecoding in that programming language.

Acknowledgements

Thanks to Chris Wilson, the other half of Selective Yellow.

References

[1] A. Bdeir. Electronics as material: littleBits. In *Proc. Tangible and Embedded Interaction (TEI)*, pages 397–400, 2009.
[2] A. Bdeir. littleBits, big ambitions! `http://littlebits.cc/littlebits-big-ambitions`, Apr. 2013.
[3] A. Blackwell, A. McLean, J. Noble, and J. Rohrhuber. Collaboration and learning through live coding (Dagstuhl Seminar 13382). *Dagstuhl Reports*, 3(9):130–168, 2014.
[4] littleBits. Sneak peek: The cloud block. `http://littlebits.cc/cloud`, May 2014.
[5] T. Magnusson. Herding cats: Observing live coding in the wild. *Computer Music Journal*, 38(1):8–16, 2014.
[6] D. McKinnon. Centripetal, centrifugal: electroacoustic music. In G. Keam and T. Mitchell, editors, *HOME, LAND and SEA: Situating music in Aotearoa New Zealand*, pages 234–244. Pearson, 2011.
[7] A. McLean, J. Rohrhuber, and N. Collins. Special issue on live coding. *Computer Music Journal*, 38(1), 2014.
[8] C. Nilson. Live coding practice. In *New Interfaces for Musical Expression (NIME)*, 2007.
[9] B. Russell, editor. *Erewhon Calling: Experimental Sound in New Zealand.* The Audio Foundation and CMR, 2012.