Funky Grooves: Declarative Programming of Full-Fledged Musical Applications

Henrik Nilsson[1(✉)] and Guerric Chupin[2]

[1] School of Computer Science, University of Nottingham, Nottingham, UK
nhn@cs.nott.ac.uk

[2] ENSTA ParisTech, Palaiseau, France
guerric.chupin@ensta-paristech.fr

Abstract. There are many systems and languages for music that essentially are declarative, often following the synchronous dataflow paradigm. As these tools, however, are mainly aimed at artists, their application focus tends to be narrow and their usefulness as general purpose tools for developing musical applications limited, at least if one desires to stay declarative. This paper demonstrates that Functional Reactive Programming (FRP) in combination with Reactive Values and Relations (RVR) is one way of addressing this gap. The former, in the synchronous dataflow tradition, aligns with the temporal and declarative nature of music, while the latter allows declarative interfacing with external components as needed for full-fledged musical applications. The paper is a case study around the development of an interactive cellular automaton for composing groove-based music.

Keywords: Functional reactive programming · Reactive values and relations · Synchronous dataflow · Hybrid systems · Music

1 Introduction

Time, simultaneity, and synchronisation are all inherent aspects of music. Further, there is much that is declarative about music, such as musical notation and many underpinning aspects of music theory. This suggests that a time-aware, declarative paradigm like synchronous dataflow [5] might be a good fit for musical applications. Indeed, there are numerous successful examples of languages and systems targeting music that broadly fall into that category, such as CSound[1], Max/MSP[2], and Pure Data[3] just to mention three.

However, systems like these primarily target artists and are not in themselves general purpose languages. It may be possible to extend them to support novel applications, but this usually involves non-declarative programming and working

[1] http://www.csounds.com/.
[2] https://cycling74.com/products/max/.
[3] https://puredata.info/.

Y. Lierler and W. Taha (Eds.): PADL 2017, LNCS 10137, pp. 163–172, 2017.
DOI: 10.1007/978-3-319-51676-9_11

around limitations such as lack of support for complex data structures [7, p. 170] or difficulties to express dynamically changing behaviour [7, p. 156][1].

With this application paper, we aim to demonstrate that Functional Reactive Programming (FRP) [8,13] in combination with Reactive Values and Relations (RVR) [15] is a viable and compelling approach to developing full-fledged musical applications in a declarative style, and, by extension, other kinds of interactive applications where time and simultaneity are central. To cite Berry [2]:

> From the points of view of modeling and programming, there is actually not much difference between programming an airplane or an electronic orchestra.

A more detailed account of this work is available as a technical report [12].

FRP combines the full power of polymorphic functional programming with synchronous dataflow, thus catering for the aforementioned temporal aspects while not being restricted by being tied to any specific application domain. Its suitability for musical applications has been demonstrated a number of times. For example, it constitutes an integral part of the computer music system Euterpea[4], which supports a broad range of musical applications [10], and it has been used for implementing modular synthesizers [9].

Generally, though, the core logic is only one aspect of a modern, compelling software application. In particular, musical applications usually require sophisticated, tailored GUIs and musical I/O, such as audio or MIDI. In practice, such requirements necessitate interfacing with large, complex, and often platform-specific imperative frameworks. In contrast to earlier work [9], we do consider external interfacing here: RVR was developed specifically to meet that need in a declarative manner.

The paper constitutes a case study of the development of a medium-sized musical application inspired by the reacTogon [4], an interactive (hardware) cellular automaton for groove-based music. The FRP system used is Yampa [13]. To challenge our frameworks, we have adapted and extended the basic idea of the reacTogon considerably to create a useful and flexible application that fits into a contemporary studio setting. Through an overview of the developed application and highlights of techniques and code fragments, we hope to convince the reader that our approach works in practice for real applications and has many merits. The source code for the application is publicly available on GitLab[5].

2 Background

2.1 Time in Music

Change over time is an inherent aspect of music. Further, at least when considered at some level of abstraction, such as a musical score or from the perspective of music theory, music exhibits both discrete-time, and continuous-time aspects

[4] http://www.euterpea.com/.

[5] https://gitlab.com/chupin/arpeggigon.

[6, p. 127]. In music theory, this is referred to as *striated* and *smooth* time, a distinction usually attributed to the composer Pierre Boulez [3]. For example, the notes in a musical score begin at discrete points in time. On the other hand, *crescendo* is the gradual increase of the loudness, *ritardando* is the gradual decrease of the tempo, and *portamento* is the gradual change of the pitch from one note to another. Contemporary electronic musical genres provide many other examples of gradual change as an integral part of the music, such as smooth filter sweeps or rhythmic changes of the volume.

Of course, there are many more aspects of time in music than discrete vs. continuous [6, pp. 123–130]. However, for musical applications, support for developing mixed discrete- and continuous-time systems, often referred to as *hybrid systems*, is a good baseline.

2.2 Functional Reactive Programming and Yampa

Functional Reactive Programming (FRP) [8] is a declarative approach to implementing reactive applications centred around programming with time-varying values in the synchronous dataflow tradition [5]. In this paper, we are using the arrows-based [11] FRP system Yampa [13]. It is realised as an embedding in Haskell and it supports hybrid systems whose structure may change over time. Thus, as discussed in Sect. 2.1, it is a good fit for musical applications. Further, the arrows-based programming model is close to the visual "boxes and arrows" approach. This also goes well with musical applications, as evidenced by systems like Max/MSP and similar. We outline some of the basic aspects of Yampa in the following for the benefit of readers not familiar with it. A more in-depth account can be found in e.g. the accompanying technical report [12].

Yampa is based on two central concepts: *signals* and *signal functions*. A signal is a function from time to values of some type:

$$Signal\ \alpha \approx Time \rightarrow \alpha$$

Time is (notionally) continuous, represented as a non-negative real number. (We will return to discrete time shortly.) The type parameter α specifies the type of values *carried* by the signal. A *signal function* is a function from *Signal* to *Signal*:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$$

When a value of type $SF\ \alpha\ \beta$ is applied to an input signal of type $Signal\ \alpha$, it produces an output signal of type $Signal\ \beta$. Signal functions are *first class entities* in Yampa. Signals, however, are not: they only exist indirectly through the notion of signal function.

Programming in Yampa consists of defining signal functions compositionally using Yampa's library of primitive signal functions and a set of combinators. Some central arrow combinators are *arr* that lifts an ordinary function to a stateless signal function, serial composition $\ggg$, parallel composition &&&, and the fixed point combinator *loop*. Figure 1 illustrates these combinators pictorially. In practice, Paterson's arrow notation [14] is often used to facilitate writing

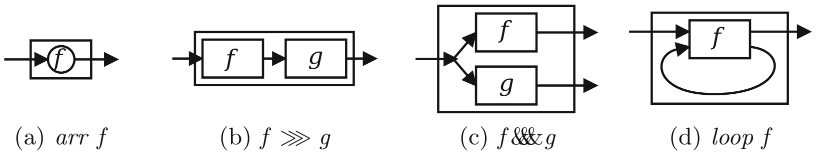


Fig. 1. Basic signal function combinators.

arrow code. It is a variation of Haskell's **do**-notation and essentially allows diagrams to be described textually by naming the arrows.

The *Event* type models discrete-time signals:

$$\textbf{data}\ \mathit{Event}\ a = \mathit{NoEvent} \mid \mathit{Event}\ a$$

A signal function whose output signal is of type *Event T* for some type *T* is called an *event source*. The value carried by an event occurrence may be used to convey information about the occurrence.

A family of *switching* primitives enable the system structure to change in response to events. The simplest such primitive is *switch*:

$$\mathit{switch} :: \mathit{SF}\ a\ (b, \mathit{Event}\ c) \to (c \to \mathit{SF}\ a\ b) \to \mathit{SF}\ a\ b$$

Once the switching event occurs, *switch* applies its second argument to the value carried by the event and switches into the resulting signal function. Yampa also includes *parallel* switching constructs that maintain *dynamic collections* of signal functions connected in parallel [13].

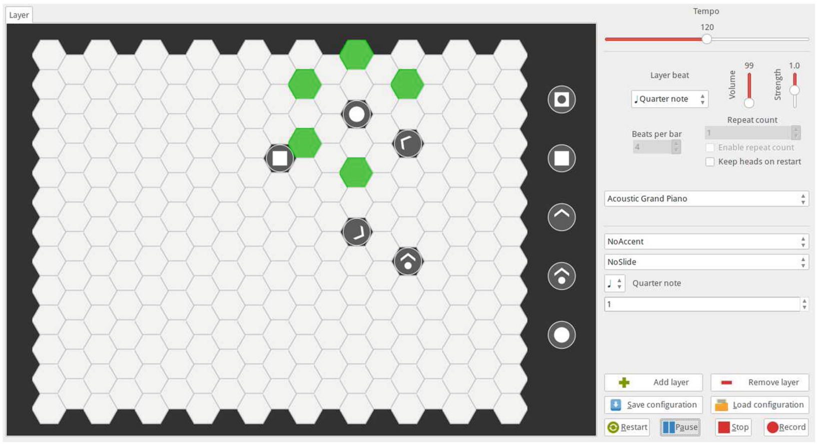
2.3 Reactive Values and Relations

A Reactive Value (RV) [15] is a typed mutable value with access rights and change notification. RVs provide a light-weight and *uniform* interface to GUI widgets and other external components such as files and network devices. Each entity is represented as a collection of RVs, each of which encloses an individual property. RVs can be transformed and combined using a range of combinators, including lifting of pure functions and lenses.

Reactive Relations (RR) specify how RVs are related *separately* from their definitions. An RR may be uni- or bi-directional. Once RVs have been related, changes will be propagated automatically among them to ensure that the stated relation is respected.

3 The Arpeggigon

Our application is called *Arpeggigon*, from *arpeggio* and *hexagon*. It was inspired by Mark Burton's hardware reacTogon: a "chain reactive performance



Tempo 120
Layer beat ♩Quarter note
Volume 99
Strength 1.0
Repeat count 1
Beats per bar 4
☐ Enable repeat count
☐ Keep heads on restart
Acoustic Grand Piano
NoAccent
NoSlide
♩ Quarter note
1

Fig. 2. The Arpeggigon (Color figure online)

arpeggiator" [4]. However, we have expanded considerably upon the basic idea to create a software application we believe is both genuinely useful in a contemporary studio setting and a credible test case for our approach.

3.1 The reacTogon

Central to the design of the reacTogon is the Harmonic Table[6]: a way to arrange musical notes on a hexagonal grid. The various directions correspond to different musically meaningful intervals. For example, each step along the vertical axis corresponds to a perfect fifth. The reacTogon uses this layout to implement a cellular automaton. See Fig. 2 for our adaptation of the idea. *Tokens* of a few different kinds are placed on the grid, at most one token per cell. These tokens govern how *play heads* move around the grid, as well as the initial position and direction of the play heads. When a play head hits a token, the kind of token determines what happens next. First, for most tokens, a note corresponding to the position of the token is played. Second, either the direction of the play head is changed, it is split into new play heads, or it is absorbed. Thus, arpeggiated chords or other sequences of notes are described. These can further be transposed in response to playing a keyboard, allowing the reacTogon to be performed.

3.2 Features and Architecture

Our Arpeggigon is a software realization of the reacTogon concept. The main features our Arpeggigon provides over the reacTogon are:

– Multiple layers: one or more cellular automata run in parallel. Layers can be added, removed, and edited dynamically through a tabbed GUI.

[6] https://en.wikipedia.org/wiki/Harmonic_table_note_layout.

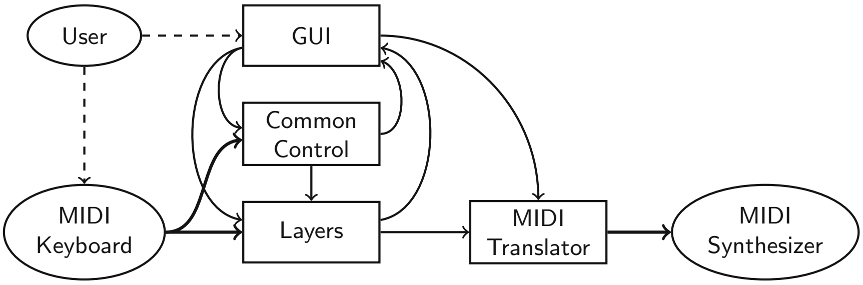


Fig. 3. The Arpeggigon architecture

- Extended attributes for tokens, such as note length, accent, and slide.
- Per-cell repeat count for local modification of the topology of the grid.
- MIDI integration.
- Saving and loading of configurations.

Figure 2 shows a screenshot. Dynamic addition and removal of layers means that both the core logic of the application and the GUI must support structural changes while the application is running. Note the different kinds of tokens to the right of the grid. They can be dragged and dropped onto the grid to configure a layer, even while the Arpeggigon is running. The play heads are coloured green.

Figure 3 illustrates the architecture of the Arpeggigon. The rectangles represent the main system components. The thin arrows represent internal communication, the thick ones MIDI I/O, and the dashed ones user interaction.

GUI is the graphical user interface. It includes a model of the state of global parameters, such as the overall system tempo, and the current configuration of each layer. Common Control is responsible for system-wide aspects, such as generating a global clock (reflecting the system tempo) that keeps the layers synchronised. Layers is the instances of the actual automata, each generating notes. MIDI Translator translates high-level internal note events and control signals into low-level MIDI messages, merging and serialising the output from all layers.

GUI communicates the current system configuration to Common Control and Layers. Note that this data is time-varying as the user can change the configuration any time. Layers needs to communicate the positions of the play heads back to GUI for animation purposes. This is thus also a time-varying signal.

4 Implementation

4.1 Layers

At its core, each layer of the Arpeggigon is a cellular automaton that advances one step per layer beat. Its semantics is embodied by a transition function:

$$\begin{aligned} &\mathit{advanceHeads} :: \mathit{Board} \rightarrow \mathit{BeatNo} \rightarrow \mathit{RelPitch} \rightarrow \mathit{Strength} \rightarrow [\mathit{PlayHead}] \\ &\qquad \rightarrow ([\mathit{PlayHead}], [\mathit{Note}]) \end{aligned}$$

In essence, given the current configuration of tokens on the hexagonal grid, henceforth the *board*, it maps the state of the play heads (position, direction, and a repeat counter) to an updated play head state and a list of notes to be played at this beat. The number of play heads may change as a play head may be split or absorbed. The remaining parameters give the current transposition of the layer, the strength with which notes should be played, and the beat number within a bar allowing specific notes in a bar to be accented (played stronger).

Using the *scanl*-like Yampa function *accumBy*, *advanceHeads* is readily lifted into an event-processing signal function:

$$\begin{aligned}&\mathit{automaton} :: [\mathit{PlayHead}] \rightarrow \mathit{SF}\ (\mathit{Board}, \mathit{DynamicLayerCtrl}, \mathit{Event}\ \mathit{BeatNo})\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{Event}\ [\mathit{Note}], [\mathit{PlayHead}])\end{aligned}$$

The static parameter is the initial state of the play heads. The first of the three input signals carries the current configuration of the board, originating from GUI (Fig. 3). The second carries a record of dynamic control parameters for the layer, including transposition, play strength, and the length of a layer beat, originating from GUI and MIDI Keyboard. These two are continuous-time signals, reflecting the fact that the configuration of the board can change and a key be struck on the MIDI keyboard at any time, not just at a beat. The third is the discrete-time layer beat clock, from Common Control, carrying the beat number within a bar. The output signals are the notes to be played, to be sent to MIDI Translator, and the state of the play heads for animation purposes, to be sent back to GUI. Note the close correspondence to the architecture in Fig. 3.

4.2 Synchronisation

As an example of turning Yampa's continuous-time capabilities to musical applications, consider automating gradual tempo changes. Imagine two sliders to set a fast and a slow tempo, a button to select between them, and a further slider to set the rate at which the tempo should change. The following signal function derives a smoothly changing tempo from these controls, regulated to within 0.1 bpm of the desired tempo. Note the feedback (enabled by **rec**):

```
smoothTempo :: Tempo → SF (Bool, Tempo, Tempo, Rate) Tempo
smoothTempo tempo0 = proc (select1, tempo1, tempo2, rate) → do
  rec
    let desiredTempo = if select1 then tempo1 else tempo2
        diff         = desiredTempo − currentTempo
        rate'        = if      diff > 0.1  then rate
                       else if diff < −0.1 then − rate
                       else                     0
    currentTempo ← arr (+tempo0) ⋘ integral ⤙ rate'
  returnA ⤙ currentTempo
```

4.3 GUI and Interaction

The GUI of the Arpeggigon is written using the cross-platform widget toolkit GTK+. The Arpeggigon does not generate any audio by itself; it needs to be connected to an external, MIDI-capable hardware or software synthesizer. MIDI I/O is handled by the JACK Audio Connection Kit.

All code for interfacing with the external world is structured using reactive values and relations (RVR). Much of this code is of course monadic (in the IO monad). However, as it is mostly concerned with creating and interconnecting interface entities, the code has a fairly declarative reading as a sequence of entity definitions and specifications of how they are related.

As a case in point, consider the following code for the system tempo slider:

```
globalSettings :: IO (VBox, ReactiveFieldReadWrite IO Int)
globalSettings = do
  globalSettingsBox ← vBoxNew False 10
  tempoAdj          ← adjustmentNew 120 40 200 1 1 1
  tempoLabel        ← labelNew (Just "Tempo")
  boxPackStart globalSettingsBox tempoLabel PackNatural 0
  tempoScale        ← hScaleNew tempoAdj
  boxPackStart globalSettingsBox tempoScale PackNatural 0
  scaleSetDigits tempoScale 0
  let tempoRV =
    bijection (floor, fromIntegral) `liftRW` scaleValueReactive tempoScale
  return (globalSettingsBox, tempoRV)
```

In essence, this code defines a box, a label, and a slider, and visually relates them by placing the last two inside the box. This is all standard GTK+. A read/write, integer-valued reactive value (RV) is finally defined and related to the real-valued value of the slider: *scaleValueReactive* associates a slider with an RV, while *liftRW* derives a new RV from an existing one by specifying two conversion functions, one for reading and one for writing.

Finally, the RVR part and the Yampa part of the Arpeggigon are connected by the following function:

```
yampaReactiveDual ::
  a → SF a b → IO (ReactiveFieldWrite IO a, ReactiveFieldRead IO b)
```

This creates two reactive values: one for the input and one for the output of the signal function. After writing a value to the input, the corresponding output at that point in time can be read.

5 Conclusions

This paper demonstrated how Functional Reactive Programming in combination with Reactive Values and Relations can be used to develop a realistic, non-trivial musical application. On the whole, we found that these two frameworks together were very well suited for this task. The performance was good, including critical aspects like jitter, without much effort so far having been spent on optimisation. Heap usage and overall memory footprint was modest. See the accompanying technical report for details [12]. Further, as most of the techniques we demonstrated are not limited to a musical context, we suggest that this is a good approach for programming time-aware, interactive applications in general.

Acknowledgments. The authors would like to thank Ivan Perez and Henning Thielemann for support and advice with the reactive libraries and the Haskell JACK bindings respectively, Michel Mauny for co-supervising the second author's summer internship with the Functional Programming Laboratory in Nottingham, and François Pessaux and anonymous reviewers for helpful feedback.

References

1. Baudart, G., Mandel, L., Pouzet, M.: Programming mixed music in ReactiveML. In: 1st Workshop on Functional Art, Music, Modeling and Design (FARM), Boston, USA, pp. 11–22. ACM, September 2013
2. Berry, G.: Formally unifying modeling and design for embedded systems - a personal view. In: Margaria, T., Steffen, B. (eds.) ISoLA 2016. LNCS, vol. 9953, pp. 134–149. Springer, Heidelberg (2016). doi:10.1007/978-3-319-47169-3_11
3. Boulez, P.: Penser la musique aujourd'hui. Gallimard, Paris (1964)
4. Burton, M.: The reacTogon: a chain reactive performance arpeggiator (2007). https://www.youtube.com/watch?v=AklKy2NDpqs
5. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for programming synchronous systems. In: 14th Symposium on Principles of Programming Languages (POPL). ACM, New York (1987)
6. Cont, A.: Antescofo: anticipatory synchronization and control of interactive parameters in computer music. In: International Computer Music Conference (ICMC), Belfast, Ireland, pp. 33–40, August 2008
7. Cont, A., Anticipation, M.M.: From the time of music to music of time. Ph.D. thesis. University of California San Diego (UCSD) and University of Pierre et Marie Curie (Paris VI) (2008)
8. Elliott, C., Hudak, P.: Functional reactive animation. In: 2nd International Conference on Functional Programming (ICFP), pp. 163–173, June 1997
9. Giorgidze, G., Nilsson, H.: Switched-On Yampa. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 282–298. Springer, Heidelberg (2007). doi:10.1007/978-3-540-77442-6_19
10. Hudak, P., Quick, D., Santolucito, M., Winograd-Cort, D.: Real-time interactive music in Haskell. In: 3rd International Workshop on Functional Art, Music, Modelling and Design (FARM), Vancouver, BC, Canada, pp. 15–16. ACM, September 2015

11. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**, 67–111 (2000)
12. Nilsson, H., Chupin, G.: The Arpeggigon: Declarative programming of a full-fledged musical application. Technical report, November 2016. http://eprints.nottingham.ac.uk/38657
13. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Haskell Workshop, Pittsburgh, PA, USA, pp. 51–64. ACM, October 2002
14. Paterson, R.: A new notation for arrows. In: International Conference on Functional Programming (ICFP), Firenze, Italy, pp. 229–240, September 2001
15. Perez, I., Nilsson, H.: Bridging the GUI gap with reactive values and relations. In: 8th ACM SIGPLAN Symposium on Haskell, Vancouver, Canada, pp. 47–58. ACM (2015)