The Arpeggigon: Declarative Programming of A Full-Fledged Musical Application

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. It illustrates the interplay between FRP and RVR as well as programming techniques and examples generally useful for musical, and other time-aware, interactive applications.

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 [6] 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[3], Max/MSP[4], and Pure Data[5] 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

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

around limitations such as lack of support for complex data structures [8, p.170] or difficulties to express dynamically changing behaviour [8, p.156][1].

With this technical report, we aim to demonstrate that Functional Reactive Programming (FRP) [9,14] in combination with Reactive Values and Relations (RVR) [17] 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 short version of this work has been accepted to PADL 2017 [13].

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[6], which supports a broad range of musical applications [11], and it has been used for implementing modular synthesizers [10].

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[7]. In practice, such requirements necessitate interfacing with large, complex, and often platform-specific imperative frameworks. In contrast to earlier work [10], we do consider external interfacing here: RVR was developed specifically to meet that need in a declarative manner. Another practical concern is cross-platform deployment. While there is no one solution to that problem, FRP has been used to develop interactive applications (video games) that run on both PCs and mobile platforms [16], and RVR facilitates cross-platform deployment further.

The paper constitutes a case study of the development of a medium-sized musical application inspired by the reacTogon [5], an interactive (hardware) cellular automaton for groove-based music. The FRP system used is Yampa [14]. 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. Our goal is not to compare our approach with alternative approaches in any detail, or attempt to argue that ours is "better" than the alternatives: what works well is very context-dependent and rarely completely objective. Instead, through an overview of the developed application as well as highlights of interesting techniques and code fragments, we hope to convince the reader that our approach works in practice for real applications and has many merits. Our specific contributions are:

- We show how FRP (specifically Yampa) and RVR together allow time-aware interactive applications with appropriate GUIs to be developed declaratively.

[6] `http://www.euterpea.com/`

[7] Musical Instrument Digital Interface: `https://en.wikipedia.org/wiki/MIDI`

– We discuss declarative programming techniques and concrete code examples that are generally useful for musical, and similar time-aware, applications.
– We present a full-fledged, somewhat unusual, musical application that we believe is interesting and useful in its own right.

The source code for the application is publicly available on GitLab[8].

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 [7, 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.

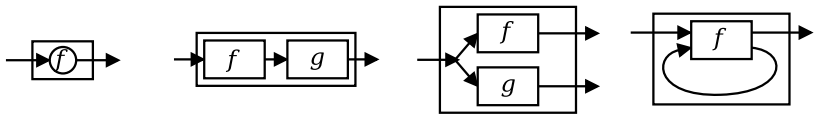
Of course, there are many more aspects of time in music than discrete vs. continuous [7, pp.123-130]. However, for musical applications, support for developing mixed discrete- and continuous-time systems, often referred to as *hybrid systems*, is a good baseline.

2.2 Functional Reactive Programming and Yampa

Functional Reactive Programming (FRP) [9] is a declarative approach to implementing reactive applications centred around programming with time-varying values in the synchronous dataflow tradition [6]. FRP has evolved in many directions and into many different systems. In this paper, we are using the arrows-based [12] FRP system Yampa [14]. 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, Yampa being arrows-based means that the 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 summarise the key Yampa features as needed for this paper in the following.

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

[8] `https://gitlab.com/chupin/arpeggigon`



(a) *arr f* (b) $f \ggg g$ (c) $f \&\&\& g$ (d) *loop f*

Fig. 1. Basic signal function combinators.

$$Signal\ \alpha \approx Time \to \alpha$$

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*:

$$SF\ \alpha\ \beta \approx Signal\ \alpha \to Signal\ \beta$$

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.

In order to ensure that signal functions are executable, they are required to be *causal*: The output of a signal function at time t is uniquely determined by the input signal on the interval $[0, t]$. If a signal function is such that the output at time t only depends on the input at the very same time instant t, it is called *stateless*. Otherwise it is *stateful*.

Composing Signal Functions. 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*. In Yampa:

$$\begin{aligned}
&arr &&:: (a \to b) \to SF\ a\ b\\
&(\ggg) &&:: SF\ a\ b \to SF\ b\ c \to SF\ a\ c\\
&(\&\&\&) &&:: SF\ a\ b \to SF\ a\ c \to SF\ a\ (b, c)\\
&loop &&:: SF\ (a, c)\ (b, c) \to SF\ a\ b
\end{aligned}$$

The programming model is thus very close to "boxes and arrows", as mentioned. Figure 1 illustrates the basic combinators using this analogy.

Arrow Syntax. Paterson's arrow notation [15] simplifies writing Yampa programs as it allows signal function networks to be described directly. In particular, the notation allows signals to be named, despite signals not being first class.

As a concrete example, the following is a realisation of a sine oscillator whose frequency can be varied dynamically by a "control voltage" as might be used in

a modular synthesizer [10]. The phase ϕ at time t given an angular frequency $2\pi f$ that may vary over time is defined as

$$\phi = 2\pi \int_0^t f(\tau)\,\mathrm{d}\tau$$

Following the common convention that raising the control voltage by one unit doubles the frequency, we can transliterate into the Yampa code:

```
oscSine :: Frequency → SF CV Sample
oscSine f0 = proc cv → do
  let f = f0 * (2 ** cv)
  phi ← integral ⤙ 2 * pi * f
  returnA ⤙ sin phi
```

The keyword **proc** is analogous to the λ in λ-expressions and is used to bind the input signal to a name (here *cv*). Note that *f0* is a normal function argument and thus *static*, not time varying. The **let**-construct is used to define a signal in terms of others by pointwise application of pure functions. Application of (stateful) signal functions is illustrated by the transliteration of the above integral equation: the signal defined to the right of the arrow tail is fed into the signal function (here *integral*, and the resulting signal is bound to the name to the left of the arrow head. Finally, *returnA* defines the output signal. An optional keyword **rec** allows recursive definitions (feedback loops). Generally, if there are more than one input or output signal, tuples are used: a product of signals is isomorphic to a signal carrying a product of instantaneous values.

Events and Event Sources. The *Event* type models discrete-time signals:

```
data Event a = NoEvent | 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.

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

```
switch :: SF a (b, Event c) → (c → SF a b) → SF a b
```

The *switch* combinator switches from one subordinate signal function into another when a switching event occurs. Its first argument is the signal function that initially is active. It outputs a pair of signals. The first defines the overall output while the initial signal function is active. The second signal carries the event that will cause the switch to take place. 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 [14].

2.3 Reactive Values and Relations

A Reactive Value (RV) [17] 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.

RVs are instances of the following type classes, reflecting access rights:

$$\begin{aligned}
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{ReactiveValueRead}\ a\ b\ m\ \textbf{where}\\
&\quad \mathit{reactiveValueRead} \qquad\quad :: a \to m\ b\\
&\quad \mathit{reactiveValueOnCanRead} :: a \to m\ () \to m\ ()\\
&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{ReactiveValueWrite}\ a\ b\ m\ \textbf{where}\\
&\quad \mathit{reactiveValueWrite} :: a \to b \to m\ ()\\
&\textbf{class}\ (\mathit{ReactiveValueRead}\ a\ b\ m, \mathit{ReactiveValueWrite}\ a\ b\ m) \Rightarrow\\
&\quad \mathit{ReactiveValueReadWrite}\ a\ b\ m
\end{aligned}$$

For example, a file could be made an instance with a being a file handle, b a string, and m the IO monad.

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. A typical GUI example is a colour picker providing different ways of specifying a colour that all should be kept mutually consistent.

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 arpeggiator" [5]. 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[9]: a way to arrange musical notes on a hexagonal grid as shown in Fig. 2. The various directions correspond to different musically meaningful intervals. For example, each step along the vertical axis corresponds to a perfect fifth. Thus, if this layout is used as the basis for a musical keyboard, many runs and chords become quite easy to play. Moreover, transposition becomes very easy: playing in a different key is just a matter of playing at a different position; the fingering need not be adjusted.

[9] `https://en.wikipedia.org/wiki/Harmonic_table_note_layout`

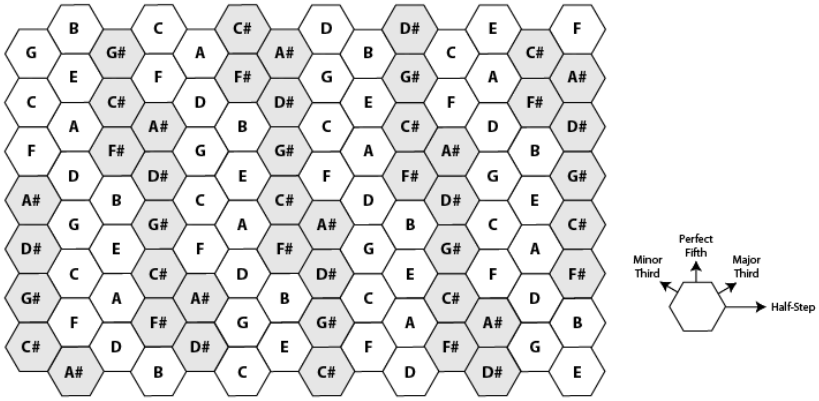


Fig. 2. The Harmonic Table

The reacTogon uses this layout to implement a cellular automaton. See Fig. 4 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

First of all, our Arpeggigon is obviously a software realization of the reacTogon concept. However, modern touchscreen interfaces can in many ways approximate the experience of directly interacting with physical objects quite well, and the Arpeggigon GUI was designed to be touchscreen-friendly. In terms of extensions, 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.
- 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.

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. The other extensions make the application genuinely useful and put further demands on the GUI and interaction with the outside world in general.

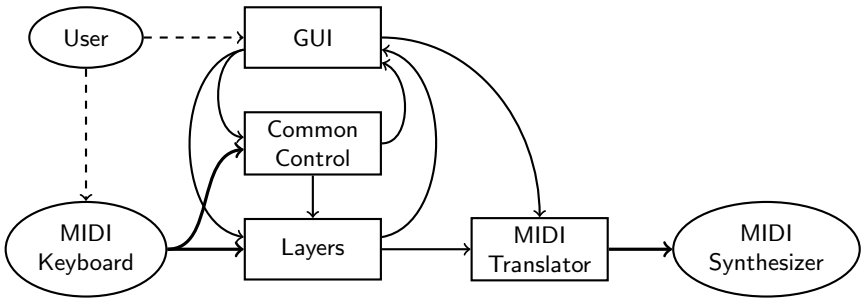


Fig. 3. The Arpeggigon architecture

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 and possibly other control signals (such as time-varying per-layer pan). 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 while the machine is running. GUI further allows different instruments to be assigned to different layers. This information is communicated to the external synthesizer via MIDI Translator. Layers needs to communicate the positions of the play heads back to GUI for animation purposes. This is thus also a time-varying signal. Finally, data from Common Control may need to be displayed. For example, in case the system tempo is synchronised with an external MIDI clock (a future feature), the current tempo needs to be communicated to GUI.

A screenshot can be seen in Fig. 4. 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.

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:

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

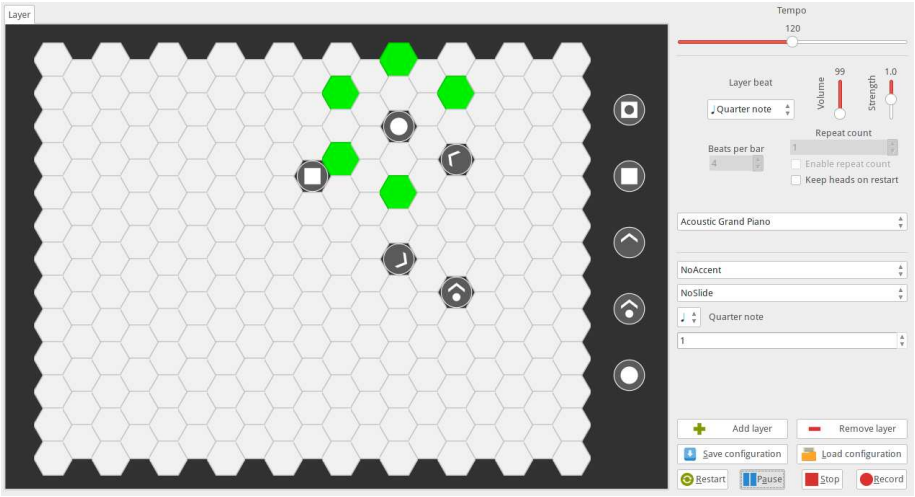


Fig. 4. The Arpeggigon

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:

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

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 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.

A complete layer has two states: running or stopped. Yampa's support for changing the system structure dynamically makes it easy to represent each by a signal function and switch between them as the run status is changed through

the GUI or in response to MIDI commands. A running layer is essentially just an instance of *automaton* along with a signal function *layerMetronome* that derives the layer beat clock from the global clock:

```
layerRunning :: StaticLayerCtrl → [PlayHead]
  → SF (Event AbsBeat, Board, LayerCtrl, Event RunStatus)
       (Event [Note], [PlayHead])
layerRunning islc iphs =
  switch (lrAux islc iphs) $ λ(rs', islc', iphs') →
    case rs' of
      Stopped  → layerStopped
      Running → layerRunning islc' iphs'

lrAux islc iphs = proc (clk, b, (slc, dlc, _), ers) → do
  lbc    ← layerMetronome islc ⪦ (clk, dlc)
  enphs ← automaton iphs       ⪦ (b, dlc, lbc)
  e      ← notYet              ⪦ fmap (λrs → (rs, slc, startHeads b)) ers
  returnA ⪦ (enphs, e)
```

Note that the initial static layer control (*islc*) only is taken into account when a layer is started, specifically to initialise the layer metronome. Also note that the switching event carries the new run status, the new values of the initial static layer control parameter, including the number of beats in a bar, and the new initial state of the play heads. The new value for the initial static layer control is obtained by sampling the continuous-time input signal *slc* at the time of a switching event. Thus, the static layer control can be modified at any time, but changes only take effect when a layer is (re)started. A stopped layer is similar, but it does not output any notes and the output list of play heads is always empty.

A useful additional feature is an option to automatically restart a layer every n bars. This can easily be arranged by adding a field *restart* :: *Maybe Int* to the static layer parameters, and, if the value is *Just* n, generate an event after counting n times the bar length plus 1 layer beats. This event is then merged with the external run status event. The additions and changes to *lrAux*, replacing line 4, are:

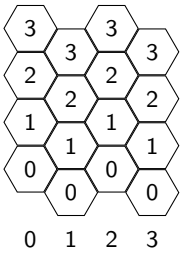
```
r ← case restart islc of
      Nothing → never
      Just n   → countTo (n * barLength + 1)
    ⪦ lbc
let ers' = ers `lMerge` (r `tag` Running)
e ← notYet ⪦ fmap (λrs → (rs, slc, startHeads b)) ers'
```

The auxiliary signal function *countTo* generates an event after having counted a specified number of events. *never* is standard Yampa and never emits any events.

Finally, a number of layers need to be active simultaneously (whether running or stopped), and it needs to be possible to add and remove layers interactively. This is accomplished using Yampa's *pSwitch* [14], a variation of *switch* that

runs a dynamic collection of signal functions in parallel. In essence, the board configuration signal from the GUI carries a list of boards, and whenever its size changes, layers are added or removed as necessary, without affecting other layers.

As to the hexagonal grid, we opted to map it onto a rectangular grid to make it possible to represent a board configuration using a standard array. The idea of the mapping is illustrated to the right. The mapping is realised by a function that computes the index of the neighbour of a cell in any of the six possible directions.



On the whole, the implementation of the layers is mostly straightforward. However, it illustrates how well suited the FRP approach is for this type of application, as well as the benefits of having a general-purpose functional language at one's disposal, including support for compound data structures like records and arrays, something that often is missing from languages and systems specifically targeting music [8, p.170].

4.2 Synchronisation

One function of Common Control (Fig. 3) is to generate the global system clock that keeps all layers synchronized. The frequency of the clock is set through the tempo slider in the GUI in quarter-note beats per minute (bpm). It is essential that the tempo can be altered smoothly, allowing an accelerando or ritardando to be performed. Yampa provides an event source *repeatedly* that could serve as a clock. However, its frequency is a static parameter, meaning that it can only be changed by switching from one instance of *repeatedly* to another. This does not allow for smooth tempo changes. Indeed, if the switching to keep up with the tempo changes becomes more frequent that the frequency of the system clock itself, there will be no time to generate a clock event before it is time to switch again, causing the performance to grind to a halt. This is not acceptable.

This problem can be addressed elegantly using the hybrid capabilities of Yampa. In Sect. 2.2 we saw how to define a sine oscillator whose frequency can be varied smoothly. At present, such an oscillator is used to derive the system clock by emitting an event on each sign change, except that the integral is restarted regularly to avoid the argument *phi* to the sine function becoming too large which would cause floating point issues. (Yampa does not do precise detection of 0-crossings, but that is not a major issue here.)

As a further example of turning Yampa's hybrid 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**):

$$\mathit{smoothTempo} :: \mathit{Tempo} \to \mathit{SF}\ (\mathit{Bool}, \mathit{Tempo}, \mathit{Tempo}, \mathit{Rate})\ \mathit{Tempo}$$
$$\mathit{smoothTempo}\ \mathit{tempo0} = \mathbf{proc}\ (\mathit{select1}, \mathit{tempo1}, \mathit{tempo2}, \mathit{rate}) \to \mathbf{do}$$

```
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 either needs to be connected to an external, MIDI-capable hardware or software synthesizer, or its output recorded to a MIDI file. MIDI I/O is handled by the JACK Audio Connection Kit. It provides extensive capabilities for handling MIDI as well as audio and is available on a wide range of platforms. Both GTK+ and JACK have well-maintained Haskell bindings.

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.

Pausing the Arpeggigon is achieved by setting the tempo to 0 when the pause button is engaged. Thanks to the possibility of combining two (or more) RVs into a new RV, this is very easy to implement:

$$tempoRV' = liftR2\ (\lambda tempo\ paused \rightarrow \textbf{if}\ paused\ \textbf{then}\ 0\ \textbf{else}\ tempo)\ tempoRV\ pauseButtonRV$$

Note that this is an equation defining *tempoRV′* once and for all: it never needs to be explicitly updated in response to the state of the pause button changing.

For another example, consider the board. Much of the associated GUI functionality, such as dragging and dropping tokens, is leveraged from a library for writing board games by Ivan Perez[10]. The library represents the board itself by an imperative array of cells. From this, we derive a read-only RV of type *Board* and a write-only RV of type [*PlayHead*], corresponding to the relevant input and output signals of a layer (*layerRunning*, Sect. 4.1). That way, changes to the configuration of the board get reflected by a time-varying input signal, and the time-varying positions of the play heads get written to the imperative array.

Interfacing to JACK provides another good use case for reactive values. In order to output MIDI messages to JACK, they have to be placed into a fixed size buffer. However, this buffer is only available from within a callback function passed to JACK when registering as a client. We use an RV consisting of a timestamped message queue to mediate: messages from the layers and other parts of the system are, after translation into low level MIDI messages by MIDI Translator (Fig. 3), simply appended to the queue as they arrive, from where the callback function progressively empties it to fill JACK's buffer.

The per-layer volume sliders are also interesting. Changing the volume results in a MIDI volume message being sent. However, we also want to *respond* to any incoming volume message from an external MIDI controller by setting the slider to reflect the new volume. This is as simple as writing to the RV of the volume slider, demonstrating how RVs enable parts of a program with little in common to be connected with a few simple and easy to understand lines of code.

Of course, any library that we wish to use in this manner needs an appropriate set of RVR bindings. For GTK+, a library of RVR bindings already exists, providing functionality like *scaleValueReactive* in the above example. But for JACK and Ivan Perez's board game library, the bindings had to be written from scratch. This part of the code is inevitably imperative in nature.

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

$$yampaReactiveDual :: a \rightarrow SF\ a\ b \rightarrow IO\ (ReactiveFieldWrite\ IO\ a, ReactiveFieldRead\ IO\ b)$$

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.

[10] `https://hackage.haskell.org/package/gtk-helpers-0.0.7`

5 Evaluation

Yampa worked very well for expressing the core logic of the Arpeggigon in a clear way. Yampa's support for structural dynamism and hybrid systems proved particularly helpful for realising musically relevant ideas with ease and clarity.

The RVR framework proved its worth for bridging the gap between the declarative core of the Arpeggigon on the one hand and imperative external libraries on the other. Most of that code is declarative in *essence*, reading as a sequence of definitions of interface entities and how they are related. The main exception is code for providing RVR bindings for external libraries where such bindings do not exist or are not appropriate. In our case, we could use existing RVR bindings for the GUI toolkit, GTK+. Writing bindings for the remaining external libraries, such as JACK, was relatively straightforward.

The only place where the facilities provided by RVR at present were a little lacking is the editing of cell and token attributes (such as per-cell repeat count and note length). When the editing focus is changed by clicking on a cell, or by switching between layers, the widgets for editing these attributes in the editing pane (Fig. 4, right hand side) need to be "reconnected" to the newly focused cell or token, and their values updated accordingly. This is possible to do, but it was not as easy as we had hoped. It is possible that *dynamic* reactive relations that are removed when no longer needed [17, Sect. 6] could help.

We have carried out some preliminary performance measurements on a mostly unoptimised version of the code base. For example, it does not exploit the fact that the input signals carrying the configuration of the board becomes constant for non-visible layers as only the currently visible layer can be edited. The measurements were done on a laptop with an Intel® Core™ i7-3537U CPU running at 2 GHz with 6 GiB of RAM.

The performance was acceptable. With the Arpeggigon stopped, the CPU load was about 4 % (of a single core). This increased to around 40 % for a very busy (lots of active play heads) 4-layer configuration. With an even busier 6-layer configuration, the load peaked at 80 %, but then the software synthesizer we used (FluidSynth) could not keep up any more, and had we used a hardware synthesizer the MIDI bus would long have been saturated due to MIDI's relatively slow speed of 31250 bits/s. Such a busy configuration is thus neither realistic nor musically very interesting (at least not to our ears!). The heap usage was modest, around 1 MiB for a typical configuration; see Fig. 5. The total memory footprint for the application was around 50 MiB.

Musically, it is crucial that the *jitter* in the generated MIDI output is small: is the timing tight, on-the-beat, or do we get a rather unsteady rhythm? We recorded the MIDI output from the Arpeggigon running a simple regular beat into a sequencer in order to quantify this. The observed jitter was below 15 ms which is not unreasonable: within one thirtieth of a beat at 120 bpm. We expect that this could be improved considerably by tagging notes with a logical time stamp, allowing MIDI messages to be placed into the output buffer close to the *intended* time as opposed to just placing them into the buffer as they arrive.

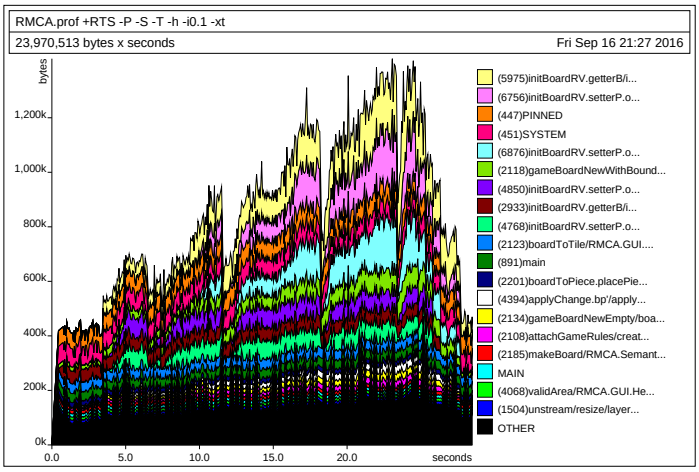


Fig. 5. Typical heap profile

6 Related Work

There are many examples of declarative languages being used for developing musical applications. We limit ourselves to a few representative examples, mostly with a strong dataflow aspect, to provide perspective on, and context for, our work. It is by no means an exhaustive review.

OpenMusic [4] is a visual programming language on top of CommonLisp. Through the visual interface, components such as score editors and players, can be interconnected to build applications ("patches") in (acyclic) dataflow style. The visual interface is not limited to this, but is general enough to allow arbitrary Lisp code to be expressed. New components can be added to the system, either by defining them through the visual interface or directly in Lisp. However, the programming style here is traditional Lisp and in many ways imperative; e.g., the interconnection mechanism is realised through callbacks. OpenMusic is a very mature and rich system geared towards musicians and composers. It is less suited for developers wanting to implement stand-alone applications.

ReactiveML is a synchronous (not dataflow) extension of OCaml in the Esterel tradition. The programming style is thus somewhat imperative. ReactiveML has been proposed for developing interactive musical applications [1]. Time is discrete, and ReactiveML's signals most closely correspond to Yampa's events: there are no (notionally) continuously varying signals. Examples such as *smoothTempo* (Sect. 4.2) could thus not be realised as directly. However, like Yampa, ReactiveML does support a dynamic system structure. As ReactiveML is impure, I/O, including GUI programming, can be accomplished in the standard imperative way. Should a more declarative approach be desired, we think abstractions similar to RVR could be beneficial also in the setting of ReactiveML.

Euterpea [11] is a computer music system embedded in Haskell. It focuses on score-level description and generation of music for subsequent playing, but it can also be used for interactive applications, and it includes an FRP-subsystem for sound synthesis. It does not provide or advocate any general-purpose approach for interfacing with external libraries, but there is a Musical User Interface (MUI) that includes GUI functionality as well as support for MIDI I/O and audio. However, as Euterpea is embedded in Haskell, it ought to be possible to use it with RVR should the provided MUI not suffice for some particular application.

Finally, Yampa itself has been used for musical applications before [10]. However, that work did not consider a systematic, declarative approach to developing GUIs or interaction more generally.

7 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. Further, as most of the techniques we demonstrated are not limited to a musical context, we conclude 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. We would also like to thank 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. Guillaume Baudart, Louis Mandel, and Marc Pouzet. Programming mixed music in ReactiveML. In *ACM SIGPLAN Workshop on Functional Art, Music, Modeling and Design (FARM '13)*, pages 11–22, Boston, USA, September 2013. ACM.
2. Gerard Berry. Formally unifying modeling and design for embedded systems — A personal view. In *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016), Part II*, number 9953 in Lecture Notes in Computer Science, pages 134–149. Springer, 2016.
3. Pierre Boulez. *Penser la musique aujourd'hui.* Gallimard, 1964.
4. Jean Bresson, Carlos Argon, and Gérard Assayag. Visual Lisp/CLOS programming in OpenMusic. *Higher Order and Symbolic Computation*, 1(22), 2009.
5. Mark Burton. The reacTogon: a chain reactive performance arpeggiator. `https://www.youtube.com/watch?v=AklKy2NDpqs`, 2007.
6. Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, New York, NY, 1987. ACM.

7. Arshia Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, pages 33–40, Belfast, Ireland, August 2008.
8. Arshia Cont. *Modeling Musical Anticipation: From the Time of Music to Music of Time*. PhD thesis, University of California San Diego (UCSD) and University of Pierre et Marie Curie (Paris VI), 2008.
9. Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of ICFP'97: International Conference on Functional Programming*, pages 163–173, June 1997.
10. George Giorgidze and Henrik Nilsson. Switched-on Yampa: Declarative programming of modular synthesizers. In *Practical Aspects of Declarative Languages (PADL) 2008*, volume 4902 of *Lecture Notes in Computer Science*, pages 282–298, San Francisco, CA, USA, January 2008. Springer-Verlag.
11. Paul Hudak, Donya Quick, Mark Santolucito, and Daniel Winograd-Cort. Real-time interactive music in Haskell. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design (FARM 2015)*, pages 15–16, Vancouver, BC, Canada, September 2015. ACM.
12. John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000.
13. Henrik Nilsson and Guerric Chupin. Funky grooves: Declarative programming of full-fledged musical applications. In *19th International Symposium on Practical Aspects of Declarative Languages (PADL 2017)*, Paris, January 2017. To appear.
14. Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, PA, USA, October 2002. ACM.
15. Ross Paterson. A new notation for arrows. In *Proceedings of the 2001 ACM SIGPLAN International Conference on Functional Programming*, pages 229–240, Firenze, Italy, September 2001.
16. Ivan Perez and Henrik Nilsson. Declarative game programming: Slides, videos, and code. `http://keera.co.uk/blog/2014/09/24/game-programming-videos-code`, September 2014.
17. Ivan Perez and Henrik Nilsson. Bridging the GUI gap with reactive values and relations. In Ben Lippmeier, editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell (Haskell'15)*, pages 47–58, Vancouver, Canada, 2015. ACM.