NNdef: Livecoding Digital Musical Instruments in SuperCollider using Functional Reactive Programming

Miguel Cerdeira Negrão
School of Technology and Management
Polytechnic Institute of Leiria
Portugal
miguel.negrao@ipleiria.pt

Abstract

The SuperCollider audio synthesis environment allows the definition of *Synths*, digital instruments which generate sound using a graph of interconnected unit generators. In SuperCollider the definition of a Synth is mostly declarative, on the other hand the logic for controlling parameters of a Synth using musical controllers is usually implemented in a different context using callbacks and explicit state.

This paper presents a different approach where functional reactive programming (FRP) is used to define the control logic of the instrument, taking inputs from musical controllers, mobile apps or graphical user interface (GUI) widgets and sending outputs to the audio graph. Both audio and FRP graphs are defined in the same context and compiled simultaneously avoiding a hard division between audio and control logic.

An FRP implementation is used in the NNdef class to enable livecoding of both audio and FRP code, with hotswap allowing an interactive workflow. Also included is a system to persist the state in the FRP network in order to save and recall the instrument at a later time.

CCS Concepts • **Applied computing → Sound and music computing**;

Keywords livecoding, functional reactive programming, digital musical instrument

Permission to make digital or hard copies of all or part 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 components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FARM '18, September 29, 2018, St. Louis, MO, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5856-9/18/09…$15.00
https://doi.org/10.1145/3242903.3242905

ACM Reference Format:
Miguel Cerdeira Negrão. 2018. NNdef: Livecoding Digital Musical Instruments in SuperCollider using Functional Reactive Programming. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '18), September 29, 2018, St. Louis, MO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3242903.3242905

1 Introduction

Algorithmic computer music sound synthesis environments enable the creation of networks of unit generators (UGens), which define an instrument or synthesizer, as well as the scheduling of sound events, using a programming language. Unit generators synthesize or transform sound and can be connected to form complex audio instruments. In a textual programming language instruments and sound events are defined using segments of code which are sometimes called "patches". *SuperCollider* is one well known example of a computer music synthesis environment with an interpreted high-level general-purpose programming language.

An important development in this area is livecoding, where snippets of code are evaluated causing changes in ongoing processes. Livecoding allows quick experimentation and incremental construction of patches, enabling a playful exploration of compositional possibilities. It also makes possible the on-stage, simultaneous creation and manipulation of patches during a live performance for an audience.

Another relevant development is the use of musical controllers for performing a digital instrument. Different sorts of devices which capture physical motion into a data stream have been used to directly control parameters of a patch. Notable examples are commercial Musical Instrument Digital Interface (MIDI) controllers, smartphone and tablet apps communicating via Open Sound Control (OSC) [20], and custom-made sensor microcontrollers and microcomputers such as the Arduino board [15] and the Raspberry Pi computer [1].

Functional Reactive Programming (FRP) deals with events which appear at unpredictable times and as such it is well suited to deal with incoming data from electronic musical controllers. An FRP network can therefore be connected to

an audio network in order to create a unified audio synthesis-processing instrument which can be played with physical controllers.

When using a callback approach the definition of the UGen graph is declarative while the control logic is not, creating a mismatch. When using a general-purpose FRP library to control an audio network it is necessary to create some boilerplate code and the two networks must be defined and managed separately.

This paper presents a specific integration of an FRP system within an audio livecoding library, implemented in the SuperCollider synthesis environment which attempts to simplify the creation of digital musical instruments. The system is interfaced through the NNdef class whose features will be explained in detail.

The specific contributions are:

- Integration between an FRP network and a constant-rate high-efficiency audio network.
- Simultaneous definition and compilation of both networks.
- Hot-swapping and persistence of the FRP network enabling livecoding digital instruments which use musical controllers.

2 SuperCollider and Livecoding Using JITLib

SuperCollider [12–14] is a programming language for real-time audio synthesis and algorithmic composition, with an efficient sound synthesis engine (*scsynth*) designed specifically for music composition and performance. Coming from the tradition of MUSIC-N, it allows the interconnection of UGens to form larger digital signal processing (DSP) networks. The SuperCollider language (*sclang*), is a general-purpose object-oriented language (OOP) with some features of functional programming. Blocks of code are evaluated in real-time by the sclang interpreter. The SuperCollider standard class library includes, besides the general purpose abstractions such as collections, powerful procedures for algorithmic creation of UGen graphs and rich abstractions for musical events (routines, patterns, etc.). The synthesis engine (*scsynth*), written in C/C++, combines UGens according to pre-made declarative descriptions, SynthDefs, instantiating them as Synths, which can be dynamically created and destroyed. scsynth is a separate process and communicates with sclang via OSC messages.

A Synth in SuperCollider consists of a network of UGens whose shape cannot be changed once it has started playing. Although this makes for a very efficient design, it limits interactivity. The *JITLib*[11, 17, 18][1] attempts to overcome this limitation by introducing *proxies*, abstract placeholders which allow seamlessly switching between different audio graph definitions. When a new Synth definition is assigned to the proxy, the previous Synth is stopped and a new one starts, usually using a cross-fade. The Ndef class[2] of JITLib associates a unique name with a proxy, enabling quick access, which is useful when livecoding.

The NNdef class[3], an extension of JITLib's Ndef, is the main focus of this paper. It is an extension of JITLib using FRP for processing incoming data streams from physical controllers and other sources.

Although SuperCollider is not a pure functional language, and is therefore not the best candidate for using FRP, given that it has a vast amount of functionality relating to computer music readily available, it was deemed a worthwhile experiment seeing what could be achieved with FRP in such a language.

The NNdef class is part of *FPLib*, a library for functional programming in SuperCollider created by the author. NNdef integrates an FRP network[4] into the audio Ugen network contained in a proxy in JITLib, with the aim of allowing the creation, through livecoding, of digital musical instruments. The audio and event networks connect at bridging ports where an outgoing event from the FRP network becomes a continuous audio/control signal[5]. Inputs to the FRP network are obtained from MIDI controllers via the Modality toolkit [4], OSC messages and GUI widgets.

The main goal when developing NNdef was enabling the livecoding of digital instruments which use musical controllers, either for live performance or for composition[6]. Often creating a digital instrument with a physical controller is a trial and error process where the instrument undergoes several iterations until arriving at a final design. The synthesis and control logic are developed iteratively, with the feedback between physical motion and sonic result being continuously evaluated, and the corresponding patch updated. Taking this into account it was desirable that the livecoding interface would allow defining the synthesis and control logic together in the same context.

It was also desirable when livecoding an instrument that its current state would not be lost when switching over to a new definition of the instrument, particularly if the instrument has modal control logic[7] which goes beyond a direct

[1] JITLib stands for Just In Time programming library.

[2] Ndef stands for node proxy definition.

[3] The name NNdef is a playful derivation of Ndef: since Ndef has one type of graph, and NNdef has two (audio and FRP), it uses two Ns.

[4] More precisely, like Ndef, NNdef can have multiple Synths and each Synth has its own associated FRP graph.

[5] In SuperCollider UGens can operate at audio-rate or at control-rate, which is slower. The later is used with low-frequency signals in order to decrease CPU usage.

[6] A broad view of what is a musical instrument should be taken. For instance, it is common for electroacoustic composers to "play" "instruments" custom made for a specific piece, recording material which is then later used to assemble the final fixed media composition. These are one-off instruments, sometimes never played again, but still they are often controlled using a musical controller.

[7] The instrument can enter different modes of operation, responding differently to input data depending on the current mode.

mapping between control values from devices and synthesis parameters. Finally, in the case of a modal instrument, it should be possible to persist the current mode of operation to disk, in order to later pick up a session in the same state.

3 FRP and Livecoding

A digital instrument can be though of as set of inputs from control devices (e.g. a MIDI surface), a set of DSP blocks that produce an audio stream, and control logic in between to allow incoming events from the physical devices to affect the state of, as well as create and destroy, the DSP blocks.

The traditional method of dealing with incoming events is through callback functions. Callback functions, although easy to define lack composability and often require explicit state manipulation. Functional Reactive Programming, or FRP, is a paradigm for programming dynamic and reactive systems using first-class composable abstractions. Most of the original work on FRP was done in the Haskell programming language with two main flavours, Classic FRP [9, 10] and Arrowized FRP [5]. In Classic FRP the two main abstractions are event streams (sequences of discrete-time event occurrences) and behaviours or signals (time-varying values). Currently there are several well-maintained libraries for FRP in Haskell such as reactive-banana [3], Yampa [5], or reflex [19].

FRP implementations typically do not allow changing code at runtime, which is sometimes referred to as *hot-swapping*, nevertheless there is ongoing research in this area. The ELM language [6] has hot-swapping capabilities [7] which are used in an interactive debugger [8] which has the stated goal of enabling interactive programming. ELM's interactive debugger also allows time-travelling, that is, the FRP graph state can be saved, rewound and replayed again. The Midair library for Haskell [16] allows hot-swapping subcomponents of the FRP graph as a first-class operation.

In textual languages hot-swapping makes possible interactive programming which is particularly well-suited for musical and audio-visual applications.

4 FRP in FPLib

FPLib is a library for functional programming in SuperCollider. Although the SuperCollider language is object-oriented it has first-class higher-order functions (closures) as well as recursion with tail-call optimization, which allows for functional programming idioms, although with some limitations[8]. FPLib implements a system similar to Haskell's type classes, but since SuperCollider is dynamically typed, whether a class is an instance of a type class is only determined at runtime. The Monoid, Functor, Applicative Functor, Traversable and Monad type classes are provided together with the associated combinator functions. The implementation mostly follows Haskell's, although for some functions (e.g. traverse when passed an empty list) types have to be provided manually as there isn't a type system available to determine them automatically. Functions in SuperCollider are not curried by default, a `curried` method was implemented to convert a non-curried function to curried.

FPLib includes an FRP library which stays close to Classic FRP. It has event streams (class EventStream) and signals (class FPSignal[9]). FPSignals are slightly different from the behaviours of Classic FRP: behaviours model any continuous time function $f : \mathbb{R} \to A$, while signals only model step functions.

The implementation of EventStream, FPSignal and child classes (corresponding to combinators such as merge, collect, hold, select and inject) was directly ported from the *reactive-web library* for the Scala programming language [2]. The interface for defining, starting and stopping an FRP network, input callback registration, reactimate functions (reacting to output events) and some combinators (e.g. `<@>` and `<*>`) were based on Haskell's reactive-banana library. The method names used tried to follow the names traditionally used for the same functionality in SuperCollider which are in some cases different from Haskell (see table 1).

Table 1. Correspondence between function names in FPLib and Haskell.

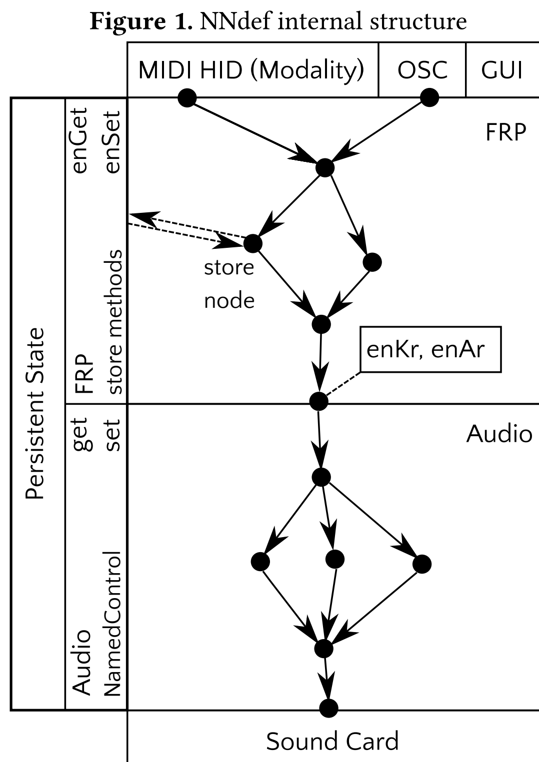
FPLib	Haskell
`select`	`filter`
`collect`	`fmap`
`inject`	`fold`
`mreduce`	`mconcat`
`|+|`	`<>`
`<%>`	`<$>`

EventStream and FPSignal both provide a Functor instance. The corresponding `collect` method returns a new object which transforms each value from the original object using a supplied function. EventStream provides a Monoid instance (`|+|` and `zero` methods) which allows merging events. FPSignal provides an Applicative Functor instance (`<*>` method) which allows applying a time-varying function to a time-varying value. It is also possible to filter an event stream using the `select` method and a predicate function. State can be explicitly held using the `inject` and `injectF` methods. With `inject` a function is supplied which given an incoming value and the current state determines the next state. The `injectF` method should be called on an event stream carrying functions. When an event arrives the function it is carrying is applied to the current state in order to determine the next state. Signals and event streams mainly interact using the `hold` method, which converts an event stream into a signal

[8] SuperCollider doesn't have pattern matching, recursive definitions, currying, or monadic I/O.

[9] SuperCollider has a single global namespace and the class name Signal was already taken.

Figure 1. NNdef internal structure



by remembering the last value, and using <@> which applies a time-varying function contained in a signal to an event stream. All functions passed to these combinators should be pure (no side-effects), nevertheless SuperCollider gives no guarantees in this regard, it is up to the user to be disciplined.

To determine side-effects to be performed on an event stream the `enOut` method (equivalent to `reactimate`) is used. This method should be called on an event stream carrying `IO` values. The `IO` class is just a wrapper for a normal function which implements the >>= and `pure` methods of the Monad type class for sequencing actions. Placing an action inside a function will delay its execution.

5 NNdef

In this section we will examine the NNdef class and its interaction with the FRP library. Similarly to Ndef, an audio network is defined with NNdef by associating a key with a function describing the interconnection of UGens. In the following example a square wave (`Pulse`) is sent to a resonant low-pass filter (`RLPF`):

```
NNdef(\x, {
  RLPF.ar(
    Pulse.ar(freq:440), //Hz
    freq: 1000, //Hz
    rq:0.1)
}).play
```

An FRP network is attached to this graph by obtaining events from an input source and sending events to a NamedControl, a UGen which receives control data from the SuperCollider language.

NNdef can receive inputs from different types of sources. For human interface devices (HID), such as pointing devices and gamepads, and MIDI based controllers, the Modality library is used. This library facilitates the acquisition of data from, and control of, commercially available controllers via the aforementioned protocols. Each physical element (e.g. fader, knob) has a representation as an object which is placed in a tree-like data structure which mimics the spatial configuration of the device. From such an object an event stream can be obtained using `enIn` and a signal with `enInSig` (initialized with the current value of the element, which is cached by Modality). Input nodes to the FRP network can also be obtained from OSC messages with a given address pattern and sclang GUI widgets.

The `enDebug` method prints incoming values to a node to the post window with a given label. This simplifies debugging the behaviour of the FRP network.

An event from the FRP network can reach the Synth's audio network by passing through a bridging node. By calling `enKr` or `enKrUni` on an FRP node, a callback is registered and a new NamedControl with the given name is created. If no name is given one is automatically assigned, in which case the created control is hidden in the NNdef GUI interface. When an event reaches this node a value is sent to the associated NamedControl either as is (`enIn`), clipped to a certain range (`enIn` with a ControlSpec[10] parameter) or mapped between ranges (`enInUni`).

When calling `enKr` on an EventSource an initial value must be supplied to be used at Synth instantiation time. Since an FPSignal already has a current value when the network is compiled, calling `enKr` on it will use that value for initializing the NamedControl.

We now go over a concrete example. In the following listing input events are obtained from the first slider of the first page of a Korg NanoKONTROL MIDI controller. The MIDI controller is accessed using the Modality library:

```
MKtl('nnkn0',"korg-nanokontrol");
```

The input is obtained as a signal, using `enInSig`. This method is used, instead of `enIn`, in order to automatically initialize the NamedControl with the value which corresponds to the current slider position. The node is then connected to the audio network with an automatic control name, and mapping values between 0 and 1 to values between 80Hz and 1000Hz exponentially. The control-rate signal created with `enKrUni` is used as the frequency of the square wave.

[10]A ControlSpec in SuperCollider represents an injective function $f : [0, 1] \to A \subset \mathbb{R}$. It is commonly used to map between two ranges, or constrain values to a given range.

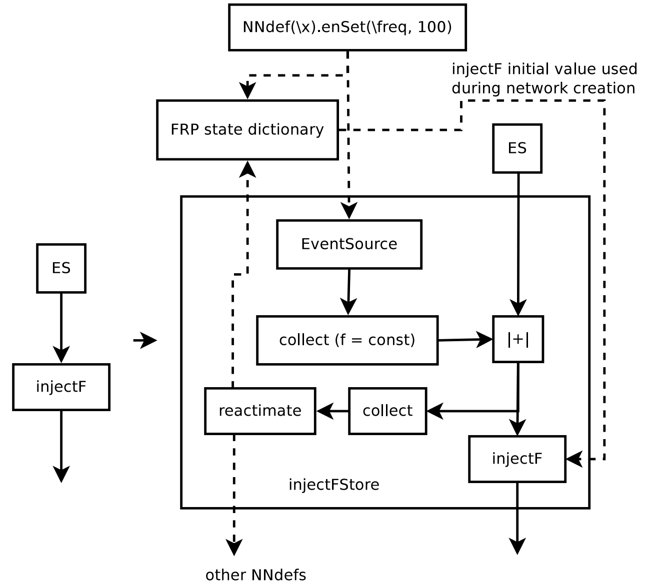
```
NNdef(\x, {
//get the dictionary of elements
  var k = MKtl('nnkn0').elementGroup;
  var page = 0;
  var column = 0;
//convert value in [0,1] range to the
//[80,1000] range exponentially
  var spec = [80,1000,\exp, 0, 80].asSpec;
//get the first slider on first page
  var freq = k[\sl][page][column]
//FRP INPUT
  .enInSig
  .enDebug("freqSlider")
//FRP OUTPUT
  .enKrUni(lag: 0.1, spec: spec);
//AUDIO
  RLPF.ar(
    Pulse.ar(freq), //Hz
    freq: 1000, //Hz
    rq:0.1)
}).play
```

5.1 Hot-swap

When the NNdef is redefined by associating a new function with the same key, for instance by re-running a modified snippet of code, the previous FRP network is destroyed (removing all registered callbacks) and a new network is automatically created. Since all functions should be pure, state is only kept in the FPSignal nodes (the current value) and in the inject nodes (the nodes which allow explicit state). If this state was erased when the FRP network was destroyed then the current state of the instrument would be lost. One consequence would be that a modal instrument would forget the current mode when being re-evaluated, reverting to the initial mode. To avoid this situation the FRP library implements *hot-swapping* which, when possible, automatically carries the state from the previous FRP network to the new one.

When redefining an NNdef, if the new graph has the same shape as the previous one, then the state of the previous graph is saved and copied to the new one. The execution of the output IO actions is implemented by merging all reactimate event sources, thus forming a single event stream carrying actions to be executed. For the purpose of saving the state, the FRP graph is traversed starting at this end node and following the parents, being careful to avoid traversing the same path twice, and taking note of the current state along the way. With this scheme, if the NNdef is redefined and only the (pure) functions passed to the combinators are changed, then the shape of the graph is the same. In this case the new functions will operate on the old state, which is saved by the hot-swap functionality. The granularity of

Figure 2. injectFStore implementation



the hot-swapping is the NNdef, it is not possible to hot-swap only a subsection of the NNdef FRP graph.

Let us consider a specific example where a single button with corresponding event stream `es` can be used to cycle forward from 0 to 9 and back to 0:

```
es
.collect({|v| {|st| (st+1).mod(10) } })
.injectF(0);
```

If the current value in `injectF` is 5 and the code is hot-swapped to

```
es
.collect({|v| {|st| (st-1).mod(10) } })
.injectF(0);
```

then on the next button press the state will be 4, with the initial value of 0 being ignored and the old value of 5 being used. It is possible to return all stateful nodes to their initial values by calling `NNdef(\x).clear` and re-evaluating again.

5.2 Persistence

The state in the FRP network can also be persisted to disk, allowing the instrument to be picked up in the same state at a later time. JITLib Ndefs are usually persisted using text, more specifically, as code which when interpreted yields the original object, and NNdef follows the same pattern. Unfortunately, the hot-swapping mechanism described above cannot usually be used for persisting the network as code, as the state stored in the nodes can be an instance of any class, including closures (the Function class), and closures which have dependencies on external values cannot be persisted as code in the SuperCollider language. It should be noted that the use of the <%> combinator with a function of two or

more arguments will automatically create a curried version of that function. This new curried version is implemented using closures, and therefore the current value of `fsig` in

```
fsig <*> ysig
```

where

```
fsig = f <%> xsig
```

is a closure and therefore cannot be persisted.

Since automatic persistence of the entire graph is not feasible, instead the state in selected nodes can be explicitly persisted using specific methods whose name ends with `Store`. Each node created with a store method is associated with a name, and its current value is updated in a dictionary in the NNdef while the FRP graph is running. Calling `asCode` on an NNdef will include one `.enSet(name, value)` instruction for each element in the dictionary. These instructions when interpreted will restore the saved state of the selected FRP nodes.

The store nodes are also useful when changing the shape of the FRP graph while livecoding, as in that case the automatic how-swap will not activate, and therefore the previous state of nodes which are present before and after reevaluation is lost. On the other hand, any state saved via store nodes will be picked up when re-evaluating an NNdef while livecoding.

Signals have a `store` method, since signals have state which corresponds to the last calculated value. In order to save the state of an NNdef the current value in specific signal nodes must be saved (usually those that are used with `<@>` and related combinators). The internal state of an `injectF` or `injectFSig` node can be persisted by switching to `injectFStore` or `injectFSigStore`. When creating a signal from an event source it might also be desirable to make the created signal node persistable in one go, in which case the `holdStore` method can be used.

In the following example the play and rewind buttons of a Korg NanoKONTROL are used to select the current octave (the state), while the two rows of buttons are used to select the current note in the twelve note equal temperament tuning. The play and rewind buttons' events are associated with functions that sum and subtract one from the current octave respectively[11].

```
var octUpES = k[\tr][\fwd]
.enIn.select(_==1)
.collect({|v| {|x| (x+1).min(3) } });

var octDownES = k[\tr][\rew]
.enIn.select(_==1)
.collect({|v| {|x| (x-1).max(-3) } });
```

The `injectFSigStore` method is used to store the current octave with persistence, also creating a signal.

[11] The code in this example would have to be surrounded by `NNdef(\x,{ ... })`.

```
var octaveSig = (octUpES |+| octDownES)
.injectFSigStore(0, \octave)
.enDebug("octave");
```

Each button of the NanoKONTROL is then associated with a note, whose frequency is computed using the current value in the octave signal. The event streams from all buttons are merged and converted to a signal with an initial value.

```
var f0 = 100, page = 0;
var twoRowsOfbuttons = k[\bt][page];
var allButtons = twoRowsOfbuttons
.flat[..12];
var freqSig = allButtons.collect({ |but,i|
  var butES = but.enIn;
  var freq0 = f0 * (2**(i/12));
  var f = { |oct| (2**oct)*freq0 };
  var freqSig = f <%> octaveSig;
  freqSig <@ butES
})
.mreduce //merge all event sources
.hold(f0); //start with f0=100Hz
```

Finally, the signal is connected to the audio network.

```
  var freq = freqSig
  .enKr(lag: 0.1, key:\freq);

  RLPF.ar( Pulse.ar(freq), 1000, 0.1 )
```

The initial value of the octave can be set (or recalled from disk) using `enSet`.

```
NNdef(\x).enSet(\octave, 2)
```

Regarding the implementation, the FPSignal `store` method creates a secondary EventStream which is merged with the FPSignal's internal event source (FPSignal is implemented using an underlying EventSource). `hold` is then called on the resulting EventStream. If the value in the state dictionary changes due to `enSet` then the secondary EventStream fires. If the original EventStream fires then the dictionary is updated via a reactimate. Finally the initial value passed to `hold` when creating the FRP network is obtained from the current value in the state dictionary. The implementation for `injectFStore` and `injectFSigStore` is similar but after the merge the methods `InjectF` or `injectFSig` are applied (see figure 2).

5.3 Inter-NNdef Communication

The persistence functionality can also be used for communication between the FRP graphs of different NNdefs. For this purpose the store method can be considered an output node of the FRP graph which can be connected to an input node of a different NNdef by calling `enIn` on the NNdef with the appropriate key:

```
NNdef(\a, {
  var k,f;
  k = MKtl('nnkn0').elementGroup;
  f = k[\sl][0][0]
  .enIn.holdStore(0, \slider1);
  SinOsc.ar(f.enKrUni(spec:\freq));
}).play;
NNdef(\b, {
  var f;
  f = NNdef(\a).enIn(\slider1);
  Saw.ar(f.enKrUni(spec:\freq));
}).play;
```

This mimics the way in which Synths belonging to different Ndefs can also send and receive audio from each other by automatic use of audio buses.

Sometimes values from control logic should affect multiple NNdefs, in that case it is useful to have one central NNdef communicating with multiple "child" NNdefs using this functionality.

5.4 Recursion

In some situations using recursion between elements of the FRP network is the most intuitive way to solve a given problem. Recursion is often used when the previous value is used to compute the current value. Unfortunately, unlike Haskell, SuperCollider does not allow recursive definitions, therefore to achieve the same result we need to "cheat".

Consider as an example the situation where one MIDI slider is controlling the frequency of an oscillator and a toggle button controls whether the the slider is in "zoom" mode. When in "zoom" mode the same slider fine-tunes the frequency within a much smaller range around the previous value. This can be achieved using one zoom signal which, is either zero if not in "zoom" mode, or the value of the slider at the moment the instrument entered "zoom" mode otherwise. We also need an (unmapped) frequency signal which is the current value of the slider if not in "zoom" mode, or the fine-tuned value otherwise. Note that these two signals will depend on each other.

First we define `zoomES` in terms of `freqSig`:

```
mult = { |a,b| a*b};
zoomES = mult
  .liftRecSampled({freqSig}, toggleES);
zoomSig = zoomES.hold(0);
```

We can then define `freqES` in terms of `zoomSig`:

```
freqES = { |zoom, slider|
  if(zoom != 0) {
    slider //linear interpolation
    .linlin(0.0,1.0,zoom-0.1,zoom+0.1)
  } { slider }
} <%> zoomSig <@> sliderES;
freqSig = freqES.hold(0.5);
```

It would be natural to use instead the definition

```
zoomES = mult <%> freqSig <@> toggleES;
```

, but this is not possible in SuperCollider since `freqSig` is not yet defined at that moment. The method `liftRecSampled` was created to circumvent this issue. It works as follows: `freqSig` is passed to `liftRecSampled` wrapped in a closure in order to delay evaluation. On an incoming value from `toggleES`, the closure is evaluated and the current value of the signal is directly extracted (something which is not allowed in pure FRP). With this "hack" it is possible to create at least some recursive definitions inside an NNdef.

6 Further Work

Currently each NNdef has a set of fixed, constantly running synths paired with corresponding FRP networks. The current work could be expanded by exploring FRP networks which create and destroy synths dynamically, for instance for generating note patterns, possibly also interacting with the SuperCollider patterns system and JITLib's Pdefs.

With the current interface the flow of data is unidirectional from the FRP network into the audio network. The flow could be made bidirectional by allowing the creation of input nodes in the FRP network which obtain their events by polling the audio output of a UGen in the audio graph.

7 Conclusion

The NNdef class demonstrates that, with some limitations, it is possible to implement FRP in a language such as SuperCollider, following quite closely an interface commonly seen in Haskell libraries. NNdef enables defining the audio and FRP graph simultaneously, which facilitates constructing the different components of an instrument with minimal distractions. Features such as automatic NamedControl names, and compact syntax for mapping values when crossing from event to audio network also aid in this regard. The presented hot-swap functionality allows livecoding the control logic of a musical instrument. An approach for persisting the state in the FRP graph was also described.

NNdef should enable a more playful experience when developing a digital instrument in SuperCollider by using a high-level declarative approach, removing some of the more tedious tasks and allowing an interactive workflow.

Acknowledgments

Part of this material is based upon work supported by the Foundation for Science and Technology Portugal under Grant No. SFRH / BD / 76694 / 2011.

References

[1] [n. d.]. Raspberry Pi - Teach, Learn, and Make with Raspberry Pi. Retrieved 2018-07-31 from https://www.raspberrypi.org/
[2] [n. d.]. Reactive-Web. Retrieved 2018-07-31 from https://github.com/nafg/reactive

[3] Heinrich Apfelmus. [n. d.]. Reactive-Banana: Library for Functional Reactive Programming (FRP). Retrieved 2018-07-31 from https://hackage.haskell.org/package/reactive-banana

[4] Marije Baalman, Till Bovermann, Alberto de Campo, and Miguel Negrão. 2014. Modality. *Int. Computer Music Conf. Proc.* 2014 (2014).

[5] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*. ACM, 7–18. https://doi.org/10.1145/871895.871897

[6] Evan Czaplicki. 2012. *Elm: Concurrent FRP for Functional GUIs*. Ph.D. Dissertation.

[7] Evan Czaplicki. 2013. Interactive Programming. Retrieved 2018-07-31 from http://elm-lang.org/blog/interactive-programming

[8] Evan Czaplicki. 2014. Elm Debugger. Retrieved 2018-07-31 from http://debug.elm-lang.org/

[9] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

[10] Conal M. Elliott. 2009. Push-Pull Functional Reactive Programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

[11] Alberto de Campo Julian Rohrhuber. 2005. Algorithms Today: Notes On Language Design for Just In Time Programming. *International Computer Music Conference* (2005), 291.

[12] James McCartney. 1996. SuperCollider, a New Real Time Synthesis Language. *Int. Computer Music Conf. Proc.* 1996 (1996).

[13] James McCartney. 1998. Continued Evolution of the SuperCollider Real Time Synthesis Environment. *Int. Computer Music Conf. Proc.* 1998 (1998).

[14] James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music J.* 26, 4 (Dec. 2002), 61–68. https://doi.org/10.1162/014892602320991383

[15] David Mellis, Massimo Banzi, David Cuartielles, and Tom Igoe. 2007. Arduino: An Open Electronic Prototyping Platform. In *Proceedings of the Conference on Human Factors in Computing*, Vol. 2007.

[16] Tom E. Murphy. 2016. A Livecoding Semantics for Functional Reactive Programming. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*. ACM, 48–53. https://doi.org/10.1145/2975980.2975986

[17] Julian Rohrhuber and Alberto de Campo. 2009. Improvising Formalisation: Conversational Programming and Live Coding. *New Computational Paradigms for Computer Music. Delatour France/Ircam-Centre Pompidou* (2009).

[18] Julian Rohrhuber and Alberto de Campo. 2011. Just in Time Programming. *The SuperCollider Book, The MIT Press* (2011).

[19] Ryan Trinkle. [n. d.]. Reflex FRP. Retrieved 2018-07-31 from https://reflex-frp.org/

[20] Matthew Wright, Adrian Freed, and others. 1997. Open Sound Control: A New Protocol for Communicating with Sound Synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, Vol. 2013. International Computer Music Association San Francisco, 10.